Implementación y uso de Bibliotecas Dinámicas

Gabriel Bonanno, Mario José Bondar, Esteban Carnuccio, Etel Elizabeth Guardia, Alfredo Portero

Departamento de Ingeniería e Investigaciones Tecnológicas, Universidad Nacional de La Matanza
Buenos Aires, Argentina
{gab.peekaboo@gmail.com, mariobondar@yahoo.com.ar, cartu32@hotmail.com, eguardia@gmail.com, acportero@yahoo.com.ar}

Resumen. El presente documento explica los diferentes pasos llevados a cabo en el sistema operativo SODIUM para poder adaptarlo a la utilización de bibliotecas dinámicas. Para poder determinar esto, se investigó las diferentes formas de administrar la memoria que posee SODIUM actualmente, como también la utilización de las diferentes secciones contenidas en el archivo ELF. Se mencionan los inconvenientes en la compilación de los archivos fuente del SODIUM, debido a que el mismo actualmente está orientado a la ejecución de procesos con bibliotecas estáticas.

Palabras Clave: ELF, SODIUM, Biblioteca Dinámica, Cargador Dinámico, Compilador de Linux.

1 Introducción

Actualmente en el sistema operativo SODIUM, los programas de usuario son link-editados de manera estática, es decir, que todos los componentes son colocados conjuntamente en un único archivo. Una de las principales limitaciones de esto es no poder utilizar bibliotecas dinámicas, imposibilitando compartir código.

Luego de realizar un estudio detallado de las distintas secciones de los archivos ELF [BON11], nos proponemos implementar bibliotecas dinámicas en SODIUM y la posibilidad que procesos puedan acceder a las mismas y utilizarlas.

Se mostrarán las distintas modificaciones que debieron llevarse a cabo en el sistema operativo SODIUM, como por ejemplo, en la compilación del mismo.

Además, se describe las modificaciones realizadas sobre los fuentes; debido que los procesos al usar bibliotecas dinámicas deben direccionar correctamente al código fuente.

Se indican además los distintos inconvenientes que se fueron presentando provocados principalmente por las adaptaciones llevadas a cabo para la utilización de bibliotecas dinámicas por parte de procesos, que son compilados de manera dinámica.

2 Objetivos

Implementar en el sistema operativo SODIUM una biblioteca dinámica, y que dos procesos puedan llamar al código de la misma.

3 Alcance

Crear y compilar una biblioteca dinámica y dos procesos, los cuales realicen una llamada a la mencionada biblioteca y pueda ser visualizado como se comparte el código de la biblioteca. Se implementará en un gestor de memoria segmentada y se dejará la documentación para gestión de memoria paginada.

4 Investigación

4.1. Cargador dinámico

Un cargador dinámico resuelve las referencias a bibliotecas dinámicas a través de la vinculación dinámica (o también llamado enlace dinámico). Gracias a este tipo de enlace se retrasa la vinculación de las rutinas de las bibliotecas dinámicas con el módulo de carga (módulo objeto del programa a ejecutar). Así, el módulo de carga contiene referencias no resueltas a otros programas. Estas referencias pueden resolverse cuando se realiza el enlace dinámico en tiempo de carga o cuando se realiza el enlace dinámico en tiempo de ejecución [STA08].

En esta sección se explicará qué acciones debe realizar un cargador dinámico para resolver las referencias a bibliotecas dinámicas tanto al momento de carga como en el momento de ejecución de un programa.

4.1.1 Resolución de referencias en tiempo de carga

A medida que el sistema crea la imagen de un proceso, va copiando lógicamente los segmentos del archivo a segmentos de memoria. Cada segmento tiene su propio set de permisos.

El fin del segmento de datos requiere un manejo especial para los datos no inicializados, que son definidos inicialmente con valor cero por el sistema.

Hay un aspecto que difiere entre la carga de un archivo ejecutable y la carga de un objeto compartido. Los segmentos de un archivo ejecutable, por lo general contienen código absoluto. Para que el proceso se ejecute correctamente, los

segmentos deben residir en la dirección virtual que se usa para construir el ejecutable, por lo cual el sistema usa el valor de p_vaddr como dirección virtual [TIS95].

Por otra parte, los segmentos de un objeto compartido contienen código independiente de la posición (PIC, por su denominación en inglés, "position-independent code") [LEV00]. Esto permite que la dirección virtual del segmento varíe de un proceso a otro, sin modificar su comportamiento. Lo que se mantiene invariable son las posiciones relativas dentro del segmento.

Vinculación implícita

A diferencia de la resolución de referencias en tiempo de ejecución, cuando esta resolución se hace en tiempo de carga no existe un STUB (encargado de resolver las referencias). En el código del archivo no hay referencias a las rutinas de una biblioteca. Al encontrar una llamada a una función de biblioteca dinámica, el linker agrega información al archivo ejecutable para indicar al sistema dónde se encuentra el código.

Entonces cuando se carga el programa en memoria se buscan en el ELF todas las llamadas a bibliotecas que necesita el programa [LEV00]. Si la biblioteca no está cargada en memoria, se sube a memoria y se le asigna un descriptor de segmentos (CS) en la GDT. Para esto, el sistema operativo debe tener una tabla donde diga qué bibliotecas están cargadas en memoria en ese momento.

Cuando se hace una llamada a una biblioteca, me fijo en la tabla del ELF si hay una referencia a la misma. De ser así, busco el CS que le fue asignado. Después busco en la tabla del ELF la referencia a la rutina que se invocó y me fijo a qué posición del código corresponde. Así, reemplazo el espacio en blanco dentro del código por el CS+desplazamiento de la rutina.

Una vez remplazados todos los espacios en blanco, se carga el programa a memoria para ser ejecutado.

A este proceso se le llama vinculación implícita [TAN99]. Se produce cuando el código de una aplicación llama a una función de una biblioteca dinámica. Cuando se compila el código fuente para el archivo ejecutable de llamada, la llamada a la función de la biblioteca dinámica genera una referencia de función externa en el código.

4.1.2 Resolución de referencias en tiempo de ejecución

Al compilar un programa con formato ELF, el compilador remplaza todas las llamadas a las rutinas externas en el código por referencias simbólicas no resueltas. Posteriormente, cuando se genera la imagen binaria del proceso se incluye dentro de la misma un código llamado stub. El stub es un fragmento de código que indica como localizar la rutina adecuada de una biblioteca [SIL05].

Los stubs son organizados en una estructura llamada tabla PLT (Procedure Linkage Table). La PLT es un conjunto de stubs, uno por cada rutina usada en la aplicación. Por cada uno de estos stubs se hace un salto a una dirección en una tabla GOT (Global Offset Table) [MUC97]. Por medio de las tablas GOT, el linker puede soportar que las bibliotecas dinámicas se puedan compartir entre varios procesos. Esto es debido a que las bibliotecas dinámicas son construidas con código PIC. La GOT contiene las direcciones absolutas para todos los datos estáticos a los cuales hace referencia en el programa. Pero también posee la dirección real de las rutinas externas, aunque esta dirección no se conoce hasta que se ejecuta la aplicación. El rol del linker en tiempo de ejecución es llenar las entradas de la tabla GOT con los valores apropiados. Entonces la GOT hace referencia directa a los símbolos de una biblioteca [TIS95], la dirección de la GOT normalmente se almacena en el registro EBX que es una dirección relativa del código que hace referencia a ella.

Cada biblioteca compartida y cada archivo ejecutable que usa bibliotecas compartidas tienen su correspondiente tabla PLT y GOT,

En resumen, cada entrada en la PLT corresponde a una rutina externa y es un salto indirecto a la tabla GOT. Así, si en tiempo de ejecución se quiere acceder a una rutina en una biblioteca, se debe acceder a su dirección real en memoria a través del salto indirecto a la entrada de la GOT correspondiente a la rutina a través de la PLT [SIL05].

Vinculación explícita

Lo descrito anteriormente es la forma básica de vinculación dinámica en tiempo de ejecución. Linux y Windows utilizan, además de esta forma básica, una adaptación de este método en la cual se invoca explícitamente en el código del programa el manejo de lazy procedure linkage. En otras palabras, por medio de instrucciones en el código se decide cuándo el linker debe realizar el enlace dinámico con las bibliotecas [TAN99]. Las instrucciones que utiliza Linux son dlopen, dlsym y dlclose. En cambio Windows utiliza loadlibrary, getprocadress y freelibrary.

4.2. Administración de memoria

Para la implementación del cargador dinámico en SODIUM fue necesario investigar acerca de las diferentes formas de administración de memoria utilizadas actualmente por los sistemas operativos. Se investigó en particular sobre los métodos de paginación y segmentación. Así mismo, se investigó acerca de la organización de la memoria en SODIUM, hoy en día, mostrando su distribución.

Se decidió trabajar sobre el método de segmentación debido a su relativa facilidad de análisis y codificación. Y porque además, SODIUM en la actualidad es más estable bajo este tipo de gestión. El método de paginación sería más fácil de llevar a cabo una vez que esté finalizado el método de segmentación.

Al utilizar segmentación, cada proceso (dinámico o estático) se carga en un segmento de memoria, y la biblioteca dinámica solicitada en otro, siempre dentro del área de memoria para procesos de usuario.

4.3. Carga dinámica

Complementando lo expuesto en el informe anterior, se profundizó en la investigación de la manera correcta de implementar la carga dinámica de bibliotecas en tiempo de ejecución y en tiempo de carga.

Por una cuestión de implementación en SODIUM, se optó por realizar la resolución de referencias en tiempo de ejecución. Debido a que si se elegía resolverlas durante la carga nos encontrábamos ante la dificultad de tener que modificar el código ya compilado para poder realizarlo.

Las referencias en tiempo de ejecución se resuelven bajo demanda. En otras palabras, se determina las direcciones virtuales de las funciones dinámicas indefinidas cuando se las ejecutan. Para poder realizar esta forma de implementación se crearon instrucciones especiales para cargar las bibliotecas dinámicas en memoria, obtener las direcciones virtuales de sus funciones, ejecutarlas y liberara el segmento de memoria que ocupa dicha biblioteca. Estas funciones especiales se deben utilizar en el código de los procesos semidinámicos. Los nombres implementados para estas directivas son similares a las utilizadas en Linux. En este trabajo práctico usamos las llamamos de la siguiente manera

- Dlopen
- Dlsym
- vFnFuncDin
- dlclose

Más adelante se explicaran con mayor detalle el funcionamiento de estas funciones.

4.4. Dependencia de biblioteca libc

Por realizarse la compilación de todos los archivos en un entorno LINUX, de manera implícita el compilador añadía la biblioteca libc.so (biblioteca estándar del lenguaje C en LINUX) a todos los archivos dinámicos. Para poder cumplir con los objetivos del trabajo, fue necesario investigar cómo quitarla de la compilación, y cómo realizar la misma lógica en SODIUM.

Fue necesario investigar de qué manera modificar los makefiles de SODIUM para poder generar los procesos como archivos semidinámicos en lugar de archivos dinámicos puros. Esto es debido a que es necesario enlazar estáticamente a los

archivos _start.o y libsodium.o (con el fin de suplir la ausencia de la libc.so), dado que estos archivos son los encargados de:

- Inicializar heap y stack del proceso
- Apuntar al comienzo del código ejecutable
- Recibir el código de retorno del main

El inconveniente encontrado fue la compilación del SODIUM. La misma se lleva a cabo a través del sistema operativo Linux, el cual asocia a los distintos programas dinámicos una biblioteca para la manipulación de dichos programas y para las bibliotecas dinámicas a las cuales hacen referencia. Por lo tanto, se modificó la compilación de los archivos para que la des-asocie, llevando a la utilización de una biblioteca similar para el manejo de los programas y bibliotecas. Lo que resulto en la implementación de la función _START, que es la primera función en invocarse al ejecutar un proceso semidinámico. Esta rutina es la encargada de realizar tareas tales como inicializar el heap y el stack del proceso, y además de apuntar correctamente al código del proceso. Esta última tarea la realiza invocando a la función main del programa dinámico.

4.5. Compilación y carga en disco virtual de SODIUM

Se modificaron distintos makefiles para compilar correctamente tanto los procesos como las bibliotecas dinámicas.

Para realizar las pruebas correspondientes, se debió crear una biblioteca dinámica con el nombre de libiblio.bin, ubicada en el directorio /usr/lib. Esta biblioteca cuenta actualmente con dos funciones: VFnCiclo y VFnSleep. Además se generaron dos procesos semidinámicos los cuales invocan a las funciones de la biblioteca dinámica. Sus nombres son Proceso1.bin y Proceso2.bin. Estos archivos están ubicados en el directorio /usr.

Para poder implementar correctamente la resolución de referencia en tiempo de ejecución se debieron crear dos PLT manualmente y no utilizar las PLT que genera automáticamente el compilador gcc. Estas dos PLT tienen el nombre pltEntrada y pltSalida. Estos dos archivos son enlazados estáticamente a los procesos y la biblioteca. Su implementación está ubicada en los distintos makefiles.

Para realizar lo anteriormente comentado se debieron modificar distintos makefiles de SODIUM. Los makefiles modificados son los que se encuentran en las siguientes ubicaciones:

- **/usr/lib:** se incluyó la compilación de la biblioteca libiblio.bin. El cual es enlazado estáticamente al archivo pltEntrada.o
- **/usr:** se agregó la compilación de los archivos ejecutables semidinámicos Proceso1.bin y Proceso2.bin. Estos archivos están enlazados dinámicamente

a la biblioteca libiblio.bin y estáticamente a los archivos _start.o, libsodium.o y pltSalida.o.

- **/solo y /build:** se modificaron los makefiles para poder cargar los procesos semidinámicos y la biblioteca dinámica en el directorio raíz en memoria de SODIUM.

Debido a diversos inconvenientes detallados en este informe se debieron modificar varias veces los pasos de la compilación de los procesos y de las bibliotecas dinámicos detallados en [BON11].

Al ejecutar el comando "ls", desde la línea de comando de SODIUM, se podrá visualizar un listado con los archivos almacenados en el directorio raíz del sistema operativo, entre los cuales estarán los tres archivos creados en los Makefiles anteriormente mencionados:

- Proceso1.bin**:** archivo dinámico
- Proceso2.bin: archivo dinámico
- Libiblio.bin: biblioteca dinámica.

4.6. Cargador dinámico

Para no vulnerar los niveles de seguridad de SODIUM se decidió implementar al cargador dinámico como parte del kernel del sistema operativo. Debido a que si la implementación se realizaba como archivo ejecutable no se podía invocar directamente a funciones ubicadas en el archivo gdt.c desde el cargador dinámico como ejecutable. Por lo tanto se determinó construirlo como funciones del archivo gdt.c y que de esta manera sea parte del kernel.

4.7. Carga de archivos ejecutables dinámicos en memoria

La carga en memoria de los archivos semidinámicos Proceso1.bin y Proceso2.bin se realiza al ejecutar cualquiera de dichos archivos desde la línea de comando. Por ejemplo escribiendo:

```
Proceso1.bin
```

Al ejecutarse, se invoca la función iFnEjecutarBinario en sodshell.c. Esta función a su vez ejecuta un fork (para duplicar el proceso padre y crear al proceso hijo) y la syscall execve (para reemplazar los datos del hijo por los valores correctos que debe tener). Cuando se refiere a proceso hijo nos estamos refiriendo al Proceso1.bin. Entonces la carga de este proceso a memoria se realiza con las funciones iFnDuplicarProceso (invocada al hacer el fork) y iFnReemplazarProceso (invocada al hacer execve) ubicadas en gdt.c.

Al intentar cargar en memoria el archivo Proceso1.bin, se descubrió que era necesario modificar la función iFnLeerCabeceraEjecutable (en el archivo gdt.c), ya que la misma tenía por dirección la posición a partir de la cual debía cargarse el archivo en formato ELF, omitiendo la cabecera del mismo. Esto funcionaba correctamente sólo para archivos estáticos, y debía adaptarse para archivos dinámicos.

Por este motivo, se buscó una manera general de cargar los procesos a memoria, en lugar de algo particular para archivos estáticos.

Finalmente, se concluyó que la manera correcta de realizar la carga era obteniendo los segmentos LOAD de la sección del Program Header del ELF.

Los archivos estáticos contaban con un único segmento LOAD, mientras que los dinámicos cuentan con al menos dos. Se determinó que la mejor manera de realizar la carga era unificando ambos segmentos LOAD en uno solo, y cargándolo en un segmento de memoria. Para realizar esto fue necesaria la creación de un archivo llamado linkerscript.lds (ubicado en /herramientas). Este archivo es usado durante la compilación de los archivos Proceso1.bin, Proceso2.bin y libiblio.bin. Este script tiene como finalidad el determinar la dirección virtual de inicio del segmento LOAD y dónde va a ser cargado en memoria, así como también las secciones que lo componen.

Además, fue necesario modificar el tamaño de la cabecera ELF en la función iFnLeerCabeceraEjecutable del archivo gdt.c, debido a que se generaban problemas al calcular el punto de carga.

4.8. Carga de biblioteca dinámica en memoria

Para realizar la carga del archivo libiblio.bin en memoria durante la ejecución de los procesos semidinámicos se decidió implementar la syscall dlopen. Esta syscall se debe invocar en el main del Proceso1.bin o del Proceso2.bin. Su forma de implementación es la siguiente.

```
Selector= dlopen("Nombre biblioteca");
```

Al ejecutarse dlopen automáticamente se invoca la función iFnCargarBiblioteca ubicada en gdt.c, encargada de cargar la biblioteca en memoria.

Un punto importante para comentar es que se determinó utilizar el registro selector ES para asociarlo al descriptor de código de la biblioteca dinámica en memoria para ubicar la biblioteca en memoria. Además este trabajo está desarrollado para cargar solamente una biblioteca dinámica a la vez en memoria. Pero realizando las modificaciones correspondientes en gdt.c se podrían cargar varias bibliotecas a la vez.

Cabe destacar que al realizar un fork en la carga de los procesos semidinámicos se le asocia en el PCB de cada proceso el valor del registro ES, para

que apunte al descriptor de segmento del código del proceso padre (en la GDT). Ver Figura 4.8.1.

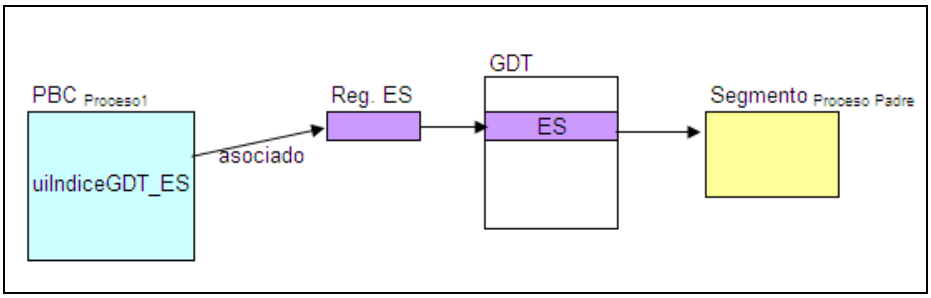

Figura 4.8.1 Asociación Registro ES-Segmento Proceso padre

Este descriptor contiene la dirección base del segmento del código del proceso padre, cuyo valor es reemplazado en la función iFnCargarBiblioteca con la dirección base del segmento donde será ubicada la biblioteca dinámica en memoria. En la figura que se muestra a continuación se podrá ver con mejor detalle esta implementación.

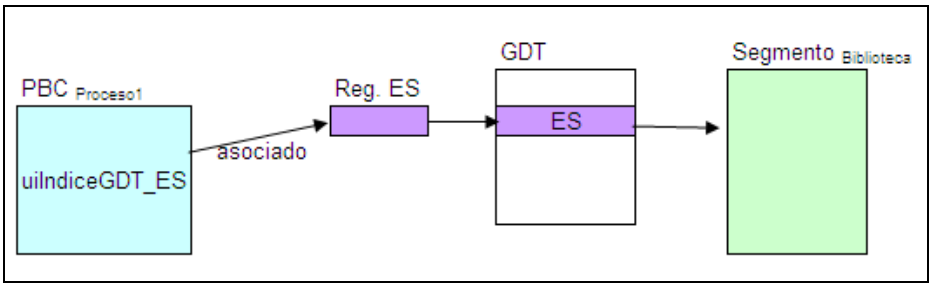

Figura 4.8.2 Asociación Registro ES-Segmento de Biblioteca

Se implementó de esta forma porque de otra manera no se podía modificar el valor asociado al registro ES correspondiente a la biblioteca en la función iFnCargarBiblioteca.

La biblioteca dinámica se ubica en memoria después del proceso que la invoca por primera vez. Esta se sitúa en un segmento de memoria diferente al que se ubicó el proceso, es decir, si el Proceso1.bin está ubicado en el segmento con dirección física 0x526974 la biblioteca se ubicará por ejemplo en el segmento con dirección 0x645896. Se realizó de esta forma porque para ubicar primero la biblioteca y después el Proceso1.bin se debía realizar la carga de la biblioteca al cargar en Proceso1.bin en memoria. Más específicamente en la función iFnReemplazarProceso. No se realizó esta forma de implementación porque no se estaría realizando la carga de la biblioteca en tiempo de ejecución sino durante la carga de Proceso1.bin. Se deja

comentado en la función iFnReemplazarProceso el método de implementación durante la carga en caso que la cátedra necesite realizar futuras investigaciones.

La forma de implementación actual de la carga de la biblioteca en tiempo de ejecución se visualiza en la siguiente figura.

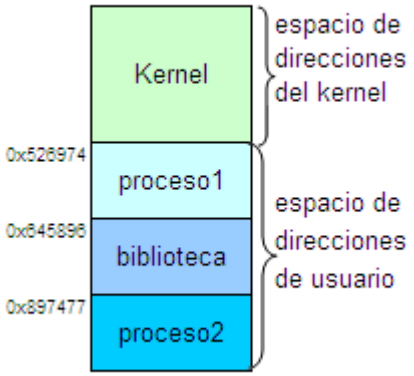

Figura 4.8 Carga de biblioteca dinámica en memoria

La función dlopen retorna el valor del selector de segmento donde está ubicada la biblioteca dinámica en memoria. Por ejemplo retorna el valor 0x00ab, este valor junto al offset determina la dirección virtual de la función de la biblioteca en memoria.

4.9. Obtención del offset de la función de la biblioteca.

Actualmente el offset de la función de la biblioteca a ejecutar desde el Proceso1.bin o el Proceso2.bin se establece de forma directa en el archivo pltEntrada.asm. Este offset se obtiene del campo value de la tabla de símbolos de la biblioteca. Esta forma de implementación se realizó por diversos factores que se detallan en este informe.

Por otro lado, se desarrolló un syscall llamada dlsym cuya función es devolver el offset donde está ubicada la función dentro de la biblioteca. Esta syscall se debe invocar en el main del Proceso1.bin o del Proceso2.bin, debe ser ejecutada después de realizar dlopen. Su forma de implementación es la siguiente:

```
Offset=dlsym("Nombre biblioteca","Nombre función");
```

Al ejecutarse dlsym automáticamente se invoca la función iFnObtenerDirFuncion ubicada en gdt.c. La cual realiza la búsqueda del offset de la

función en la tabla de símbolos de la biblioteca. Una vez encontrado el valor lo retorna al proceso semidinámico.

La función dlsym en ciertas ocasiones trae aparejado inconvenientes durante la ejecución de SODIUM. Si el nombre de la función pasada como parámetro a dlsym es "vFnSleep", en el Proceso1.bin por ejemplo, se observan problemas al querer ejecutar otro programa luego de que dicho proceso finalice su ejecución. Detectamos que este problema se presenta al ejecutar la siguiente línea en la función iFnObtenerDirFuncion.

```
if (( stuTS->name != 0 )&&(stuTS->type==18))
  {
     pcNombreMayuscula =
stFnBuscaNameSectionHeader( ucLecturaTemporalELFTSTR,
stuTS->name );
```

Si se comenta la línea de código indicada no se generan problemas después que el Proceso1.bin finalice. Se analizó profundamente este problema pero no se pudo detectar la causa de la falla.

En el caso de comentar la línea de código de dlsym en el main de los procesos semdinámicos, el programa se ejecuta correctamente debido a que el offset de la función dinámica está establecido en forma manual en el archivo pltEntrada.asm.

4.10. Ejecución de la función ubicada en la biblioteca dinámica.

Para poder ejecutar la función ubicada en la biblioteca dinámica desde el main del Procesos1.bin o del Proceso2.bin se debe ejecutar la siguiente función después de hacer dlopen y opcionalmente dlsym:

```
vFnFuncDin(Selector,offset)
```

Esta función esta desarrollada en el archivo pltSalida.asm. vFnFuncDin debería recibir como parámetro, desde el main, el valor del selector donde esta ubicada la biblioteca (que se obtiene al realizar dlopen) y el offset correspondiente a la ubicación de la función a ejecutar (que se obtiene al hacer dlsym).

Como la función dlsym a veces trae inconvenientes en la ejecución, se sugiere escribir directamente el valor del Offset de la función dinámica como parámetro en la llamada a la función mencionada. Por ejemplo:

`vFnFuncDin(Selector,0x19),` donde 0x19 es el Offset de la función dentro de la biblioteca.

Al recibir el Offset, se almacena en el registro EAX para que cuando llame a la función vFnPltBiblioteca ubicada en el archivo pltEntrada.asm éste obtenga de este registro el Offset y posteriormente realice el llamado a la función de la biblioteca a ejecutar. De esta forma se logró parametrizar el salto a las llamadas a funciones dinámicas que no se pudo lograr anteriormente con el jump.

En la figura que se muestra a continuación se podrá visualizar como es el funcionamiento de las PLT tanto de entrada como de salida. Las flechas en color negro indican el call a la plt de entrada y las de color rojo son las que ejecutan el retf.

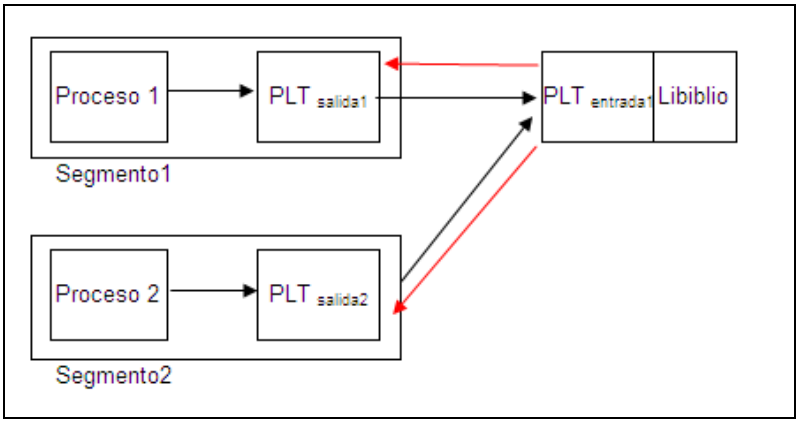

Figura 4.10 Llamada a PLT de Entrada

Una de las razones por las que se decidió implementar dos archivos PLT(uno de entrada y otro de salida) es porque al retornar la función dinámica a la PLT del proceso dinámico se ejecutaba la instrucción ret, la cual retorna dentro del mismo segmento de memoria. Pero como el proceso y la biblioteca están ubicados en distintos segmentos, se producía en SODIUM una excepción numero 13. La manera de solucionar este inconveniente fue generando el archivo pltEntrada.asm, el cual se enlaza estáticamente a la biblioteca. Este archivo contiene la función vFnPltBiblioteca, mencionada anteriormente, cuya finalidad es llamar a la función a ejecutar y finalmente ejecutar la instrucción retf. Esta instrucción retorna a otro segmento de memoria, desapilando los valores de los registro CS y EIP correspondientes a la instrucción posterior al call en el archivo pltSalida.asm. De esta manera se consigue ejecutar correctamente la función invocada en vFnFuncDin.

4.11. Liberación del segmento de memoria donde esta ubicada la biblioteca

Para poder liberar el segmento de memoria donde está ubicada la biblioteca se desarrolló la syscall dlclose. La cual se debe ejecutar desde el main de los Proceso1.bin y Proceso2.bin, después de haber ejecutado la función vFnFuncDin. El

formato de uso de esta syscall es el siguiente: `dlclose("Nombre biblioteca");`

Al ejecutarse dlopen automáticamente se invoca a la función iFnCerrarBiblioteca ubicada en gdt.c. Esta función libera el segmento donde está ubicada la biblioteca. Además modifica el descriptor de segmento perteneciente al proceso dinámico que está asociada a la biblioteca para que vuelva a apuntar al segmento de código del proceso padre. De esta forma se consigue liberar el segmento donde está ubicada la biblioteca.

Para ejecutar esta syscall correctamente los procesos semidinámicos, tanto Proceso1.bin como Proceso2.bin, deben ejecutar dlclose para liberar el segmento donde estaba ubicada la biblioteca.

El inconveniente que trae esta forma de uso, es que si los dos procesos semidinámicos ejecutan dlclose no podrán compartir la biblioteca entre sí. Debido a que al ejecutar un proceso dlclose este libera el segmento, por lo que el otro proceso deberá cargar nuevamente la biblioteca. Para poder ver la compartición de la biblioteca entre ambos procesos semidinámicos, ninguno debe ejecutar dlclose. De esta forma se consigue que si por ejemplo el Proceso1.bin carga por primera vez la biblioteca en memoria y finaliza su ejecución, la biblioteca permanezca en memoria. Por lo que si el Proceso2.bin necesita usar la biblioteca, se evita que el cargador tenga que cargarla nuevamente en memoria.

Se implementó la alternativa de que solo un proceso ejecute dlclose y el otro no lo haga. Pero se encontró el inconveniente de que al ejecutarse nuevamente dlclose no se libera eficientemente el segmento. Por consiguientes se consideró que la mejor forma de probar el dlclose es que los dos procesos ejecuten este sycall, a través del método explicado anteriormente.

5 Conclusión

Luego de lo detallado anteriormente, se pudo visualizar que SODIUM está arraigado a los procesos estáticos. Se puede ver que fueron necesarias muchas modificaciones en la forma de compilar los fuentes del sistema operativo, para poder permitir subir un programa a disco y luego que el mismo pueda ser cargado a memoria. Hasta ahora, las distintas adaptaciones pretendieron lograr estabilizar el sistema operativo con programas estáticos como dinámicos, para luego lograr visualizar las llamadas de manera dinámica.

Otro punto importante a resolver es el correcto funcionamiento de la función dlsym, que por motivos de tiempo no se pudo solucionar. Pero sin embargo sin la utilización de esta syscall se puede ejecutar correctamente la carga y ejecución de las funciones dinámicas como se mencionó anteriormente.

Para evitar que un proceso cierre una biblioteca dinámica existiendo otros procesos que tengan referencia hacia la misma, se deberá crear una estructura en la gdt.h que contenga el nombre de la biblioteca, un contador de referencias y un vector con el offset de cada una de las funciones que contiene la biblioteca, como una tabla de símbolos global.

A futuro deberían analizarse los cambios y el impacto que los procesos estáticos y dinámicos tendrán a la hora de implementar el cargador dinámico para el método de paginación.

6 Referencias

- [BAI10] Baisel Javier, Federico Brucchieri, Nicolás Dimalow, Claudio Mayoral, Martín Stricagnoli. “Paginación en SODIUM”. Trabajo de los alumnos de la cátedra de Sistemas Operativos de la UNLaM. Año 2010.
- [BON11] Bonanno Gabriel, Mario José Bondar, Esteban Carnuccio, Etel Elizabeth Guardia, Alfredo Portero. “Cargador de ELF dinámico”. Trabajo de los alumnos de la cátedra de Sistemas Operativos de la UNLaM. Año 2011.
- [CAS08] Casas Nicanor, Graciela De Luca, Martín Cortina, Gerardo Puyo, Waldo Valiente. “La implementación de diferentes tipos de memoria en un sistema operativo didáctico“. Año 2008.
- [DAR01] Pierre-Alain Darlet, Runtime Loador-Linker Technologies, año 2001.
- [DUA10] Duartes Laura, Silvina Torres, Ezequiel Yacono, Ignacio Barrionuevo, Lucas Tapia. “Habilitar paginación”. Trabajo de los alumnos de la cátedra de Sistemas Operativos de la UNLaM. Año 2010.
- [KER03] Mickel Kerrish, Man Linux, año 2003.
- [LEV00] John R. Levine. "Linkers & Loaders". Morgan Kaufmann Publishers, año 2000.
- [MUC97] Steven S Muchknick. “Advanced compilers: Design and Implementation” Morgan Kaufmann Publishers. Año 1997.
- [SIL05] Abraham Silverschatz. “Fundamentos de Sistemas Operativos”, Séptima edición. editorial McGraw Hill, año 2005.
- [STA08] William Stallings. “Operating Systems Internals and Design Principals”, publicado por Prentice Hall, año 2008.
- [STA10] Richard M. Stallman, “Using the GNU compiler”, publicado por GNU Press, año 2010.
- [TAN99] Andrew Tanenbaum. “Organización de computador: un enfoque estructurado”, cuarta edición. Editorial Pearson Education, año 1999.
- [TIS95] Tool Interface Standard (TIS). “Executable and Linking Format (ELF)”. Specification Version 1.2, año 1995.
- [MEL04] Gorman, Mel. “Understanding the Linux, Virtual Memory Manager”. Editorial Prentice Hall, año 2004.
- Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A: Instruction Set Reference, A-M.

- [TAN09] Andrew Tanenbaum. “Sistemas Operativos Modernos”, tercera edición. Editorial Prenticel Hall, año 2009.
- [ANG06] Angulo, José María. “Microprocesadores avanzados 386 y 486”. Cuarta edición. Editorial Paraninfo Hall, año 2006.