Técnicas Para La Inyección De Defectos De Hardware Y Software Bajo El Sistema Operativo GNU/Linux
Informe de proyecto de grado

Juan Gómez, Alvaro Martínez

Tutores: Andrés Aguirre Dorelo, Ariel Sabiguero Yawelak

Tribunal: Javier Baliosián, Jorge Merlino, Mónica Wodzislawski

Instituto de Computación - Facultad de Ingeniería
Universidad de la República Oriental del Uruguay

16 de diciembre de 2010

Resumen

Existe una tendencia creciente en la sociedad a depender de sistemas de computación en casi cualquier actividad. Es sumamente importante que dichos sistemas estén libres de defectos y sean tolerantes a fallos de manera de poder confiar en los servicios ofrecidos por los mismos.

La inyección de defectos involucra la modificación de forma controlada del estado de datos durante la ejecución de un sistema informático, para determinar si responde según su especificación en presencia de un conjunto de defectos previamente definido.

Al construir un sistema informático la etapa de verificación es esencial para detectar y corregir los defectos, pero los escenarios necesarios para que algunos defectos se activen son poco frecuentes o difíciles de reproducir. En estos casos la inyección de defectos es una opción dado que permite generar escenarios poco frecuentes con la posibilidad de reproducir los mismos.

En este trabajo se presenta la inyección de defectos como una técnica de gran importancia en la verificación de sistemas informáticos. Para ello se introducen conceptos básicos relacionados a la categorización de defectos, distintas técnicas de inyección de defectos y características deseables en herramientas que utilicen estas técnicas.

Posteriormente se aborda en profundidad la inyección de defectos sobre llamadas al sistema operativo y sobre el protocolo USB, exponiendo la investigación de distintas tecnologías en búsqueda de opciones sobre las cuales basarse para la construcción de una herramienta de inyección de defectos sobre llamadas al sistema operativo y el protocolo USB. Por último se expone el producto construido.

Palabras clave: Inyección de defectos, llamadas al sistema, USB, Glibc, verificación.

Índice general

1. Introducción 6
1.1. Motivación 9
1.2. Objetivos 10
1.3. Público objetivo 10
1.4. Organización del documento 10

2. Introducción a la inyección de defectos 12
2.1. Conceptos básicos 12
2.2. Categorización de defectos 13
2.2.1. Categorizaciones elementales de defectos 13
2.2.2. Defectos naturales 15
2.2.3. Defectos fabricados por el hombre 15
2.2.4. Defectos maliciosos 17
2.2.5. Defectos en la interacción 17
2.3. Categorización por tipo de falla inducida 18
2.3.1. Bohrbugs 18
2.3.2. Heisenbugs 18
2.4. Métricas para el impacto de defectos 20
2.5. Resumen 21

3. Métodos y herramientas de inyección de defectos 24
3.1. Características deseables para la inyección de defectos 24
3.2. Categorización de técnicas 26
3.2.1. Técnicas de inyección de defectos por hardware 26
3.2.2. Técnicas de inyección de defectos por software 27
3.2.3. Técnicas de inyección de defectos por simulación 28
3.2.4. Técnicas híbridas de inyección de defectos 29
3.3. Arquitecturas para programas de inyección de defectos 29
3.3.1. Características generales en una arquitectura de inyección de defectos 30
3.3.2. Ejemplo de Arquitectura para una herramienta de inyección de defectos 30
3.4. Técnicas de inyección de defectos de interés para este proyecto . 33

4. Inyección de defectos sobre `syscalls` 34
4.1. Introducción a `syscalls` 34
4.2. Opciones de inyección sobre `syscalls` 36
4.2.1. Modificación de atención de `syscalls` 37
4.2.2. Ptrace 39
4.2.3. LD_PRELOAD 40

4.2.4. Kprobes 43
4.3. Prototipos de inyección sobre `syscalls` 47
4.3.1. Prototipo Ptrace 47
4.3.2. Prototipo LD_PRELOAD 52
4.3.3. Prototipo Kprobes 58
4.4. Conclusión sobre inyección de defectos sobre `syscalls` 59

5. Inyección de defectos sobre USB **60**
5.1. Introducción a USB 60
5.1.1. Protocolo USB 62
5.1.2. USB y el Kernel de GNU/Linux 64
5.2. Opciones de inyección sobre USB 65
5.2.1. Modificación del USB Core 65
5.2.2. USBMon 67
5.2.3. Libusb 69
5.2.4. USB/IP 70
5.3. Conclusión sobre inyección de defectos sobre USB 72

6. Producto **74**
6.1. Arquitectura 74
6.1.1. Módulo FIK 75
6.1.2. WrapperGlibc.so 78
6.1.3. Directorios virtuales 78
6.1.4. Módulo USBFI 79
6.2. Evaluación del producto 81

7. Conclusiones y Trabajo Futuro **84**
7.1. Conclusiones 84
7.2. Trabajo Futuro 85

8. Glosario **88**

A. Pruebas de prototipo Ptrace **96**
A.1. Pruebas sobre File System 96
A.1.1. Pruebas sobre apertura de archivos 96
A.1.2. Pruebas sobre cierre de archivos 97
A.1.3. Pruebas sobre lectura de archivos 97
A.1.4. Pruebas sobre escritura de archivos 98
A.2. Pruebas sobre Red 98
A.2.1. Pruebas sobre lectura de sockets 98
A.2.2. Pruebas sobre escritura de sockets 99

B. Pruebas de prototipo LD_PRELOAD **100**
B.1. Pruebas sobre File System 100
B.1.1. Pruebas sobre apertura de archivos 100
B.1.2. Pruebas sobre cierre de archivos 101
B.1.3. Pruebas sobre lectura de archivos 102
B.1.4. Pruebas sobre escritura de archivos 102
B.2. Pruebas sobre Red 103
B.2.1. Pruebas sobre lectura de sockets 103

B.2.2. Pruebas sobre escritura de sockets 103

C. Pruebas de prototipo KProbes **106**

D. Estructuras USB **108**
D.1. usb_device_descriptor 108
D.2. usb_config_descriptor 109
D.3. usb_interface_descriptor 110
D.4. usb_endpoint_descriptor 112
D.5. urb 114
D.6. usb_ctrlrequest 118

E. Pruebas de prototipo USB **122**
E.1. Filtrado de URBs 122
E.1.1. Primer conjunto de pruebas 122
E.1.2. Segundo conjunto de pruebas 123
E.2. Modificación de datos transmitidos 125
E.2.1. Primer conjunto de pruebas 125
E.2.2. Segundo conjunto de pruebas 126
E.3. Cambio de estado en complete 126
E.3.1. Primer conjunto de pruebas 126
E.3.2. Segundo conjunto de pruebas 127
E.3.3. Tercer conjunto de pruebas 129
E.4. Modificación de datos en la enumeración 130
E.4.1. Primer conjunto de pruebas 131

F. Pruebas USB/IP **136**
F.1. Primer conjunto de pruebas 136
F.2. Segundo conjunto de pruebas 137
F.3. Tercer conjunto de pruebas 137
F.4. Cuarto conjunto de pruebas 139
F.5. Quinto conjunto de de pruebas 139

G. Manual de Producto **142**
G.1. Funcionamiento 142
G.1.1. Modo Ptrace 142
G.1.2. Modo LD_PRELOAD 143
G.1.3. Inyección USB 143
G.2. Comandos 143
G.2.1. Comandos de uso general 143
G.2.2. Defectos para modo LD_PRELOAD 144
G.2.3. Defectos para modo Ptrace 150
G.2.4. Comandos USB 153
G.3. Logs 154

H. Pruebas del módulo USBFI **156**
H.1. Pruebas 156
H.2. Conclusión 157

Índice de figuras

2.1. Defecto, Error, Falla 13
2.2. Clasificación de defectos 14

3.1. Clasificación de técnicas 30
3.2. Estructura de la solución propuesta 31

4.1. Arquitectura GNU/Linux 35
4.2. Opciones 1 y 3 de modificación a la implementación de las `syscalls` 37
4.3. Esquema de FIG 42
4.4. Arquitectura de KProbes [17] 43
4.5. Arquitectura del prototipo realizado con Ptrace 50
4.6. Jerarquía de los defectos implementados en el prototipo Ptrace . 51
4.7. Esquema del prototipo LD_PRELOAD 52
4.8. Jerarquía de defectos 56

5.1. Topología USB 61
5.2. Hub 62
5.3. Ejemplo de dispositivo USB 63
5.4. USB Core 65
5.5. Esquema USB/IP 70

6.1. Descomposición en módulos del producto 75
6.2. Módulo FIK 76
6.3. Jerarquía de defectos LD_PRELOAD 78
6.4. Módulo USBFI 79

Capítulo 1

Introducción

Aunque dispongamos de software perfecto, éste va a ser ejecutado en hardware que presenta fallas, con una tasa muy baja, pero fallas al fin. Usualmente los defectos en el software son más frecuentes que los problemas introducidos por fallos en el hardware, que no es ajeno a malos funcionamientos. Estos fallos pueden conducir al software a estados no considerados y generar desviaciones en el comportamiento del sistema bajo prueba, haciéndolo apartarse de sus objetivos operacionales.

Estos estados y situaciones no pueden ser reproducidos con las técnicas tradicionales de testing sin lograr tener una gran intrusión sobre el sistema bajo prueba. En general, en sistemas donde no tenemos acceso a su implementación, no es posible reproducir estos problemas sin técnicas de inyección de defectos. En este contexto se refuerza la idea de utilizar estas técnicas para lograr tener una medida de calidad del software bajo situaciones externas que pueden ser predecibles pero que generalmente no son tomadas en cuenta.

Las técnicas de inyección de defectos son un proceso que no aporta información sobre cuan buena es determinada implementación, en su lugar nos permiten estimar cuan mal puede llegar a comportarse ante eventos externos. Utilizando estas técnicas podemos anticipar el comportamiento de nuestro sistema en entornos operativos reales, y medir el impacto de estas fallas.

Desde hace unos años la tecnología USB se ha convertido en un estándar, lo que ha llevado a una proliferación de dispositivos y un auge en su uso. La facilidad de uso, ancho de banda y funcionalidad Plug&Play son algunas de las características más atractivas para utilizar al USB como medio de comunicación. Estas ventajas tienen como contrapartida un aumento de la complejidad del protocolo en comparación con otros que no proveen estas características, lo cual redunda en una mayor dificultad a la hora de poder asegurar que la implementación que realiza determinado dispositivo o host USB se comporta correctamente de acuerdo a su especificación. En esta situación resulta interesante utilizar técnicas de inyección de defectos para evaluar como se comportan dispositivos que cumplen con el estándar USB en presencia de defectos como los que pueden encontrarse en sistemas en producción.

Este trabajo se basa en el estudio de la inyección de defectos sobre sistemas informáticos bajo el sistema operativo GNU/Linux. Con dicho fin se analizan técnicas y tecnologías que permiten la inyección y se exponen prototipos que demuestran la viabilidad de utiliza dichas técnicas para crear una herramienta

de inyección de defectos.

Al referirnos a inyección de defectos es importante tener claro los conceptos de error, defecto y falla. Un defecto es una anomalía en el software, en el hardware o en los datos que tiene el potencial de causar errores y fallos [1]. Ejemplos de defectos incluyen cortos en circuitos de hardware, o la división entre cero en un fallo de software. Los defectos son las causas de los errores, pero no todo defecto lleva a un error.

El error es la ocurrencia de la condición invalida o valor incorrecto en el sistema. Un error es la parte del estado del sistema, que es susceptible de ocasionar un fallo. Un error es entonces la manifestación de un defecto en el sistema. Por ejemplo, para una variable puede ser el resultado de una división entre cero. Una falla de un sistema, se da cuando el mismo no se comporta como esta especificado.

Existen varias formas de clasificar los defectos, algunas de estas independientemente del tipo de sistema que se este evaluando. Cada una de estas categorizaciones toma en cuenta aspectos diferentes de los defectos. Algunas de ellas se basan en la fase de creación u ocurrencia, los limites del sistema, el grado de intervención humana, la dimensión, el objetivo, la intención, la duración, el tipo de fallas que inducen etc.

La inyección de defectos surge como una solución viable para el estudio de una característica fundamental de los sistemas como es la confianza (dependability). Ésta se define usualmente como la propiedad de un sistema tal que se puede confiar en el servicio que ofrece [10]. La confianza abarca varios aspectos de los sistemas y no existe una única definición de la misma. Entre los aspectos más frecuentemente relacionados con la confianza se encuentran la disponibilidad (availability), fiabilidad (reliability), seguridad (safety), integridad (integrity), mantenimiento (mantenability), cobertura (coverage), latencia de defecto y comprobabilidad (testeability). Se profundizan estos conceptos en la sección 2.1.

Existen múltiples técnicas de inyección de defectos. Estas se pueden comparar en base a aspectos como baja intrusividad, tiempo requerido para repetir un experimento, reproducibilidad de los experimento, la capacidad de realizar el monitoreo, costo de hardware y software involucrado, y tiempo necesario para adaptar el sistema de inyección de defectos al sistema objetivo entre otros.

Las técnicas de inyección de defectos se pueden dividir en tres grandes categorías, las basadas en hardware, las basadas en software y las basadas en simulación. Luego de investigar las distintas opciones se decidió enfocar este trabajo a las las técnicas de inyección de defectos por software y más específicamente las que son clasificadas como en tiempo de ejecución. Algunas de las características por las cuales se optó este enfoque son que no es necesaria la construcción de hardware adicional por lo que su costo es menor, es posible seleccionar el proceso a ser inyectado y no requieren de un conocimiento exhaustivo del sistema bajo prueba por lo que son potencialmente portables.

El primer objetivo de la inyección de defectos abordado en este trabajo fueron las llamadas al sistema (syscalls). Estas son el mecanismo proporcionado por el kernel para ofrecer distintos servicios y acceso a los recursos del sistema. Cada vez que un proceso necesita leer de un dispositivo, enviar datos por un socket, memoria dinámica para una variable u otros servicios del kernel se invoca a una syscall. De esta manera las `syscalls` se presentan como un punto estratégico para la inyección de defectos.

Con el objetivo de lograr la inyección de defectos en `syscalls` nos encontramos con variadas opciones. Entre ellas modificar el kernel para introducir código extra que permita la inyección, utilizar APIs usadas usualmente para debug como son Ptrace o Kprobes y la interposición a bibliotecas mediante LD_PRELOAD. Estas técnicas son explicadas en el capítulo Inyección de defectos sobre `syscalls`. Luego de investigar cada una de estas opciones se optó por construir una herramienta de inyección de defectos basada en Ptrace y LD_PRELOAD.

Ptrace es una llamada al sistema que proporciona un medio por el que un proceso puede observar y controlar la ejecución de otro examinando y/o cambiando su imagen de memoria o registros [27].

LD_PRELOAD es una variable de entorno usada por el linker de GNU. En esta se puede especificar una lista de bibliotecas compartidas para ser cargadas antes que otras, brindando la posibilidad de crear bibliotecas que implementen funciones del sistema, y hacer que dichas funciones sean utilizadas por los programas en lugar de las originales [23][24].

El segundo objetivo de inyección de defectos es el protocolo USB, para generar una herramienta que permita evaluar el correcto funcionamiento de los dispositivos USB y sus drivers.

En el kernel de GNU/Linux el subsistema encargado de manejar los dispositivos USB es el USB Core [22]. Su objetivo es generar abstracción del control del hardware y dispositivos para evitar dependencias a implementaciones especificas, para ello se definen estructuras, macros y funciones. En este los dispositivos son representados mediante estructuras llamadas descriptores que contienen su descripción y configuración. La comunicación entre el sistema y los dispositivos se hace en unidades llamadas URBs (USB Request Block).

De esta manera el estudio de la inyección de defectos sobre USB se enfoca en encontrar la forma de cambiar la especificación de los dispositivos e inyectar datos en el tráfico de URBs. Esto permite evaluar como reaccionan en situaciones adversas los drivers de los dispositivos o incluso el mismo USB Core.

Entre las características estudiadas para lograr la inyección en el tráfico de URBs nos encontramos con el USBMon. El USB Core provee soporte para la monitorización del tráfico USB, esto se logra mediante la notificación de los URBs que se transmiten a través de un conjunto de operaciones que deben ser registradas previamente. Esta facilidad esta presente en el kernel desde su versión 2.6.11 y fue incluida para soportar la herramienta USBMon [31]. En este trabajo se utiliza el mecanismo ofrecido por el kernel para USBMon registrando un módulo desarrollado por el equipo para inspeccionar el tráfico y realizar la inyección de defectos.

En el transcurso del documento se explican todos los conceptos necesarios para entender la inyección de defectos, se profundiza en las distintas técnicas existentes y varias tecnologías, se exponen los prototipos y los resultados de las pruebas realizadas con ellos y como cierre de este trabajo se presenta una herramienta de inyección que integra las tecnologías Ptrace, LD_PRELOAD y la monitorización del tráfico USB permitiendo la inyección de defectos en `syscalls` y el tráfico USB. Dicha herramienta demuestra la viabilidad de las técnicas elegidas para realizar inyección de defectos, teniendo además características de portabilidad y poca intrusividad.

1.1. Motivación

En los últimos tiempos existe una tendencia creciente a depender de sistemas de computación en casi todas las actividades de la sociedad. Cabe destacar los sistemas de tiempo real, o aquellos utilizados en áreas en las que un error puede provocar perdidas económicas o humanas, como son los sistemas de soporte vital o los usados en el sistemas financieros. En estas circunstancias es fundamental que los sistemas estén libres de defectos y sean tolerantes a fallos de manera de poder confiar el los servicios ofrecidos por los mismos. Ejemplos de las consecuencias de errores en estos sistemas son Ariane 5 Flight 501 [12] y Therac-25 [11].

La inyección de defectos es el proceso de corrupción de un estado de datos durante la ejecución, para determinar si el sistema responde según su especificación en presencia de un conjunto de defectos previamente definido. Esto puede verse como una forma de medir la tolerancia a los fallos y el nivel de confianza (dependability) que posee un sistema. La confianza (dependability) es la propiedad de un sistema tal que con razón se puede confiar en el servicio que ofrece [10] y solo puede ser evaluada mediante la observación del comportamiento del sistema después de la aparición de un fallo. El objetivo de la inyección puede consistir simplemente en medir como afecta a una salida, o estar orientadas a determinar si alguno de los atributos del sistema, tales como el rendimiento, se han visto afectados.

Al construir un sistema informático la etapa esencial para detectar y corregir los defectos y errores es la de verificación. En esta etapa se realiza la depuración del sistema, probando las funcionalidades que brinda y otros atributos como seguridad y rendimiento. Generalmente en dicha verificación se hacen supuestos sobre el funcionamiento de la plataforma (funcionamiento del hardware o el software de base), debido a que se asume generalmente que los errores en la misma no son frecuentes y a las dificultades para reproducir los escenarios de error.

La inyección de defectos es de gran utilidad para la depuración de los sistemas al exponerlos a situaciones adversas y poco frecuentes como pueden ser errores de hardware o software base, o en sistemas grandes o complejos donde brinda la posibilidad de recrear un escenario de falla que no son posibles de recrear por otros medios.

En algunos sistemas, si se utiliza una métrica de tiempo medio entre fallos, estos pueden ser fiables en el orden de años. En estos casos la aparición fallas tiene que ser acelerada artificialmente con el fin analizar la reacción del sistema sin esperar a su aparición natural [3]. En estos escenarios las técnicas de inyección de defectos mejoran la cobertura de la depuración, y ayuda a comprender cómo se comporta el software cuando ocurren eventos inusuales.

Estas técnicas se pueden usar tanto en sistemas de hardware como en sistemas de software. Para el hardware, los defectos pueden ser inyectados en las simulaciones del sistema y en la ejecución, tanto a nivel de pines o en el plano interno de algunos chips. Para el software, los defectos pueden ser inyectados en simulaciones del sistema, sistemas distribuidos, o en el sistema en funcionamiento, desde el nivel de los registros de la CPU hasta nivel de red. Variaciones en las técnicas de inyección de defectos permiten que sean aplicadas a muchos tipos de software, y para distintos fines en distintas fases de desarrollo de software.

Uno de los temas abordados en este trabajo es la inyección de defectos sobre

`syscalls`. Estas son el mecanismo por el cual el kernel ofrece a los procesos distintos servicios y acceso a los recursos del sistema. Que un proceso solicite un servicio mediante una `syscall` no garantiza el éxito de la solicitud, por lo que los escenarios de error deben ser contemplados por los procesos que las invocan. Es así que para evaluar el correcto funcionamiento de las aplicaciones ante la ocurrencia de errores o funcionamiento anómalo del sistema base, las `syscalls` se presentan como un punto estratégico para la inyección de defectos.

Otro tema abordado en este trabajo refiere a la tecnología USB. Ésta se ha convertido en un estándar ampliamente usado. La facilidad de uso, ancho de banda y funcionalidad Plug&Play son algunas de las características más atractivas para utilizar al USB como medio de comunicación. Para soportar estas características el protocolo usado por USB es de mayor complejidad que el de las tecnologías que no proveen estas características. Esto redunda en una mayor dificultad a la hora de poder asegurar que la implementación que realiza determinado dispositivo o host USB se comporta correctamente de acuerdo a su especificación. En esta situación la inyección de defectos sobre el protocolo USB surge como una opción de interés para evaluar el comportamiento de los dispositivos y host USB en situaciones adversas.

1.2. Objetivos

Este proyecto consta de dos grandes objetivos. El primero consiste en relevar el estado del arte en lo que refiere a los defectos en sistemas informáticos y la inyección de defectos en los mismos. Esto abarca el estudio de distintas técnicas y tecnologías que permiten realizar esta tarea, más concretamente en lo que refiere a inyección de defectos sobre procesos y protocolo USB.

El segundo gran objetivo es la construcción de prototipos que evalúen la factibilidad de inyectar defectos en procesos y el protocolo USB mediante distintas tecnologías, para luego pasar a la construcción de una herramienta de inyección que se base en las más apropiadas.

1.3. Público objetivo

Este documento tiene por público objetivo personas relacionadas al área de Tecnología de la Información con conocimientos en sistemas operativos, arquitectura de computadores y testing. Además para una mejor comprensión del documento es aconsejable que el lector tenga conocimientos referentes al protocolo USB y del sistema operativo GNU/Linux, más específicamente relacionados con el manejo de `syscalls`. Estos no son un requisitos excluyentes dado que en el transcurso del documento son introducidos los conceptos básicos necesarios para la comprensión del mismo.

1.4. Organización del documento

Este documento está organizado en siete capítulos, además del glosario, bibliografía y apéndices. Estos intentan introducir al lector en la temática del proyecto desde los conceptos básicos, pasando por técnicas y tecnologías estudia-

das, presentando luego un producto desarrollado donde se aplica lo investigado y finalizando con las conclusiones y trabajos futuros.

Los distintos capítulos son los siguientes:

1. Introducción: Presenta las motivaciones y objetivos del proyecto.

2. Introducción a la inyección de defectos: En este capítulo se presentan conceptos necesarios para comenzar a entender la inyección de defectos como lo son la definición de defecto, error y falla, las distintas categorizaciones de los defectos y como afectan a los sistemas.

3. Métodos y herramientas de inyección de defectos: Se introducen las herramientas de inyección de defectos, describiendo las características con que deben contar, distintas técnicas, ejemplos de los componentes que comúnmente se pueden encontrar en herramientas de este tipo y por último se explican que técnicas son las de interés en el marco de este proyecto.

4. Inyección de defectos sobre `syscalls`: En este capítulo se describe el funcionamiento de las `syscalls` para luego explicar diferentes tecnologías estudiadas que permiten la inyección de defectos sobre `syscalls`. Finalmente se exponen diferentes prototipos realizados basados en esas tecnologías usados en la evaluación de las mismas.

5. Inyección de defectos sobre USB: En este capítulo se describe el protocolo USB, varias opciones de inyección que fueron investigadas y las pruebas realizadas para evaluar dichas tecnologías.

6. Producto: Luego de evaluadas las distintas tecnologías se selección un conjunto de ellas para implementar un herramienta de inyección de defectos. En este capítulo se describe dicha herramienta.

7. Conclusiones y trabajos futuros: Para cerrar el trabajo se presentan las conclusiones del trabajo realizado y las posibles mejoras a la herramienta elaborada así como el trabajo futuro.

Además de los capítulos mencionados el documento cuenta con el Glosario, Bibliografía y Apéndices. El glosario contiene la definición de varios de los conceptos manejados en el documento. Los apéndices abarcan las pruebas de los prototipos y del producto, detalles de las estructuras del protocolo USB y el manual del producto.

Capítulo 2

Introducción a la inyección de defectos

En esta sección se introduce al lector en el tema de la inyección de defectos de software y hardware. Se abordan los temas referentes a clasificación de los defectos y métricas para el impacto de las fallas en el software.

2.1. Conceptos básicos

Es importante contar con una definición precisa de los conceptos de error, defecto y falla.

Un **defecto** (fault) es una anomalía física (defecto, imperfección) en el software, en el hardware o en los datos que tiene el potencial de causar errores y fallos [1]. Ejemplos de defectos incluyen cortos en circuitos de hardware, o la división entre cero en un fallo de software. Los defectos son las causas de los errores, pero no todo defecto lleva a un error. Un defecto se dice activo cuando produce un error.

El **error** (error) es la ocurrencia de la condición invalida o valor incorrecto en el sistema. La interacción entre el defecto y un estímulo es lo que produce el error [1]. Un error es la parte del estado del sistema, que es susceptible a ocasionar un fallo. Un error es entonces la manifestación de un defecto en el sistema. Por ejemplo, una cantidad de voltaje incorrecta en un circuito es un error que puede ser causado por un corto circuito o un valor incorrecto. Para una variable puede ser el resultado de una división entre cero.

Un error es por naturaleza temporal. Existen dos estados posibles de un error, latente o detectado. Un error es latente mientras que no ha sido reconocido como tal. Puede ser detectado por mecanismos de detección de errores que analizan el estado del sistema, o por los efecto del error sobre el sistema.

Una **falla** (failure) de un sistema se da cuando el mismo no se comporta como esta especificado [1].

Múltiples errores se pueden originar de un defecto. Si se ve al sistema como un conjunto de componentes, los errores en un componente pueden transformarse en fallas, que originan defectos que se propagan a más componentes a través del sistema. En la Figura 2.1 se muestra las transiciones entre defecto error y falla.

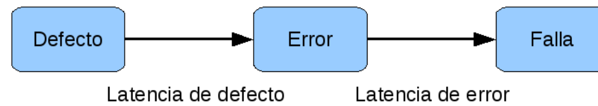

Figura 2.1: Defecto, Error, Falla

El tiempo entre la aparición de un defecto y la primera manifestación de un error se llama latencia de defecto. El tiempo entre la ocurrencia de un error y que el mismo es detectado se llama latencia de error.

2.2. Categorización de defectos

Existen varias formas de clasificar los defectos, algunas de estas independientemente del tipo de sistema que se este evaluando. Cada una de estas categorizaciones toma en cuenta aspectos diferentes de los defectos.

2.2.1. Categorizaciones elementales de defectos

A continuación se exponen criterios de categorización elementales, basados en características de los defectos, y algunas combinaciones entre ellos [10]. Un esquema de esta categorización se muestra en la Figura 2.2.

Por fase de creación o ocurrencia

- Defectos de desarrollo: todos los defectos que ocurren durante las etapas de desarrollo y mantenimiento.
- Defectos operacionales (o de funcionamiento): ocurren en la ejecución de los servicios del sistema durante la fase de uso.

Por limites del sistema

- Defectos internos: ubicados dentro de los limites del sistema.
- Defectos externos: originados fuera de los limites del sistema y propagados al sistema por interacción o interferencia. Estos son introducidos desde afuera por errores o mal funcionamiento de componentes, insumos o software, de los que depende el sistema.

Por intervención humana

- Defectos naturales: generalmente físicos (hardware). Son causados por fenómenos naturales, sin participación humana.
- Defectos causados por el hombre: resultado de una acción del hombre.

Por dimensión

- Defectos de Hardware: afectan al hardware u originados en el hardware.
- Defectos de Software: afectan al software, programas o datos.

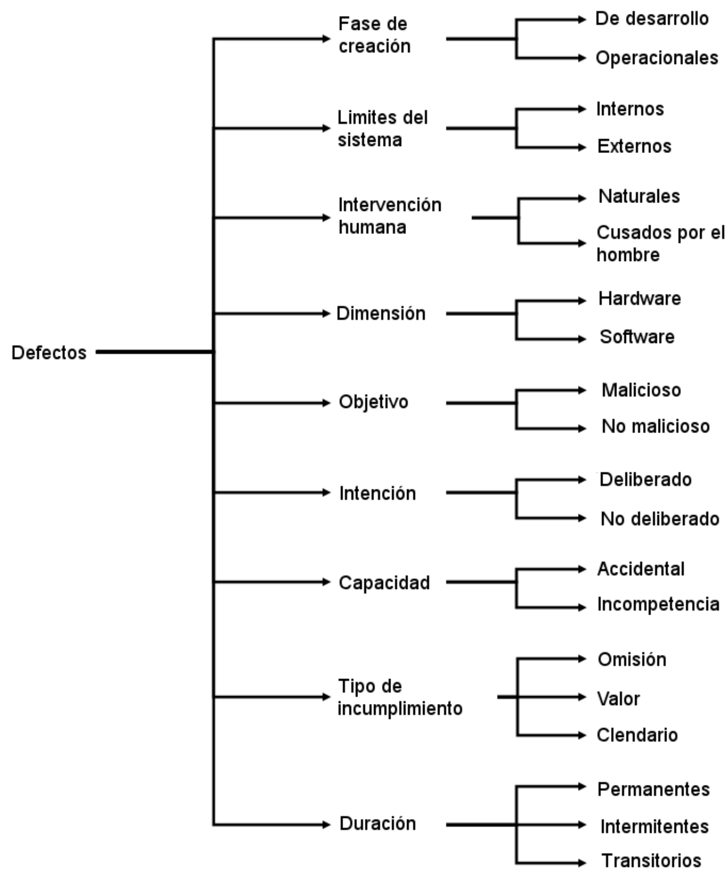

Figura 2.2: Clasificación de defectos

Por objetivo

- Defectos maliciosos: introducidos por el hombre con malas intenciones.
- Defectos no maliciosos: introducidos sin malas intenciones.

Por intención

- Defectos deliberados: resultado de una decisión nociva.
- Defectos no deliberados: introducidos sin conciencia.

Por capacidad

- Defectos accidentales: introducidos inadvertidamente.
- Defectos por incompetencia: causados por falta de competencia profesional de las personas o insuficiencias de la empresa de desarrollo.

Por tipo de incumplimiento

- Defectos de omisión: se dan cuando el sistema, o un componente del mismo, no realiza alguna de las funcionalidades que debe realizar ante las entradas correspondientes. Por ejemplo cuando no se obtiene una respuesta a una solicitud de otro componente.
- Defectos de valor: son aquellos en los que se obtiene una respuesta en el tiempo correcto pero con un valor incorrecto.
- Defectos de calendario: se dan cuando se obtiene la respuesta correcta, pero en un tiempo que esta fuera del rango aceptable. Los defectos de calendario se asocian tanto a respuestas con demoras como a respuestas tempranas. Este tipo de defectos son de gran relevancia en sistemas con requerimientos de respuestas en tiempo real.

Por duración

- Defectos permanentes: son irreversibles. La recuperación de estos defectos se logra solo con el remplazo del componente dañado.
- Defectos transitorios: son usualmente causadas por elementos del entorno que interactúan con el sistema, causando que el mismo entre en un estado erróneo. Algunos ejemplos son interferencia electromagnética y cambios de voltaje.
- Defectos intermitentes: Ocurren debido a hardware inestable o variaciones en el estado del hardware.

Si se analiza un defecto desde los distintos puntos de vista propuestos, el mismo, dependiendo de combinación de las características caerá en varias categorías. No todas las combinaciones son válidas, por ejemplo, las faltas naturales no pueden ser clasificadas por objetivo, intención o capacidad. A continuación se analizan algunas de las combinaciones de las categorías antes mencionadas, muchas más pueden ser inferidas posteriormente.

2.2.2. Defectos naturales

Los defectos naturales pueden ser clasificados según la fase de ocurrencia de los mismos. Así se obtienen los defectos naturales de la fase de producción, que se originan durante el desarrollo, y los defectos de la etapa de funcionamiento.

También pueden ser clasificados por los limites del sistema. Los defectos naturales internos se deben a los procesos naturales que causan deterioro físico.

Los externos, se deben a procesos naturales que se originan fuera del sistema, y causan interferencia física por penetrar en los límites del hardware (radiaciones, etc) o introducidos mediante las interfaces.

2.2.3. Defectos fabricados por el hombre

Las dos clases fundamentales de los defectos de desarrollo se distinguen por el objetivo de los desarrolladores, o de quien interactúa con el sistema durante su uso. Estos pueden ser entonces defectos maliciosos o defectos no maliciosos.

Defectos no maliciosos

No presentan objetivos maliciosos. De acuerdo a la intención se dividen en:

- defectos no maliciosos no deliberados: se deben a acciones no deseadas de las cuales el desarrollador, operador, etc. no tiene conocimiento.
- defectos no maliciosos deliberados: que se deben a malas decisiones, es decir, acciones que son erróneas y provocan defectos. Suelen ser reconocidos como defectos sólo después de que el sistema presenta un comportamiento inaceptable, por lo tanto, una falla.

Si además consideramos la etapa de aparición de los defectos tenemos:

- defectos no maliciosos deliberados de desarrollo: son el resultado general de compensaciones, ya sea destinadas a la preservación de rendimiento aceptable, a facilitar la utilización del sistema o inducidas por consideraciones económicas.
- defectos no maliciosos deliberados de interacción: puede ser el resultado de la acción de un operador, ya sea destinado a la superación de una situación imprevista, o por la violación deliberada del procedimiento de funcionamiento.
- defectos no malicioso de desarrollo: pueden existir en el hardware y en el software. En hardware, especialmente en los microprocesadores, algunos defectos de desarrollo se descubren después de que la producción ha comenzado. Tales defectos se especifican en las actualizaciones. La búsqueda de estos continúa durante toda la vida de los procesadores, por lo tanto, nuevas especificaciones de las actualizaciones se publica periódicamente.

Se suele considerar que tanto los errores como las mala decisiones son accidentales, siempre que no se realicen con objetivos maliciosos. Sin embargo, no todos los errores y las malas decisiones tomadas por personas no maliciosas son accidentes. Del punto de vista de la capacidad, podemos observar que algunos errores muy perjudiciales son el fruto de malas decisiones tomadas por personas que carecen de competencia profesional para hacer el trabajo que se han comprometido. Se dividen entonces en los defectos no maliciosos accidentales, y por incompetencia.

Algunos defectos de desarrollo se presentan debido a que las herramientas hechas por el hombre son defectuosas. Los COTS son inevitablemente utilizados en el diseño del sistema. Su uso introduce problemas adicionales. Ellos pueden venir con defectos conocidos y pueden contener defectos desconocidos (bugs, vulnerabilidades, etc.). Sus especificaciones pueden ser incompleta o incluso incorrecta.

Algunos defectos que afectan el desarrollo de software puede causar envejecimiento de software, es decir, errores progresivamente acumulados causantes de la degradación del rendimiento o falla. Ejemplos de ello son el consumo creciente de memoria, hilos no terminados, bloqueos de archivos no liberados, la corrupción de los datos, fragmentación del espacio de almacenamiento, y la acumulación de errores de redondeo [7].

2.2.4. Defectos maliciosos

Tiene por objetivo causar daño al sistema durante su uso. Debido al objetivo, la clasificación de acuerdo con la intención y capacidad no es aplicable. Las posibles metas de estos defectos son:

- interrumpir o detener el servicio, causando la denegación de servicio (Denial of Service).
- acceder a información confidencial.
- modificar indebidamente el sistema.

Se agrupan en dos clases:

- Defectos maliciosos lógicos: abarcan los defectos de desarrollo tales como caballos de Troya, las bombas lógicas o de calendario, las amenazas programadas y defectos de funcionamiento introducidos por virus o gusanos.
- Intentos de intrusión: son defectos externos operacionales. El carácter externo de los intentos de intrusión, no excluye la posibilidad de que puedan ser realizados por los operadores del sistema, o administradores que exceden sus derechos. Los intentos de intrusión pueden valerse de medios físicos para provocar los defectos, como fluctuación de energía, radiación, calefacción/refrigeración, etc.

2.2.5. Defectos en la interacción

Los defectos de interacción se producen durante la fase de utilización, por lo que son todos defectos de funcionamiento. Son causados por elementos del entorno que interactúan con el sistema, por lo tanto todos son externos. La mayoría de estos se originan debido a la acción humana en el uso del entorno.

Una excepción a esto son los defectos externos naturales. En estos, la naturaleza interactúa con el sistema sin la participación humana (rayos cósmicos, erupciones solares, etc).

Una amplia clase de defectos de interacción causados por el hombre son los defectos de configuración, es decir, mal ajuste de parámetros que puede afectar a la seguridad, trabajo de red, almacenamiento, etc. Tales defectos pueden ocurrir durante los cambios de configuración realizados durante el período de de mantenimiento que es realizado simultáneamente con la operación del sistema. Estos se llama defectos de reconfiguración.

Una característica común de los defectos de interacción es que para que tengan éxito normalmente requieren la previa existencia de una vulnerabilidad, es decir, una defecto interno que permite a uno externo dañar el sistema.

La vulnerabilidad puede ser un defecto de desarrollo o funcionamiento, que puede ser malintencionado o no. En el caso de los no maliciosos pueden ser el resultado de defectos deliberados de desarrollo, por razones económicas o de utilidad, lo que da lugar a una protección limitada, o incluso la ausencia de protección.

2.3. Categorización por tipo de falla inducida

Jim Gray propuso una clasificación de defectos de software en Bohrbugs y Heisenbugs, basado en el tipo de fallas que inducen [5]. A continuación se presentan estos conceptos y sus principales características.

2.3.1. Bohrbugs

Bohrbug fue nombrado por el modelo del átomo de Bohr. Son aquellos defectos que causan una falla siempre que se ejecuta la operación que los contiene. Son defectos fácilmente detectables por las técnicas de depuración tradicionales.

Un Bohrbug siempre se repite al volver a repetir la operación, esto sólo es posible si el diseñador o programador del sistema codificó un comportamiento incorrecto para esa operación. En consecuencia los defectos de tipo Bohrbugs se deben a malas especificaciones sobre el sistema, o errores en la conversión de las especificaciones a diseño.

2.3.2. Heisenbugs

La palabra Heisenbug proviene del principio de incertidumbre de Heisenberg, que afirma que es imposible de predecir la posición y velocidad de una partícula al mismo tiempo. Los Heisenbugs son defectos temporales e intermitentes, o sea que la falla podría no aparecer al volver a ejecutar la operación que la produjo en una primera instancia. Por lo general las condiciones de activación de este tipo de defectos ocurre raramente por lo que son difícilmente reproducibles.

En la mayoría de la industria de sistemas de software, los productos son liberados después de realizado el diseño de revisiones, aseguramiento de la calidad, alpha, beta y gamma testing. Esto libera al producto de la gran mayoría de Bohrbugs que son fácilmente capturados, pero muchos de los Heisenbugs persisten. Como resultado, la mayoría de los fallos en sistemas de software son blandos, es decir, si el sistema de software se reinicia vuelve a funcionar correctamente.

Si se pudiese hacer ocurrir una vez más cualquier falla, todo Heisenbugs podría clasificarse como Bohrbugs. Para lograr esto se debe recrear exactamente el entorno de la ejecución del programa la primer vez y proporcionar las mismas entradas, entonces la falla ocurriría de nuevo y, por tanto, de acuerdo con la definición, el error debe ser un Bohrbug. Sin embargo es prácticamente imposible recrear las condiciones que existía cuando la falla ocurrió en primera instancia.

Para explicar las dificultades asociadas a reproducir un estado del sistema, se puede simplificar un programa como un conjunto de procesos. Cada uno consiste del código, sus datos y la pila. Dado el programa, un estado inicial de los datos, la pila y una entrada, el programa actúa moviendo los datos y la pila de un estado a otro previsible. Esto se puede inferir del correcto funcionamiento de las reglas del programa. Pero además depende de factores del entorno como son el hardware, el código del sistema operativo y su estructura de datos, otros programas con los cuales se sincronice, interacción con los dispositivos de entrada/salida, etc.

No se puede asumir que los factores del entorno son los mismos cada vez que el programa se ejecuta. El código del sistema operativo puede gestionar la memoria de forma diferente cada vez que se ejecuta el programa, el hardware podría haber cambiado, otros programas pueden haber cambiado su comportamiento.

Un ejemplo de estos cambios es el uso del reloj del sistema. Muchas operaciones no están sincronizados con el reloj del sistema. La mayoría de los dispositivos periféricos usan su propio reloj o no usan reloj. Esto significa que los procesos y los correspondientes periféricos involucrados no estén sincronizados. Las operaciones asincrónicas crean incertidumbres en la duración de las tareas que se pueden traducir en la desaparición de fallas.

En operaciones de búsqueda sobre el disco duro. Si se ejecuta el programa una vez y luego se reinicia el sistema y se vuelve a ejecutar el programa una segunda vez, debido a la variabilidad del tiempo de búsqueda se pueden obtener diferentes secuencias de accesos a disco. Los procesos normalmente se bloquea en caso de que necesiten una página desde el disco. Los procesos pueden permanecer bloqueados o desbloqueados una cantidad de tiempo variable y así ser planificados en orden diferente. Esto significa que una condición que pudo aparecer en la primera ejecución no ocurra en la siguiente.

Otro ejemplo de las dificultades para reproducir exactamente el estado en la ejecución en un programa, se da en los que utilizan conectividad de red. Es muy complejo reproducir la carga en la red con el fin de volver a un estado anterior del sistema, reproduciendo por ejemplo situaciones como la perdida de paquetes y retrasos en una red con protocolos de tipo "best effort".

Otro elemento difícil de reproducir es el fenómeno de envejecimiento de software [7]. Este refiere a como un programa luego de iniciado acumula con el paso del tiempo condiciones de falla, que conducen a la degradación (lentitud, incoherencia en datos) o fallas transitorias. Causas típicas del envejecimiento son no liberar memoria, no cerrar archivos, la corrupción de los datos, la fragmentación en el espacio de almacenamiento, etc.

Se puede observar en técnicas de depuración cíclica que ante una falla se adjuntan en el código las herramientas de depuración y se realiza una repetición. Sin embargo las herramientas de depuración que se han adjuntado al programa pueden hacer que el entorno del programa cambie. Por ejemplo, si el depurador y el programa se ejecuta en el mismo espacio de direcciones la cantidad de memoria podría ser afectada. Además, al agregar un proceso adicional, el proceso de planificación también puede ser alterado. Esto podría conducir a la desaparición de la falla.

Una razón común para el efecto Heisenbug en la depuración, es que la ejecución de un programa de depuración a menudo limpia la memoria antes de iniciar el programa, y coloca las variables en la pila. Estas diferencias en la ejecución, pueden alterar el efecto de los errores de acceso fuera de los rangos permitidos, o suposiciones incorrectas sobre el contenido inicial de la memoria.

La dificultad para reproducir las fallas provocadas por un defecto Heisenbug son consecuencia de la interacción entre las condiciones que se producen fuera del programa que se está depurando y el propio programa. Estos defectos derivan de un entorno inestable, suposiciones erróneas sobre el entorno (cantidad de memoria, tiempos de respuesta, etc.) o suposiciones erróneas acerca de la interacción entre los diversos sub-componentes del sistema.

Aunque no siempre se puede repetir una falla provocada por un defecto Heisenbug, se puede obtener una buena estimación probabilística de donde debe estar el defecto. Esto da lugar a la elaboración de modelos probabilísticos de defectos de software [4].

2.4. Métricas para el impacto de defectos

La confianza (dependability) se define usualmente como la propiedad de un sistema tal que con razón se puede confiar en el servicio que ofrece[3]. A diferencia de rendimiento, la confianza no puede ser evaluada utilizando referencias estándar de los sistemas y metodologías de ensayo, sino de la observación del comportamiento del sistema después de la aparición de un fallo. En este contexto, medir la confianza de un sistema se traduce en el estudio de defectos y errores en el mismo.

La inyección de defectos surge como una solución viable para el estudio de la confianza, y ha sido profundamente investigada y explotada por las universidades y la industria.

No existe una única definición de la característica de confianza (dependability) para un sistema. A continuación se muestran algunas de ellas:

1. Confianza es un concepto integrador que abarca los siguientes atributos [10]:
   - disponibilidad (availability): el correcto acceso al servicio.
   - fiabilidad (reliability): la continuidad de servicio de forma correcta.
   - seguridad (safety): ausencia de consecuencias catastróficas para el usuarios y el medio ambiente.
   - integridad (integrity): ausencia de alteraciones inadecuadas del sistema.
   - mantenimiento (mantenability): la capacidad de soportar modificaciones y reparaciones.
2. La confianza (dependability) es un atributo cualitativo del sistema que se cuantifica a través de medidas específicas [15]. Las dos medidas principales son:
   - La fiabilidad (reliability) de funcionamiento: probabilidad de sobrevivir (sin fallas) en un intervalo de tiempo.
   - La disponibilidad (availability): probabilidad de estar operativo (sin fallas) en un determinado instante en el tiempo.
   - La media de tiempo hasta la falla (MTTF mean time to failure) y el tiempo medio entre fallos (MTBF mean time between failure) son también frecuentemente utilizadas.
3. La confianza (dependability) es la propiedad de un sistema tal que se puede confiar en el servicio que ofrece. Es un término utilizado para la descripción general de las características de un sistema pero no se puede expresar mediante un único indicador. Hay varias métricas que fundamentan la confianza [3]:
   - Fiabilidad (reliability): probabilidad condicional de que el sistema realice correctamente todo el intervalo [t0, t], suponiendo que el sistema estaba actuando correctamente en el momento t0. Se refiere a la la continuidad del servicio.

- Disponibilidad (availability): probabilidad de que un sistema esté en funcionamiento correctamente y disponible para desempeñar sus funciones en el instante en el tiempo t. Se refiere a que el sistema se encuentre preparado para el uso.
- Seguridad (safety): probabilidad de que un sistema o bien realice su funciones correctamente o se aleje de sus funciones de una manera que no no perturbe el funcionamiento de otro sistema o comprometa la seguridad de cualquier personas asociadas con el sistema. Se refiere a la no ocurrencia de las consecuencias catastróficas para el entorno.
- Media de tiempo hasta la falla (MTTF): el tiempo que se espera que un sistema funcione antes de que el primer fallo ocurra. Se refiere a la aparición del primer fracaso.
- La cobertura (coverage): probabilidad condicionada dada la existencia de un defecto de que el sistema se recupere. Refiere a la capacidad del sistema para la detección/recuperación de un fallo.
- Latencia de defecto: tiempo entre la aparición del defecto y la aparición de un error que resulte de dicho defecto.
- Mantenimiento (mantenability): medida de la facilidad con la que un sistema puede ser reparado, una vez que ha fallado. Relacionado con la aptitud a someterse a reparaciones y evolución.
- Comprobabilidad (testeability): medio por el cual la existencia y la calidad de ciertos atributos dentro de un sistema están determinados. Refiere a la validación y el proceso de evaluación del sistema.

Existen diversos métodos de evaluación de la confianza, cada uno con diferentes variaciones y propiedades. Al clasificar estos métodos según sean usados para prevenir o verificar la confiabilidad se obtienen tres categorías de evaluación:

1. por análisis: análisis de procesos y de producto. Por ejemplo comparando con la cantidad de defectos de otros proyectos, analizando fallas creíbles a distintos niveles de abstracción del sistema.

2. por experiencia de campo: evidencia de la observación (reporte de incidentes), por usos previos del sistema en situaciones similares.

3. por verificación: probando la salida del sistema a prueba, dadas entradas representativas en un test de confianza.

En la última categoría es que son aplicadas las técnicas de inyección de defectos.

2.5. Resumen

Un defecto es una anomalía (defecto, imperfección) en el software, en el hardware o en los datos que tiene el potencial de causar errores. Por su parte los errores son condiciones invalidas o valores incorrecto en el sistema que pueden derivar en fallas, siendo una falla un comportamiento que no cumple con la especificación del sistema.

El origen de los defectos es muy variado. Los mismos pueden encontrarse dentro del sistema, como los que tienen origen en las etapas de desarrollo debido

a una mala especificación, o pueden provenir de fuera del sistema, como puede ser una entrada de datos que no cumple con lo que espera el sistema, el mal funcionamiento de algún sistema externo del que se necesita algún servicio o una fuente de interferencia externa.

En la construcción de sistemas confiables y de calidad se deben considerar la existencia de los defectos y errores, para detectarlos de forma temprana y evitar que se transformen en fallas. También es importante ante la ocurrencia de fallas tener algún mecanismo de recuperación que evite la caída del sistema o al menos amortigüe el costo de la suspensión temporal de los servicios que el mismo ofrece.

Las técnicas de inyección de defectos permiten verificar las características de los sistemas que refieren a la confiabilidad. En estas técnicas se introducen defectos en los sistemas y se analizan las salidas del mismo para verificar que su comportamiento sea el especificado.

Los sistemas de inyección de defectos son herramientas de depuración de gran importancia en las etapas de desarrollo. Permiten evaluar el comportamiento de los sistemas ante situaciones adversas como pueden ser errores de hardware o software base, acelerar la aparición de defectos poco frecuentes y permiten recrear los escenarios de falla mediante la repetición de las condiciones iniciales.

En la próxima sección se describen distintas técnicas y herramientas de inyección de defectos.

Capítulo 3

Métodos y herramientas de inyección de defectos

El objetivo de una herramienta de inyección de defectos es introducir defectos en un sistema objetivo. La finalidad es determinar si el sistema responde según su especificación o evaluar su tolerancia a los defectos.

La inyección de defectos debe ser acompañada de un análisis previo, en el cual se determinan los defectos que son de interés introducir y los distintos estados y puntos en los cuales se debe inyectar dichos defectos. Después de dicho análisis se procede al diseño de los casos de prueba, especificando los tipos de defectos a introducir, puntos y estados de prueba y resultados esperados.

Esta sección se centra en los métodos y herramientas de inyección de defectos. Primero se describen algunas características deseables de dichas herramientas, presentando luego distintas técnicas de inyección de defectos, y por último se discute una posible composición de un sistema de inyección.

3.1. Características deseables para la inyección de defectos

Algunas de las características deseables en una herramienta de inyección de defectos son: bajo nivel de intrusividad, alta velocidad, fácil y adecuada reproducibilidad de los experimentos, buena capacidad de observabilidad y bajo costo.

Intrusividad:

Es medido como la diferencia entre el comportamiento del sistema objetivo en ejecución normal, y cuando es objeto de la inyección de defectos. Las herramientas de inyección de defectos deben presentar baja intrusividad, garantizando que el comportamiento del sistema objetivo difiera mínimamente entre ambas situaciones [9].

Una causa de intrusividad es la introducción de instrucciones o módulos para soportar la inyección de fallas. Estos provocan que la secuencia de los módulos e instrucciones ejecutados sea diferente.

Se puede diferenciar la intrusividad de tiempo. Esta se debe a cambios introducidos en el sistema objetivo, que tienen por resultado una disminución en la velocidad del sistema o de algunos de sus componentes. Puede deberse a la introducción de instrucciones o módulos para el soporte a la inyección de defectos, o a que la herramienta de inyección de defectos comparta recursos del entorno con el sistema objetivo. También existe intrusividad en la memoria del sistema objetivo. Esta se da cuando se introduce nuevo código y datos necesarios para soportar la inyección de defectos.

Velocidad:

Refiere al tiempo requerido para repetir un experimento. Dado que en la inyección de defectos generalmente se dan múltiples repeticiones para cada defecto en particular, se requiere que la ejecución de cada experimento se demore lo menos posible. Esto apunta a reducir los tiempos necesarios para poner en marcha la inyección, inyectar los defectos y observar las salidas [9].

Los tiempos de inyección pueden acelerarse con funcionalidades que permitan la definición de campañas de inyección. En estas se indican varios defectos a introducir en un mismo experimento. En lo que refiere a las salidas es importante que la información recabada sea presentada de forma clara para que su interpretación sea lo más sencilla.

Reproducibilidad de los experimentos:

Es la capacidad de reproducir un determinado experimento de inyección. La reproducción puede ser determinista o probabilista. Es determinista cuando siempre es posible reproducir las mismas condiciones iniciales del experimento y obtener las mismas respuestas del sistema. Cuando las condiciones iniciales solo se puedan reproducir de forma aproximada la respuesta del sistema varía entre experimentos. La reproducción probabilística consiste en generar varias muestras, cada una en un número suficiente de experimentos de forma que se reproduzca con alto grado de confianza los valores de las medidas analizadas [9].

Observabilidad:

Es la capacidad de realizar el monitoreo del sistema objetivo de la inyección, obteniendo lecturas válidas o siguiendo la evolución de un servicio luego de la inyección [9].

Costo:

Está dado por el hardware y software involucrado, y además el tiempo necesario para adaptar el sistema de inyección de defectos al sistema objetivo [9].

El hardware y software a utilizar depende de las tecnologías elegidas.

La adaptabilidad del sistema depende del diseño de la herramienta. Este diseño debe permitir el uso de la herramienta sobre distintos sistemas objetivos, con el menor número de modificaciones posibles. Debe ser lo suficientemente flexible como para poder ser adaptado con facilidad.

3.2. Categorización de técnicas

Las distintas técnicas de inyección de defectos pueden dividirse en tres categorías, inyecciones por hardware, por software y simulación. Adicionalmente de la combinación de las técnicas mencionadas surgen las técnicas híbridas de inyección de defectos. A continuación se describen aspectos generales sobre cada una de estas categorías.

3.2.1. Técnicas de inyección de defectos por hardware

Los inyectores de defectos implementados por hardware (HWIFI), usan hardware adicional especialmente diseñado, para introducir los defectos en el hardware del sistema objetivo [3].

Estos métodos son adecuados para el estudio de las características de funcionamiento de prototipos, con fuertes requerimientos de tiempo de respuesta, disparados por eventos de hardware o monitoreo. También son de gran utilidad para inyectar defectos en lugares donde no es posible acceder por otros medios.

Como contrapartida, el conjunto de posibles defectos a inyectar es limitado. Además el alto nivel de integración de algunos dispositivos limita los puntos de inyección para estos métodos.

Dependiendo de los defectos y su ubicación, los métodos de inyección de defectos pueden clasificarse en inyecciones por hardware con contacto o sin contacto.

Las inyecciones con contacto, son aquellas en las que el inyector y el sistema objetivo esta directamente en contacto físico. Por ejemplo produciendo cambios de voltaje.

Las inyecciones sin contacto, son aquellas en las que el inyector no tiene contacto directo con el sistema de objetivo. En este tipo de inyección una fuente externa produce fenómenos físicos, como radiación o interferencia electromagnética, causando falsas corrientes dentro del chip objetivo.

Inyección con contacto

Un método muy común de inyección de defectos por hardware, es la inyección por contacto directo con los pines del circuito, a menudo denominada inyección a nivel de pin (pin-level injection). Existen dos grandes técnicas de modificación de las corrientes eléctricas y voltajes en los pines:

- **Active probes:** Esta técnica altera la corriente eléctrica del circuito a través de dispositivos que se adjuntan a los pines. Se debe tener cuidado dado que una excesiva cantidad de corriente puede dañar el dispositivo objetivo de la inyección. Otra opción es adjuntar a la fuente de alimentación de hardware dispositivos para inyectar perturbaciones de suministro eléctrico. Sin embargo, esto aumentan el riesgo de una inyección destructiva.

- **Socket insertion:** Esta técnica introduce un zócalo entre el hardware objetivo y la placa del circuito. El zócalo insertado permite controlar que señales analógicas llegan a los pines. Las señal en cada pin puede ser alterada. Por ejemplo se pueden invertir, o realizarse operaciones de and y or con señales de pines adyacentes o con señales previas del propio pin.

Ambos métodos ofrecen gran control del tiempo, lugar y magnitud de la perturbación en la inyección de los defectos. Se debe tener en cuenta que como los defectos son introducidos a nivel de pines, no son idénticos a los que se producen en el interior del chip. Sin embargo, se pueden lograr muchos de los mismos efectos.

Inyección sin contacto

En este tipo de inyección, los defectos se inyectan mediante radiación de iones pesados, o la exposición del hardware a un campo electromagnético. Estos métodos imitan los fenómenos físicos naturales. Sin embargo, es difícil determinar el tiempo de activación y ubicación de la inyección de un defecto. Esto se debe a que no se puede controlar con precisión, el momento exacto de la emisión de iones pesados, o la creación de un campo electromagnético.

3.2.2. Técnicas de inyección de defectos por software

En los últimos años, los investigadores han tenido más interés en el desarrollo herramientas de inyección de defectos implementados en software (SWIFI) [2].

La inyección de defectos de software es una técnica atractiva porque no requiere de hardware costoso. Además, se pueden utilizar para aplicaciones y sistemas operativos, lo cual es difícil de hacer con inyección de defectos por hardware.

Si el objetivo es una aplicación, el inyector puede ser insertado en la propia aplicación, o entre las capas de la aplicación y el sistema operativo. Si el objetivo es el sistema operativo, el inyector debe ser incorporado en el sistema operativo.

Aunque los métodos por software son flexibles, tiene sus deficiencias:

- No puede inyectar defectos en los lugares que son inaccesibles al software.
- El software necesario para el funcionamiento del inyector puede perturbar el trabajo que se ejecuta en el sistema objetivo, e incluso cambiar la estructura de software original. Un diseño cuidadoso del ambiente de inyección puede reducir al mínimo las alteraciones.
- Un alto tiempo de la resolución puede causar problemas. Para fallas de tiempo de latencia grande, tales como fallos de memoria, el tiempo de resolución puede no ser un problema. Para fallas de corta latencia, como las de bus y los fallos de CPU, la técnica puede fallar en la captura ciertos errores de comportamiento, como la propagación.

Podemos clasificar los métodos de inyección de defectos en base a software según cuando se inyectan los defectos: durante tiempo de compilación o durante el tiempo de ejecución.

Inyección en tiempo de compilación

Para inyectar defectos en tiempo de compilación, el programa debe ser modificado antes de que su imagen sea cargada y ejecutada. Esta técnica inyecta errores en el código fuente, o en el código en lenguaje ensamblador del programa

objetivo, para emular el efecto de fallas de hardware o software. El código modificado altera las instrucciones de programa objetivo, generando una imagen del software con errores, y cuando el sistema ejecuta el defecto el mismo es activado.

Un método para este tipo de inyección es el test de mutación, donde se cambian lineas de código del programa para que contengan defectos. Una variante de este es el método de inserción de código, en el cual se agregan lineas de código para alterar el programa. Esto usualmente se hace a través de funciones simples aplicadas a variables del programa, perturbando de alguna manera sus valores.

La técnica de inyección en tiempo de compilación requiere la modificación del programa objetivo, pero no de software adicional durante el tiempo de ejecución. No causa alteración del sistema destino durante la ejecución.

Este método de aplicación es sencilla, pero no permite la inyección de defectos cuando el programa esta corriendo.

Inyección en tiempo de ejecución

En las técnicas de inyección en tiempo de ejecución, es es necesario un mecanismo para activar la inyección. Entre los mecanismos comúnmente usados se encuentran:

- **Por Tiempo de Espera:** Un temporizador expira en un determinado tiempo, provocando la inyección. Al cumplirse el tiempo de espera se genera una interrupción para invocar la inyección del defecto. El temporizador puede ser implementado en hardware o software. Este método no requiere ninguna modificación a la aplicación o programa. La inyección no dependen de eventos específicos o del estado del sistema, por lo que produce efectos impredecibles en el comportamiento del programa. Sin embargo, es un método adecuado para emular defectos intermitentes o transitorios de hardware.
- **Por Excepción / Trampa:** En este caso, una excepción de hardware o trampa, transfiere el control al inyector de defectos. De esta manera se inyectan defectos cuando ciertos eventos o condiciones ocurren. En este método las inyecciones de defectos se vinculan al vector de interrupciones. Por ejemplo, una trampa de software insertada en un programa objetivo invoca la inyección de un defecto antes de que se ejecute una instrucción. Cuando se ejecuta la trampa, se genera una interrupción que transfiere el control a un manejador de interrupción. También una excepción de de hardware puede invocar una inyección por ejemplo ante el acceso a una dirección de memoria.
- **Por Inserción de Código:** En este método, las instrucciones que permiten introducir el defecto son añadidas al programa objetivo antes de instrucciones particulares. A diferencia del método de inserción de código en tiempo de compilación, el código que realiza la inyección del defecto es añadido durante el tiempo de ejecución. El defecto inyectado puede existir como parte del programa objetivo y ejecutar en modo de usuario.

3.2.3. Técnicas de inyección de defectos por simulación

La inyección de defectos basada en simulación consiste en la construcción de un modelo del sistema bajo análisis, incluyendo un detallado modelo de

simulación del procesador en uso.

Esto significa que los errores o fallos de la simulación del sistema se producen de acuerdo con una distribución predeterminada. Los modelos de simulación son desarrollados generalmente utilizando lenguajes de descripción de hardware como el VHDL o Verilog. El modelo debe ser una representación exacta de la realidad del sistema bajo análisis.

Como ventajas, esta técnica no lleva a cabo intrusión en el sistema real. Dependiendo del nivel de detalle se puede llegar a obtener el control total de inyección de faltas por hardware y software, dando el máximo grado de monitoreo y control.

Como desventajas, se necesita un gran esfuerzo para desarrollar el modelo. Esto implica grandes costos y larga duración. La exactitud de los resultados depende de la fidelidad del modelo utilizado. Los modelos pueden no incluir todos los defectos de diseño que se pueden presentar en el verdadero hardware. El tiempo necesario para realizar una simulación es elevado.

3.2.4. Técnicas híbridas de inyección de defectos

El enfoque híbrido combina dos o más técnicas de inyección de defectos para un mejor análisis del sistema.

Por ejemplo, la ejecución de inyección de defectos basada en hardware o software pueden proporcionar el beneficio significativo en términos de tiempo para llevar a cabo experimentos de inyección de defectos, se puede reducir tiempo inicial de la instalación.

El enfoque híbrido puede combinar la versatilidad de la inyección de defectos por software, y la exactitud del monitoreo de hardware; esto es muy adecuado para medir latencias extremadamente cortas.

Dada la significativa ganancia en la controlabilidad y observabilidad con un enfoque basado en la simulación, podría ser útil combinar un enfoque basado en la simulación con uno de los otros a fin de ejercer plenamente el sistema bajo análisis.

Por ejemplo, la mayoría de los investigadores y los profesionales pueden optar por modelar una parte del sistema bajo análisis, como la Unidad de Aritmética y Lógica (ALU) en el microprocesador, en un nivel muy detallado, y realizar experimentos de inyección de defectos basados en simulación por el hecho de que el interior de los nodos de una ALU no son accesibles mediante otro enfoque. En la Figura 3.1 se muestra la clasificación de las técnicas de inyección de defectos.

3.3. Arquitecturas para programas de inyección de defectos

El desarrollo de las herramientas de inyección tiene como punto de partida el diseño de las mismas. Es en esta etapa donde se toman las decisiones que marcan las características futuras de la herramienta. Para facilitar esta tarea es importante contar con modelos de arquitecturas que nos guíen en la construcción de los sistemas de inyección de defectos.

A continuación se presentarán aspectos a tener en cuenta en el diseño de un sistema; posteriormente se describe un modelo de arquitectura, la descripción de sus componentes y su funcionamiento.

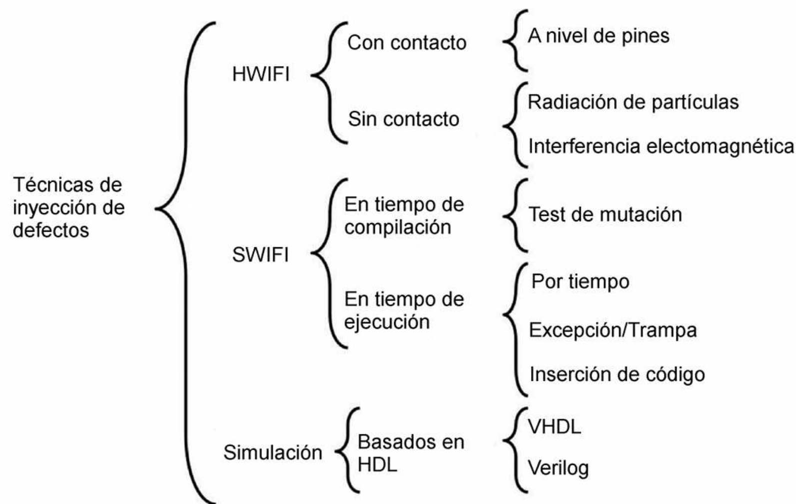

Figura 3.1: Clasificación de técnicas

3.3.1. Características generales en una arquitectura de inyección de defectos

Un sistema de inyección de defectos tiene un conjunto actividades características que deben ser tenidos en cuenta [6].

- En un principio la herramienta debe activar el sistema objetivo. Si el sistema esta inactivo la aparición de un defecto no tendrá consecuencias, ya que solo cuando se requiera por parte del sistema realizar alguna tarea, es que la presencia de una defecto puede ocasionar algún comportamiento inesperado en el mismo.
- Después de realizada la anterior tarea, la herramienta podrá inyectar defectos en el sistema objetivo tal como se haya especificado.
- Tras la inyección, la monitorización del sistema puede comenzar con el fin de verificar que el sistema se comporta como se espera.
- También se debe tener en cuenta que tanto la actividad de inyección como la monitorización deben ser controladas y coordinadas adecuadamente.
- Otro punto a tener en cuenta para un mejor aprovechamiento de una herramienta de inyección es disponer de una interfaz adecuada, donde un usuario puede formular las inyecciones que desea realizar, y a su vez consultar los resultados de la monitorización.

3.3.2. Ejemplo de Arquitectura para una herramienta de inyección de defectos

A continuación se plantea un modelo de arquitectura para herramientas de inyección de defectos [6]. La arquitectura propuesta consta de 5 módulos, los

cuales se muestran en la Figura 3.2

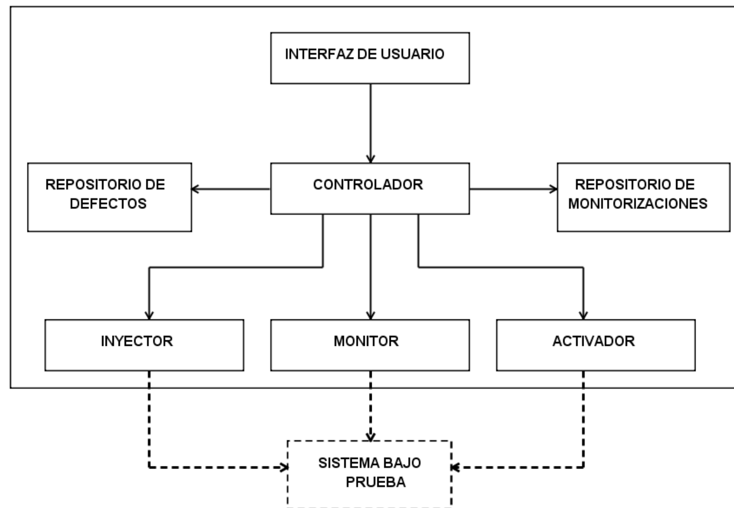

Figura 3.2: Estructura de la solución propuesta

- **Activador:** Este módulo se encarga de activar el sistema objetivo, con el fin de que quede listo para ser inyectado.
- **Inyector:** Se encarga de inyectar los defectos en el sistema bajo prueba.
- **Monitor:** Controla el sistema bajo prueba. Su finalidad es verificar que el sistema funcione como se espera y recabar los datos de la ejecución del mismo.
- **Controlador:** Es el responsable de controlar los distintos módulos de la herramienta, con el fin de que estos realicen sus actividades coordinadamente.
- **Interfaz de Usuario:** Recibe las especificaciones de los usuarios para la ejecución de las inyecciones y devuelve los resultados.

De los módulos arriba descriptos, solo el Inyector, Monitor y Activador se comunican con el sistema bajo prueba. Estos intercambian los mensajes necesarios a través del módulo de control (Controlador), el cual se encarga de la coordinación entre ellos.

El Controlador recibe las especificaciones por parte de la interfaz para realizar las inyecciones necesarias. En estas especificaciones se definen los parámetros que se le enviarán a los otros módulos. Cuando los demás módulos necesitan devolver los datos obtenidos, dichas devoluciones también se realizaran a través del Controlador.

Además de los módulos ya presentados, esta arquitectura cuenta con dos más, cada uno de estos participa en calidad de repositorio de datos, ellos son:

- **Repositorio de defectos:** Este repositorio almacena los defectos a ser inyectados.

- **Repositorio de monitorizaciones:** Se encarga de almacenar el resultado de las monitorizaciones realizadas sobre el sistema objetivo.

Un posible escenario de ejecución de una herramienta de inyección de defectos que sigue el modelo presentado podría ser el siguiente:

1. El módulo Controlador inicia la ejecución de la herramienta de inyección, es decir, inicializa los módulos según sea necesario. De esta forma el sistema esta listo para ser utilizado.

2. Seguidamente, el Controlador recibe por parte de la Interfaz de Usuario las especificaciones necesarias para poder realizar las inyecciones, luego de esto el Controlador almacena las mismas en el Repositorio de Defectos.

3. El controlador le pide al Activador iniciar el sistema bajo prueba. El Activador intenta, y en caso de fracaso los test terminan mostrando al usuario un mensaje adecuado.

4. Si el Activador tuvo éxito, entonces el Controlador le solicita al Repositorio de Defectos el/los defectos a ser inyectados. Los datos sobre estos son pasados al Inyector, el cual tiene como objetivo realizar las inyecciones. Además de esto, este informa al Controlador sobre el estado de la inyección.

5. El Controlador inicia el Monitor, que comenzará a monitorizar el sistema bajo prueba. Los datos obtenidos en el proceso son pasados nuevamente al Controlador, el cual almacena los datos en el Repositorio de Monitorizaciones.

6. Después que todas los defectos fueron inyectados y se reunieron los datos, el Controlador termina el experimento desactivando los módulos que sean necesarios, como por ejemplo el Monitor, ya que solo tiene sentido que este activo mientras dura el experimento.

7. Por último, los datos recogidos pueden ser pasados a la interfaz, a través del Controlador.

Alguna de las ventajas de este modelo son:

- Esta arquitectura permite construir una herramienta claramente modular, cada módulo realiza una tarea especifica de forma que si uno debe ser cambiado es posible reutilizar el código de los demás.

- El modelo presentado centraliza la comunicación entre los módulos en el Controlador. Este se encarga de coordinar los mensajes de manera que se evita el acoplamiento entre los módulos restantes. Este hecho aporta en una mayor simpleza a la hora de implementar e integrar cada uno de ellos.

- Solo los módulos que realmente necesitan comunicarse con el sistema bajo prueba lo hacen, de esta forma se logra un nivel más bajo de perturbación sobre el mismo.

Algunas de las desventajas del modelo presentado son:

- Dado que toda la comunicación se centraliza en el Controlador, potencialmente nos podemos encontrar con un problema de performance en el mismo, lo que puede conllevar a aumentar el nivel de perturbación en el sistema bajo test.

- Dependiendo del tipo de herramienta que se este construyendo el Controlador puede resultar complejo en su implementación, depuración y mantenimiento. Esto se debe a la carga de tareas que realiza.

3.4. Técnicas de inyección de defectos de interés para este proyecto

De la variedad de técnicas de inyección de defectos expuestas en este capítulo nos centraremos en las técnicas de inyección de defectos por software. Más específicamente en las aplicadas en tiempo de ejecución.

Entre las características que las hacen más atractivas encontramos:

- No es necesaria la construcción de hardware adicional por lo que su costo es menor.

- En estas técnicas es posible seleccionar el proceso a ser inyectado. Esto puede ser complejo en técnicas basadas en hardware.

- No requieren de un conocimiento exhaustivo del sistema bajo prueba a diferencia de las técnicas basadas en simulación, o las inyección por software en tiempo de compilación. Esto hace que sean potencialmente más portables.

Capítulo 4

Inyección de defectos sobre `syscalls`

Las llamadas al sistema (syscalls) son el mecanismo proporcionado por el kernel para ofrecer distintos servicios y acceso a los recursos del sistema. Cada vez que un proceso necesita leer de un dispositivo, enviar datos por un socket, memoria dinámica para una variable u otros servicios del kernel se invoca a una syscall.

Que un proceso solicite un servicio mediante una `syscall` no garantiza el éxito de la solicitud, y los escenarios de error deben ser contemplados por los procesos que las invocan.

De esta manera al pensar en la construcción de herramientas que evalúen el correcto funcionamiento de las aplicaciones, ante la ocurrencia de errores o el funcionamiento anómalo del sistema base, las `syscalls` se presentan como un punto estratégico para la inyección de defectos.

La inyección de defectos en las `syscalls` refiere a la posibilidad de decidir sobre el éxito o fallo, e inclusive la modificación de los parámetros de invocación o retorno ante la invocación de una `syscall` por un proceso. Esto permite simular desde la no disponibilidad de un recurso o servicio, hasta el mal funcionamiento del mismo mediante la modificación de los datos de entrada o salida.

En esta sección se detallan las herramientas investigadas referentes a inyección de defectos en `syscalls` y los conceptos necesarios para entender su funcionamiento. También se especifican los prototipos creados con cada tecnología.

4.1. Introducción a `syscalls`

Dado que el hardware es compartido por múltiples procesos, el acceso a los recursos debe ser administrado para garantizar su correcto uso y evitar problemas de seguridad, por ejemplo que procesos malintencionados deseen acceder a recursos asignados a otros procesos. Por estas razones en los sistemas multitarea el acceso al ambiente físico es manejado exclusivamente por el sistema operativo. La única forma de acceder al hardware es a través de las `syscalls` [13].

Por lo tanto, una `syscall` es el mecanismo usado por un proceso para solicitar un servicio al sistema operativo.

Típicamente las aplicaciones no son programadas usando directamente las `syscalls` sino, las operaciones provistas por algún API. En Unix el estándar POSIX[46] define una interfaz para el sistema operativo de forma que una aplicación pueda ejecutarse en distintas plataformas utilizando APIs que implementen dicho estándar. Esto hace que POSIX este fuertemente relacionado con las `syscalls`. GNU/Linux es considerado compatible con POSIX, y el API que provee para el espacio de usuario esta dada por el estándar de C, proporcionada como la librería Glibc.

En la Figura 4.1 se muestra como se disponen la interfaz de `syscalls` ofrecida por GNU/Linux y GLibc respecto al espacio de usuario y el espacio del kernel.

POSIX define el API no las `syscalls`, por lo que define un comportamiento para cada función pero no como debe implementarse. Cada `syscall` pueden constar de uno o varios argumentos de entrada y tienen un retorno de tipo entero que indica error o éxito. Usualmente, pero no siempre, un valor negativo denota un error mientras un valor cero indica éxito. Ante un error se escribe en la variable global `errno` el código que corresponde al mismo. El valor de esta variable se puede traducir a un mensaje comprensible mediante el uso de operaciones como `perror()`.

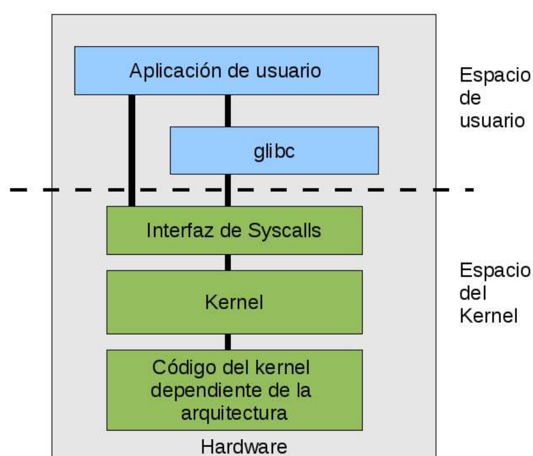

Figura 4.1: Arquitectura GNU/Linux

En GNU/Linux cada `syscall` tiene asignado un número. Este número es único y se usa para hacer referencia a la llamada.

Los procesos se refieren a las `syscall` por dicho número por lo que es importante que no cambie para que aplicaciones compiladas no sean afectadas. Si una `syscall` es removida su número no puede ser reciclado para que código compilado con anterioridad al cambio no llame a una `syscall` por otra. GNU/-Linux provee la llamada `sys_ni_syscall()` que lo único que hace es retornar `-ENOSYS`; este error corresponde a una invocación a una llamada del sistema inválida. `sys_ni_syscall()` es usada para cubrir las situaciones en que una `syscall` es removida o deja de estar disponible.

El kernel conserva la lista de las `syscalls` en la tabla de llamadas al sistema `sys_call_table`. Esta es dependiente de la arquitectura y asigna un número único a cada `syscall` válida.

Los procesos de espacio de usuario no pueden ejecutar `syscalls` directamente debido a que estas existen en el espacio de memoria del kernel. Por ello en espacio de usuario se envía una señal al kernel para indicar que se requiere ejecutar una syscall, luego sucede un cambio a modo kernel para ejecutar la `syscall` en el espacio de memoria del kernel y con los privilegios correspondientes.

La señal enviada es una interrupción de software que provoca el cambio a modo kernel para ejecutar el manejador de la excepción. Este manejador es el **system call handler**, que es una sección de código en ensamblador. En x86 la interrupción de software puede realizarse mediante dos métodos: instrucción `int $0x80` o mediante la instrucción `sysenter`. Sysenter provee una forma más rápida y especializada para pasar a modo de kernel y ejecutar una syscall, se ejecuta de forma similar a system_call pero sin pasar por la IDT (Interrupt Descriptor Table). La IDT contiene descriptores para cada interrupción.

En x86 el número de la `syscall` es pasado al kernel en el registro `eax`. Luego el system call handler lee el valor, chequea la validez del mismo comparando con la cantidad de `syscalls` de la tabla de llamadas al sistema. Si el número es igual o mayor que la cantidad antes indicada se retorna `-ENOSYS` en caso contrario la `syscall` especificada es invocada.

Ante una `syscall` el system call handler sigue los siguientes pasos:

- Guarda el contexto, colocando el contenido de los registros en la pila del kernel.
- Invoca la rutina correspondiente a la syscall.
- Al terminar la rutina de la `syscall` los registros son recuperados del kernel stack y vuelve al modo previo a la invocación.

Cuando se requiere pasar parámetros a las `syscalls` la forma más fácil de hacerlo es a través de los registros. En x86 `ebx ecx edx esi` y `edi` contienen los primeros cinco argumentos. Si fueran necesarios más de cinco argumentos un registro guarda el puntero a espacio de usuario donde se encuentran los restantes parámetros. El valor de retorno de una `syscall` es devuelta al espacio de usuario a través de un registro, en x86 el `eax` [14].

Mientras el kernel ejecuta una `syscall` se encuentra en el contexto del proceso que invoca; el puntero current apunta a una estructura con la información del proceso que invocó la syscall, dicha estructura es la struct task.

4.2. Opciones de inyección sobre `syscalls`

Con el objetivo de lograr la inyección de defectos en `syscalls` nos encontramos con variadas opciones. Entre ellas modificar el kernel para introducir código extra que permita la inyección, utilizar APIs usadas usualmente para debug como son Ptrace[27] o Kprobes[16] y la interposición a bibliotecas mediante LD_PRELOAD[23][24]. En esta sección se exponen los aspectos más importantes de cada una de estas opciones.

4.2.1. Modificación de atención de `syscalls`

Una primera opción para tener control sobre las invocaciones a `syscalls` es modificar la implementación de la atención de `syscalls` que hace el kernel. A continuación se explican métodos que permiten dichas modificaciones y se analiza la factibilidad de desarrollar herramientas de inyección de defectos basados en dichos métodos.

Como modificar la implementación de la atención de `syscalls`

Hay tres puntos claves de la atención de las `syscalls` en los cuales se puede introducir código para obtener el control de las `syscalls`:

1. `syscall table`: sustituir las direcciones de las rutinas de atención de cada `syscall` que se quiera interceptar por la de una rutina implementada para realizar la intercepción.

2. manejador de `syscall` y `sysenter`: como se dijo antes estas rutinas son encargadas de invocar al manejador de la `syscall` correcta. Las modificaciones a estos manejadores implican introducir código extra para interponerse a las `syscalls` de interés.

3. manejador de cada `syscall` específica: introducir código extra en el manejador de las `syscalls` a las que se necesita interponerse.

Las opciones 1 y 3 se ejemplifican en la Figura 4.2.

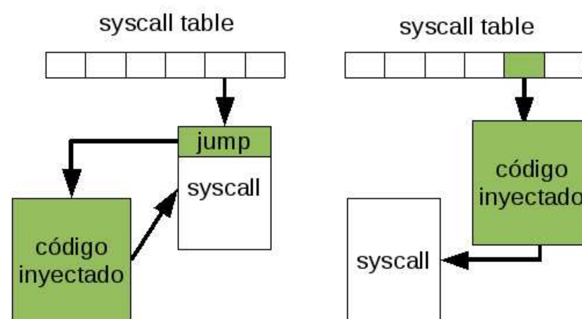

Figura 4.2: Opciones 1 y 3 de modificación a la implementación de las `syscalls`

Las modificaciones antes mencionadas se pueden lograr mediante modificación de los fuentes del kernel y su recompilación, o mediante el uso de técnicas que permiten inyectar código en el kernel sin necesidad de recompilar. Estas últimas son comúnmente usadas por rootkits. Los rootkit son herramientas usadas comúnmente para ganar el control de funcionalidades que permitan obtener privilegios de administrador. Típicamente usan vulnerabilidades del sistema para lograr esto, y están asociados a software mal intencionado y de uso común de hakers.

Mediante un LKM (loadable kernel module) se puede modificar el contenido de la sys_call_table, los manejadores de atención de las syscall o el manejador de una syscall en particular, redireccionando a código inyectado en la memoria. Para ello debe conseguirse la dirección de memoria de la sys_call_table.

En la versión del kernel 2.4 era posible obtener la dirección del la sys_call_table exportándola como símbolo del kernel. Esto se lograba mediante la llamada a extern void *sys_call_table[ ]; desde la versión 2.5.27 del kernel ya no es posible exportar la sys_call_table. Esto hace necesario buscar otra forma para conseguir la dirección de la tabla a las syscalls.

Una opción para encontrar la dirección de la syscall_table es buscarla en el archivo System.map que contiene las direcciones de los símbolos del sistema. El inconveniente de esta opción es que dicho archivo se genera con cada compilación del kernel y la dirección de los símbolos puede cambiar.

Otra opción para GNU/Linux en x86 para obtener la dirección de la sys_call_table es buscar en el archivo entry.S de los fuentes del kernel donde se implementa el manejador de syscalls. En la versión del kernel 2.6.27.7-9 para x86 de 32 bits se define en /usr/src/linux/arch/x86/kernel/entry_32.S. La porción de dicho archivo donde se define el handler de la int 0x80, system_call es el siguiente:

```
...
  # system call handler stub
ENTRY(system_call)
  RING0_INT_FRAME # can't unwind into user space anyway
  pushl \%eax     # save orig_eax
  CFI_ADJUST_CFA_OFFSET 4
  SAVE_ALL
  GET_THREAD_INFO(\%ebp)
  /* Note, _TIF_SECCOMP is bit number 8,
    and so it needs testw and not testb */
  testw \$_TIF_WORK_SYSCALL_ENTRY,TI_flags(\%ebp)
  jnz syscall_trace_entry
  cmpl \$(nr_syscalls), \%eax
  jae syscall_badsys
syscall_call:
  call *sys_call_table(,\%eax,4)
  movl \%eax,PT_EAX(\%esp) # store the return value
...
```

Como se puede apreciar en el código, se hace un call *sys_call_table(, %eax,4) por lo que en esa instrucción se incluye la dirección del la sys_call_table. El código de la instrucción sería algo así: 0xff 0x14 0x85 *"dirección de sys_call_table"*, donde los tres primeros bytes corresponden a la instrucción del call y los siguientes cuatro bytes a la dirección del sys_call_table, de esta manera con la dirección del handler de int 0x80 y buscando de ahí en adelante la secuencia 0xff 0x14 0x85 podemos encontrar la dirección del sys_call_table.

Si lo que se quiere es la dirección del handler de int 0x80 (system_call) para modificarla, esta se encuentra en la IDT. El otro símbolo de interés es el de sysenter, este en la versión del kernel 2.6.27.7-9 para x86 de 32 bits es ia32_sysenter_target y esta exportado como símbolo, con lo que para conseguir la dirección puede hacerse mediante: $ grep ia32_sysenter_target

/proc/kallsyms. Conociendo la dirección de dichos símbolos se puede modificar los handlers de systemcall y sysenter para mediante saltos direccionar las llamadas a syscalls a porciones de código propias. En estas secciones de código se puede manejar la syscall a conveniencia o saltar al handler original.

Características de métodos de inyección basados en modificación de atención de syscalls

La modificación del código del kernel para la interposición a las syscalls tiene como gran desventaja que es fuertemente dependiente de la arquitectura y de la implementación del kernel. Esto hace que una herramienta de inyección que se base en estos métodos tenga portabilidad casi nula.

Para evitar la modificación del código del kernel y su posterior recompilación es que aparecen las técnicas de rootkit. Estas técnicas, debido a que usan vulnerabilidades del sistema para lograr sus objetivos comúnmente no pueden aplicarse según aparecen nuevas versiones del kernel, dado que las vulnerabilidades explotadas comúnmente son corregidas.

4.2.2. Ptrace

La llamada al sistema ptrace proporciona un medio por el que un proceso padre puede observar y controlar la ejecución de un proceso hijo, de esta forma el padre puede examinar y/o cambiar su imagen de memoria o registros [27]. Se usa principalmente en la implementación de programas de control tal como depuradores y en el rastreo de syscalls.

Tenemos dos opciones para comenzar a controlar un proceso hijo a través de ptrace, en una de ellas el padre puede indicar el rastreo llamando a fork y hacer que el hijo resultante realice un PTRACE_TRACEME, seguido (normalmente) por un exec.

El siguiente fragmento de código ejemplifica lo dicho anteriormente.

```
switch ((mpid = fork())) {
    case 0: /* hijo */
       if (ptrace(PTRACE_TRACEME,0,0,0) == -1) {
            perror("ptrace(PTRACE_TRACEME)");
            exit(1);
        }
        exec(...);
        perror("exec");
        exit(1);
    case -1: /* error */
        perror("fork");
        exit(1);
    default: /* padre */
        /* esperamos por una señal*/
        wait(&status);
    break;
}
```

Alternativamente, el padre puede comenzar a rastrear un proceso existente usando PTRACE_ATTACH.

```
rc = ptrace(PTRACE_ATTACH, pid, 0, 0);
```

```
if(rc != -1){
    /* esperamos por una señal*/
    wait(&status);
    ...
} else {
    perror("ptrace(PTRACE_ATTACH)");
    exit(1);
}
```

Mientras está siendo rastreado, el hijo se detendrá cada vez que reciba una señal, aun cuando la señal se haya ignorado (La excepción es `SIGKILL` que tiene su efecto habitual). El padre será informado en su siguiente `wait` y puede inspeccionar y modificar el proceso hijo mientras está parado. A continuación, el padre puede hacer que el hijo continúe, ignorando opcionalmente la señal recibida (o incluso entregando una señal distinta en su lugar). Cuando el padre termina de rastrear, puede terminar el hijo con `PTRACE_KILL` o hacer que se continúe ejecutando en un modo normal sin rastreo mediante `PTRACE_DETACH` [28].

4.2.3. LD_PRELOAD

La generación de programas pasa por la compilación, enlace (linker) y carga. En la etapa de ensamblaje se crea un archivo ejecutable a partir de los códigos objeto. En esta etapa se resuelven las referencias externas y las posiciones relativas de los símbolos en los diferentes módulos.

En el enlace estático las bibliotecas compartidas son incorporadas al archivo ejecutable generado. Por ejemplo `/usr/lib/libc.a`. Esto hace que un programa no dependa de ninguna librería, por lo que su distribución es más fácil.

En el enlace dinámico las bibliotecas compartidas son cargadas en tiempo de ejecución. Por ejemplo `/lib/libc.so`. La finalidad de esto es el ahorro de memoria dado que las bibliotecas dinámicas se cargan una única vez en la memoria principal. El comando `ld` es el linker de GNU. `ld.so/ld-linux.so` es el cargador de bibliotecas dinámicas. Carga las bibliotecas compartidas necesarias para un programa, prepara el programa para ser ejecutado y lo ejecuta. A menos que se especifique la opción -static a ld durante la compilación, lo que implica que el enlace se hace de forma estática.

Con la variable de entorno LD_PRELOAD se puede especificar una lista de bibliotecas compartidas ELF para ser cargadas antes que las otras. Esto puede ser usado para ignorar funciones en otras bibliotecas compartidas [23][24]. Lo anterior nos brinda la posibilidad de crear bibliotecas que implementen funciones del sistema, y hacer que dichas funciones sean utilizadas por los programas en lugar de las originales. Esto permite la interposición a las `syscalls` de un programa compilado, agregándole comportamiento extra a las funciones implementadas.

Como se deduce de lo antes mencionado, la interposición a las `syscalls` solo será posible si el programa a ejecutar usa linkeo dinámico y además llama a las operaciones implementadas. Las bibliotecas pueden tener funciones internas que si no son publicadas o si se resuelven de forma estática no pueden ser interceptadas.

FIG

En el estudio de técnicas de inyección mediante interposición de bibliotecas nos encontramos con FIG (Fault Injection in GLibc). FIG es una herramienta de inyección de defectos sobre sistemas mediante la interposición a llamadas a funciones de libc [29]. Para ello se vale de LD_PRELOAD y la generación de stubs de las funciones provistas por libc que proveen la posibilidad de inyectar defectos y registrar los mismos. Fue desarrollado en el 2001 en la UC Berkeley como proyecto de grado de Pete Broadwell, Naveen Sastry y Jonathan Traupm para el proyecto ROC (Recovery-Oriented Computing) [30].

Formas de usar FIG

1. versión stand-alone: se genera la librería `libfig.so`, esta se debe colocar junto al archivo control.out y setup. Setup es un script que inicializa las variables de entorno `LD_PRELOAD`, `LIBFIG_CONTROL` (ruta al archivo de control), y `LIBFIG_OUTPUT` (ruta al archivo de log). Luego de ejecutado setup, para cualquier programa que se corra en la misma consola serán interceptadas las funciones que invoque de forma dinámica, para la que se haya implementado un stub.

2. la versión "wrapper": en este modo lo que se genera es el wrapper fig y la librería libfig.so que deben colocarse junto a el archivo control.out. Luego para ejecutarlo se ejecuta el comando:
   `fig [opciones] programa [argumentos del programa]`
   La descripción de las diferentes opciones pueden verse ejecutando el comando:
   `fig -h`

El archivo control.out es donde se define la campaña de inyección. Los defectos que son posible inyectar con FIG se configuran de cualquiera de estas dos maneras:

- `callnumber [código de operación] return [código de error retornado] errno [errno, si es aplicable] probability [probabilidad de inyectar entre 0 y 1]`

- `callnumber [código de operación] return [código de error retornado] errno [errno, si es aplicable] interval [primer llamada a la operación interceptada] to [ última llamada a la operación interceptada, puede usarse 'infinity']`

De esta manera cada defecto a inyectar solo refiere a una función específica de Libc. Ante una llamada a una función para la que hay definido un defecto, si el mismo debe aplicarse se retorna el error especificado; en caso contrario se procede con la llamada a la función.

FIG no implementa stubs para todas las funciones de Libc, por lo que para extender esta herramienta los mismos deben ser implementados. Esto no es complejo dado que la lógica es idéntica en cada uno. Aprovechando esto FIG brinda un script para la generación automática de stubs. Este requiere definir en un archivo las funciones para las que se desea generar stubs, especificando el símbolo secundario (strong symbols) correspondiente a cada función y los

parámetros de las mismas. La definición de símbolos para las funciones depende de la distribución y versión de Libc; inclusive dependiendo de la versión de Libc muchas de las funciones ofrecidas no tienen definidos de forma pública símbolos secundarios, por lo que los stubs para dichas funciones solo pueden implementarse manualmente mediante el uso de `dlsym()` para obtener un puntero a la función original. Esto limita el uso de generador automáticos de stubs, además de que los stubs generados son poco portables.

En la ejecución de FIG cada stub llama a una función que revisa la configuración de la campaña actual de inyección y retorna si debe o no realizarse la inyección. La carga de la configuración de la campaña de inyección es realizada en la primera llamada a una función de Libc para la que se haya instrumentado un stub. Los datos de la campaña quedan almacenados en memoria de la aplicación, y son accedidos y actualizados ante cada llamada de los stubs.

En la Figura 4.3 se muestran un esquema de como se relaciona la aplicación bajo prueba con FIG y el sistema operativo.

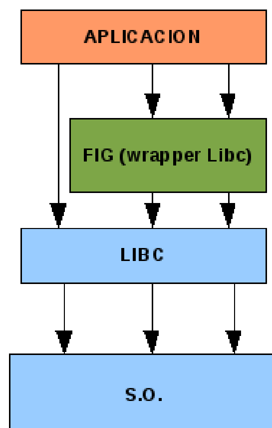

Figura 4.3: Esquema de FIG

Posibilidades de extender FIG

Es de nuestro interés generar una herramienta que brinde más posibilidades de inyección, como ser inyectar dependiendo de los parámetros de la función, cambiar las variables de retorno de las operaciones, o poder realizar invocaciones a las funciones cambiando los parámetros. Para extender FIG de manera que cumpla con estas exigencias se debe re-definir el mecanismo de atención de las llamadas y de definición de defectos para incluir cambios a parámetros o valores de retorno y la posibilidad de decidir en base al contexto de la llamada. Esto implica además agregar mayor lógica a cada stub.

Otra falencia de FIG es la ausencia de mecanismos de mutua exclusión, que garanticen la correcta gestión de los datos de la campaña de inyección ante un escenario multi-hilo, donde pueden ser llamadas de forma simultánea varias funciones que actualicen dichos datos.

Tampoco brinda la posibilidad de agregar defectos luego de empezada una campaña, identificar o gestionar inyecciones a distintos hilos de un mismo programa, ni funcionalidades de monitoreo más allá del registro de inyecciones realizadas.

Los cambios necesarios para extender FIG para obtener una herramienta de mayor utilidad son de importancia. Pero no anulan totalmente las ventajas de extender una herramienta que ya tiene una lógica definida, con la respectiva ganancia de tiempo de implementación e investigación.

4.2.4. Kprobes

Kprobes es un mecanismo que permite insertar dinámicamente puntos de ruptura (breakpoints) en casi cualquier parte del kernel de linux. El estado de la ejecución en ese momento puede ser inspeccionado a través de handlers que kprobes proporciona al usuario de este mecanismo, estos handlers se ejecutan inmediatamente antes que ejecute dicha instrucción e inmediatamente después de ejecutar la misma. Por lo tanto, para definir una kprobe se debe especificar una dirección dentro del kernel, (que puede estar definida de manera absoluta o a través del nombre de su símbolo) y proporcionar una implementación para los dos handlers que se dispone, llamados pre-handler y post-handler [16].

Arquitectura de Kprobe

La Figura 4.4 describe la arquitectura de kprobes.

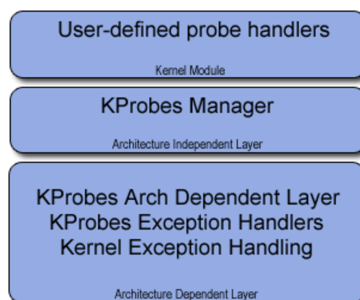

Figura 4.4: Arquitectura de KProbes [17]

En esta, se puede diferenciar tres capas:

1. la primera hace referencia al API que kprobes proporciona para manipular las kprobes definidas. por ejemplo, registrar y desregistrar kprobes, habilitar o deshabilitar las mismas etc.

2. la segunda básicamente se encarga de implementar el registro y desregistro de las kprobes a través de la modificación de la instrucción alojada en la dirección especificada, posterior a esto, el Manager se encarga de invocar de manera adecuada al pre-handler y post-handler según corresponda.

3. la tercera es una capa dependiente de la arquitectura que proporciona funcionalidades a la capa superior de forma que lo referente a la manipulación de las kprobes sea lo más independiente posible de la arquitectura.

Interfaz KProbes

La siguiente estructura define la interfaz que KProbes proporciona, la misma está definida bajo ¡linux/kprobes.h¿.

```
struct kprobe {
    struct hlist_node hlist;            /*Internal*/
    kprobe_opcode_t addr;               /*Address of probe*/
    kprobe_pre_handler_t pre_handler;   /*Address of pre-handler*/
    kprobe_post_handler_t post_handler; /*Address of post-handler*/
    kprobe_fault_handler_t fault_handler;/*Address of fault handler*/
    kprobe_break_handler_t break_handler;/*Internal*/
    kprobe_opcode_t opcode;             /*Internal*/
    kprobe_opcode_t insn[MAX_INSN_SIZE]; /*Internal*/
};

typedef int (*kprobe_pre_handler_t)(struct kprobe*, struct pt_regs*);

typedef void (*kprobe_post_handler_t)(struct kprobe*,
    struct pt_regs*, unsigned long flags);

typedef int (*kprobe_fault_handler_t)(struct kprobe*, struct pt_regs*,
    int trapnr);
```

Como se muestra en la definición anterior, las funciones `pre_handler` y `post_handler` reciben como parámetro la estructura `kprobe` que provocó la invocación, también en estas se recibe como parámetro los registros salvados del contexto donde se encontraba la ejecución. Para el caso de `pre_handler` si el valor retornado es 0, entonces se habilita la invocación de la función `post_handler` luego que ejecute la instrucción asociada al breakpoint, en caso contrario esta no será invocada [17]. La flags en la función `post_handler` hasta el momento no tiene uso.

Existe una tercera función que es invocada en caso de que ocurra alguna falla durante la ejecución de `pre_handler`, `post_handler` o durante la ejecución de la instrucción asociada al breakpoint.

```
int register_kprobe(struct kprobe *p);
int unregister_kprobe(struct kprobe *p);
```

Estas funciones son las encargadas de registrar y desregistrar una kprobe. Partiendo de la base y apoyándose en KProbes, se implementan mecanismos similares a este último que facilitan aun más la tarea de inspeccionar a través de breakpoints la ejecución del kernel de linux. Son el caso de JProbe y KRetProbe

JProbe

Este mecanismo permite inspeccionar de manera más "natural" los argumentos de una función que se ejecuta dentro del kernel. En el momento inmediatamente antes de que se ejecute la instrucción asociada al breakpoint el mecanismo de JProbe invoca a un handler con la misma firma que la función original que esta por ejecutar.

Para utilizar JProbe se dispone de la siguiente estructura

```
struct jprobe {
    struct kprobe kp;
    void *entry;    /* probe handling code to jump to */
};
```

y las siguientes funciones para registrar y desregistrar JProbes

```
int register_jprobe(struct jprobe *p);
void unregister_jprobe(struct jprobe *p);
```

A continuación se muestra un ejemplo de funcionamiento de JProbe para inspeccionar los argumentos de la llamada `sys_read` en el kernel de linux.

```
asmlinkage ssize_t jsys_read(unsigned int fd, char __user * buf,
    size_t count)
{
    //codigo
}

static struct jprobe my_jprobe = {
    .entry = jsys_read,
    .kp = {
        .symbol_name = "sys_read",
    }
};

ret = register_jprobe(&my_jprobe);

if (ret < 0) {
  printk(KERN_ERR "register_jprobe failed, returned %d\n", ret);
}else {
  printk(KERN_INFO "register_jprobe success, returned %d\n", ret);
}
```

Con esto se logra de manera sencilla inspeccionar los argumentos de la llamada read cada vez que un proceso de usuario realiza dicha llamada. Cuando se quiera detener la inspección alcanza con ejecutar la siguiente función.

```
unregister_jprobe(&my_jprobe);
```

KRetProbe

Este mecanismo permite insertar breakpoint en cualquier función del kernel de GNU/Linux y lograr capturar el momento inmediatamente antes a que comience a ejecutar dicha función y el momento inmediatamente antes a que retorne. Para utilizar KRetProbe se dispone de la siguiente estructura

```
struct kretprobe {
    struct kprobe kp;
    kretprobe_handler_t handler;
    kretprobe_handler_t entry_handler;
    int maxactive;
    int nmissed;
    size_t data_size;
    struct hlist_head free_instances;
    spinlock_t lock;
};
```

Los campos principales de esta estructura son los siguientes:

- **kp:** Este campo sirve para especificar la dirección o el nombre del símbolo de la función que se quiere monitorizar.
- **entry_handler:** Es el manejador que se invoca en el momento anterior a que comience a ejecutar la función que se está monitorizando.
- e**entry_handler:** Es el manejador que se invoca en el momento anterior al retorno de la función que se está monitorizando.
- **maxactive:** Cantidad máxima de invocaciones concurrentes que se permiten
- **data_size:** Especifica el tamaño de un buffer de datos privado que se necesite utilizar, el mecanismo de KRetProbe se encarga de solicitar su memoria y dejar disponible su acceso en ambas funciones.

Al momento de comenzar y retornar de la función que se está monitorizando se invoca para la KRetProbe asociada una función con la siguiente firma

```
typedef int (*kretprobe_handler_t) (struct kretprobe_instance*,
    struct pt_regs *);
```

Donde la estructura kretprobe_instance es la siguiente:

```
struct kretprobe_instance {
    struct hlist_node hlist;
    struct kretprobe *rp;
    kprobe_opcode_t *ret_addr;
    struct task_struct *task;
    char data[0];
};
```

A continuación se mencionan sus principales campos

- **rp:** Puntero a la kretprobe asociada.
- **task:** Puntero a la estructura que define el proceso que está realizando la invocación a la función monitorizada.
- **data:** Datos privados en caso de que se haya definido.

Las siguientes funciones son necesarias para registrar y desregistrar KRetProbe

```
int register_kretprobe(struct kretprobe *rp);
void unregister_kretprobe(struct kretprobe *rp);
```

A continuación se muestra un ejemplo de funcionamiento de KRetProbe para inspeccionar los argumentos de la llamada sys_read en el kernel de linux.

```
static int entry_handler(struct kretprobe_instance *ri,
    struct pt_regs *regs)
{
    //codigo
```

```
}
static int ret_handler(struct kretprobe_instance *ri,
    struct pt_regs *regs)
{
    //codigo
}
...
...
static struct kretprobe my_kretprobe = {
    .handler = ret_handler,
    .entry_handler = entry_handler,
    .data_size = SIZE_BUFFER,
    /* Probe up to 20 instances concurrently. */
    .maxactive = 20,
};
...
...
my_kretprobe.kp.symbol_name = "sys_write";
ret = register_kretprobe(&my_kretprobe);
if (ret < 0) {
    printk(KERN_ERR "register_kretprobe failed, returned %d\n", ret);
}else {
    printk(KERN_INFO "register_kretprobe success, returned %d\n", ret);
}
```

Cuando se quiera detener la inspección alcanza con ejecutar la siguiente función

```
unregister_kretprobe(&my_kretprobe);
```

4.3. Prototipos de inyección sobre `syscalls`

Para evaluar cada una de las opciones de inyección sobre `syscalls` investigadas se realizaron prototipos. En esta sección se describen dichos prototipos y la evaluación de los mismos. Las pruebas y resultados obtenidos para cada prototipo se encuentran en los apéndices A, B y C.

4.3.1. Prototipo Ptrace

Para nuestro propósito utilizamos ptrace para rastrear y controlar las `syscalls` que un proceso realiza, de este modo cada vez que un proceso controlado realiza una llamada al sistema el proceso controlador tendrá la posibilidad de interponerse a la misma, y de este modo realizar determinados cambios, tanto en parámetros como en el resultado, que permitan implementar la inyección del defecto deseado.

La llamada ptrace es capaz de notificar al proceso controlador inmediatamente antes de que comience a ejecutar la llamada al sistema invocada por el proceso controlado. De esta forma es que el proceso controlador puede realizar las distintas acciones que encuentre pertinente y permitir que la ejecución continúe normalmente e incluso cambiar la llamada por otra distinta.

Ptrace también es capaz de notificar inmediatamente después que una llamada al sistema termina y antes de devolver el control al proceso que la invoco,

de esta forma nos permite tener un mayor control sobre los cambios que son necesarios realizar para implementar la inyección del defecto en cuestión.

Estrategia a seguir

En GNU/Linux las `syscalls` se implementan a través de interrupciones de software (más precisamente la int 80h), además, el mecanismo utilizado para el pasaje de parámetros es a través de los registros del CPU. La llamada ptrace nos proporciona la siguiente estructura para representar los mismos en arquitecturas x86.

```
struct pt_regs {
    long ebx;
    long ecx;
    long edx;
    long esi;
    long edi;
    long ebp;
    long eax;
    int  xds;
    int  xes;
    int  xfs;
    /* int  gs; */
    long orig_eax;
    long eip;
    int  xcs;
    long eflags;
    long esp;
    int  xss;
};
```

GNU/Linux soporta hasta 6 argumentos para las `syscalls`. Ellos son pasados en **ebx**, **ecx**, **edx**, **esi**, **edi** (y **ebp** usado temporalmente).

El registro **eax** es utilizado para indicar el número de la llamada al sistema que se desea invocar, luego que ésta finaliza GNU/Linux devuelve el código de retorno en ese mismo registro. Cuando la llamada no ha tenido éxito, el valor devuelto es negativo. Normalmente los procesos realizan las llamadas a través de una biblioteca (Glibc) para facilitar y ocultar los detalles de bajo nivel, en este caso si el valor devuelto es negativo, la biblioteca copia dicho valor sobre una variable global llamada errno y devuelve -1 como valor de retorno de la función. Aun así, algunas llamadas realizadas con éxito pueden devolver un valor negativo (p.ej. lseek). La variable errno contiene el código de error de la última llamada que falló. Una llamada que se realice con éxito no modifica errno.

A continuación se analizan algunas opciones para realizar la implementación de los distintos defectos a inyectar. A modo de ejemplo se utiliza la llamada open para exponer los distintos casos, intentando lograr que la misma fracase cuando el proceso controlado intenta ejecutarla, es decir, que al realizar un open sobre un archivo que existe y para el cual se tienen los permisos necesarios, la misma retorne por ejemplo `ENOENT` indicando que el archivo no existe. Vale la pena mencionar que si bien el análisis se basa en la llamada `open()` el mecanismo es similar para otras `syscalls`.

- En el momento de realizar la inyección o sea inmediatamente antes que se ejecute la `syscall` se puede colocar en el registro `ORIG_EAX` el valor 223, este valor corresponde a una `syscall` no implementada, por lo que cuando se retorna de ésta se tendrá en el registro eax el valor 38 (`ENOSYS`) que indica que esa llamada no está implementada y de esta forma lograremos no haber ejecutado la syscall. Luego inmediatamente antes de retornar de la llamada se coloca el valor `-ENOENT` en el registro `eax`. El problema surge en que en versiones recientes de ptrace en este caso NO se obtiene el resultado esperado, o sea `errno = ENOENT` aunque si el valor de retorno -1 ya que la biblioteca Glibc o la llamada mismo, vuelve a cambiar ese valor en el registro `eax` a `ENOSYS`, obteniendo como resultado al culminar la `syscall errno=ENOSYS` y valor de retorno -1.

- En el momento de realizar la inyección o sea inmediatamente antes que se ejecute la `syscall` no se realiza ninguna acción, por lo tanto la `syscall` se ejecutara normalmente, pero inmediatamente antes de salir de la misma se puede colocar el valor `-ENOENT` en el registro eax, provocando que errno valga ENOENT y el valor de retorno sea -1. Como desventaja se obtiene que en la tabla de archivos abiertos se tiene registrado el archivo recientemente abierto, pudiendo influenciar en futuros intentos de abrir ese archivo, por ejemplo para escritura si es que anteriormente se lo abrió para escritura.

- Otra alternativa es que inmediatamente antes de que se ejecute la `syscall` cambiar el valor del registro ebx (puntero al nombre del archivo) a `NULL`, por lo tanto la llamada a open va a fallar y no va a registrar ningún file descriptor en la tabla de archivos abiertos, y luego inmediatamente antes de salir de la `syscall` se coloca el valor `-ENOENT` en el registro eax para asignar de este modo a errno el valor `ENOENT` y tener un valor de retorno igual a -1.

La ultima alternativa parece ser la más adecuada y es en ella que se ha basado la construcción del prototipo.

Arquitectura del prototipo

Tomando como referente lo explicado en el capítulo correspondiente a la arquitectura (2.3), el prototipo cuenta con los módulos Controller, Repository, Monitor e Injector. En la Figura 4.5 se muestra un esquema de la arquitectura del prototipo.

1. **Controller:** Encargado de inicializar los módulos del sistema prototipo, es decir, realiza la carga de los defectos e inicializa el proceso de monitoreo e inyección.

2. **Repository:** Es donde se almacenan las Faults que componen la campaña de inyección.

3. **Monitor:** Se encarga de monitorizar el sistema bajo prueba, quedando a la espera que este realice alguna llamada al sistema, cuando este es el caso notifica a las Faults que se encuentran en el Repositorio para que estos decidan si es su turno de inyectarse, luego haciendo uso de los servicios del módulo Injector se lleva cabo la inyección.

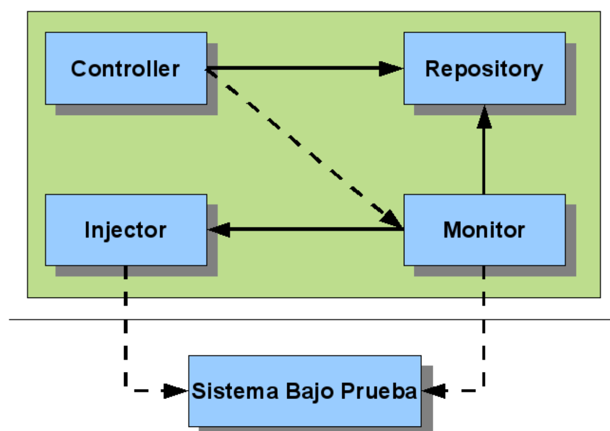

Figura 4.5: Arquitectura del prototipo realizado con Ptrace

4. **Injector:** Encapsula los detalles de bajo nivel que se pueden encontrar en ptrace, proporcionando a quien utilice sus servicios un API más amplia y fácil de utilizar.

Defectos implementados

Todos los defectos implementados extienden de la clase `SysCallFault` que a su vez extiende de `Fault`, esta última proporciona la operación `notify()` que se debe invocar cada vez que ocurre una `syscall` por parte del sistema bajo prueba. La operación `notify()` simplemente verifica que la `syscall` que está en curso es la misma `syscall` para la que se definió el Defecto, por último determina si se está ante una situación de entrada hacia una `syscall` o ante una situación de salida invocando a las operaciones `startSysCall()` y `endSysCall()` redefinidas por `SysCallFault()`.

Por lo tanto la clase SysCallFault solo se debe encargar de las tareas correspondientes a cuando se está ante una situación de entrada de una `syscall` o ante una salida de la misma.

La clase SysCallFault contiene parámetros comunes a todos los Defectos que se implementan, permitiendo abstraer la lógica de la inyección del Defecto de cuando es el momento de ser inyectado según la parametrización dada, por ejemplo: Si el Defecto deber ser inyectado a la tercera vez que ocurra la llamada o el defecto debe ser inyectado una vez o siempre a partir de la primer inyección. Por lo tanto cada Defecto implementado solo se debe preocupar del cometido que debe cumplir.

Las clases ReadFault y WriteFault encapsulan los detalles de cómo se realiza una inyección correspondiente a una lectura y escritura respectivamente.

Por lo tanto, extendiendo las anteriores, las clases ReadSocketFault, ReadFileFault, WriteSocketFault y WriteFileFault encapsulan e implementan los detalles de cómo realizar la inyección de los Defectos que definen.

Además se implementa FileNotFoundFault y CloseErrorIOFault que imple-

mentan los Defectos correspondientes a la apertura y cierre de un archivo.

En la Figura 4.6 se muestra la jerarquía de defectos implementados.

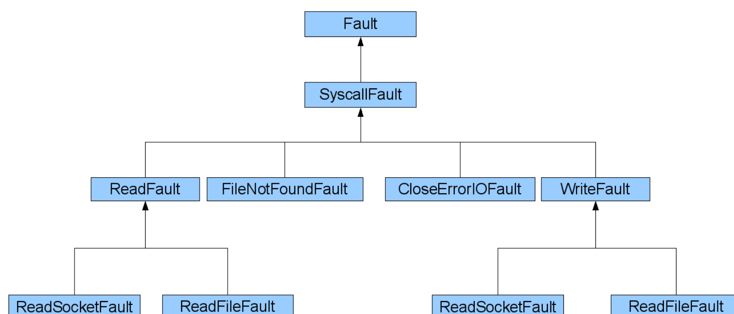

Figura 4.6: Jerarquía de los defectos implementados en el prototipo Ptrace

Algunas Ventajas y Desventajas al utilizar ptrace

Ventajas:

- Fácil de utilizar: Con pocas líneas de código ptrace ya permite realizar acciones sobre el proceso controlado.
- Buen control sobre el proceso controlado: Luego que se tiene el control, con ptrace se puede tanto inspeccionar como modificar desde los registros y espacio de memoria hasta el propio código del proceso controlado.
- Portabilidad: La llamada ptrace se encuentra disponible en todos los sistemas Linux. De todas maneras cabe aclarar que existe un pequeño conjunto de funcionalidades no standard de ptrace que pueden contar con alguna particularidades en su implementación dependiendo de la versión, o implementadas solo por algunos sistemas o en determinadas versiones (como `PTRACE_O_TRACEEXIT` implementada a partir del kernel 2.5.60). Esto no afecta la portabilidad de la solución implementada con ptrace dado que solo utiliza funcionalidades standard.

Desventajas

- Performance: Dado que realizar una acción con ptrace implica realizar una llamada al sistema, en algunos casos dependiendo del proceso que se está controlando el overhead puede ser significativo y de este modo verse perjudicada la performance del proceso controlado.
- Dependencia de la Arquitectura: Dado que ptrace es una llamada de "bajo nivel", utilizar la misma requiere tener conocimiento de la arquitectura para la cual se está programando para el acceso a los registros, y a menos que se escriba código adicional para lograr una aplicación independiente el código quedara dependiente de la misma.

4.3.2. Prototipo LD_PRELOAD

A continuación se especifican detalles del prototipo que realiza inyección de defectos basada en interposición de Libc usando LD_PRELOAD. El objetivo que se siguió al construir el mismo es ahondar más en los detalles de la implementación para evaluar sus ventajas y desventajas. Fue desarrollado usando C y C++, con el compilador gcc 4.3.2, sobre el SO openSUSE 11.1 con la versión de Glibc 2.9

Estructura

Se decidió para el prototipo dividir la aplicación en dos componentes para estudiar un enfoque diferente al que se observa en FIG. El primero es la librería destinada a interponerse a Libc y el segundo el software de inyección. En la Figura 4.7 se muestra como se relacionan la aplicación bajo prueba con ambos componentes del prototipo y el sistema operativo.

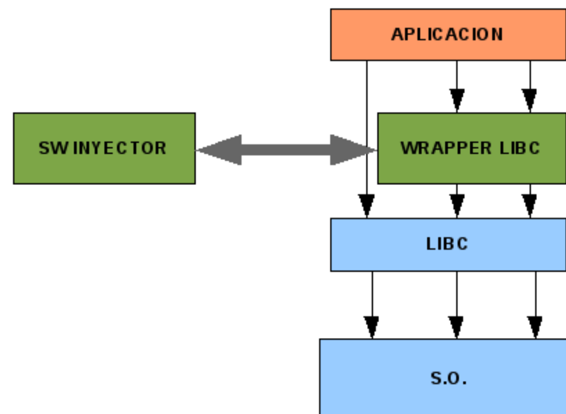

Figura 4.7: Esquema del prototipo LD_PRELOAD

La librería generada contiene los stubs de las funciones de Libc que serán interceptadas, y cada uno de estos stubs tiene la responsabilidad de comunicar al software de inyección cuando se produce una llamada a dicha función y en qué contexto (parámetros y pid del proceso). Además se encarga de tomar la respuesta brindada por el software de inyección y aplicar la decisión que el mismo haya tomado. Esta decisión refiere a si se debe retornar error, cambiar parámetros de la llamada o de retorno de la misma, o simplemente ejecutar la llamada de forma normal.

El software de inyección es el encargado de administrar la campaña de inyección, manteniendo los datos de la misma y realizando la toma de decisiones sobre la inyección de las llamadas a funciones de Libc que son interceptadas.

La interacción entre ambos componentes es mediante memoria compartida. Se define una estructura donde los stubs dejan la información referente a las llamadas interceptadas y el software inyector la respuesta a ésta que será interpretada por el stub para realizar la acción pertinente.

El software inyector se compone de cuatro elementos principales que son el Controller, Injector, Repository y Faults.

1. Controller: encargado de inicializar el Injector y Repository, realizando la carga de los defectos e inicializando el proceso de inyección.

2. Injector: encargado de atender los mensajes provenientes de los stubs, notificar a las Faults definidas en la campaña de inyección y enviar la respuesta al stub.

3. Repository: es donde se contienen las Faults que componen la campaña de inyección.

4. Faults: cada tipo de fault representa un defecto. Las mismas contienen la definición de cuando debe ser inyectado un defecto y que acción debe realizarse.

Funcionamiento

Para poner en marcha la inyección se debe primero poner en marcha el software de inyección. Este despliega un menú indicando al usuario que seleccione el tipo de defecto que se desea inyectar de una lista. Luego de elegida la opción se carga el defecto indicado y se inicializa el Injector que queda esperando los mensajes de los stubs.

Luego en una consola debe cambiarse la variable LD_PRELOAD con la ruta a la librería wrapper de Libc (`new_libc.so`), y correr la aplicación bajo test.

Ante cada llamada de la aplicación a una función que tenga un stub definido en la librería wrapper, el stub correspondiente creará un mensaje con los datos de la llamada. El Inyector notificará a los defectos del repositorio y si alguno debe activarse retornará un mensaje adecuado. Dicho mensaje se copia a memoria compartida permitiendo continuar al stub para que interprete la respuesta. Es así que la lógica de inyección está repartida entre los stubs y los defectos.

Stubs

Los stubs implementados cumplen con el siguiente formato:

```
nombrefuncion(...){//cabezal de la funcion de libc
    //se crea el mensaje m indicando funcion, parametros
    //y pid del proceso y se obtienen la respuesta
    sendMessage(m);
    switch(m->error){//se interpreta la respuesta
    case 0:
        //continua sin error llamando a la funcion con los
        //parametros originales
    case [Codigo de Inyeccion 1]:
        //manejo necesario antes de la llamada
        //llamada a la funcion
        //manejo posterior a la llamada
        ...
    default:
        //si no continua de forma normal y no es
```

```
            //inyeccion la respuesta indica retornar un error
            val =-1;
            errno=m->error;
        }
        return val;
    }
```

Al interpretar la respuesta hay tres acciones posibles, continuar, inyección y retorno de error.

Para realizar la llamada a la función original se obtiene la dirección de esta usando la función `dlsym()` con el parámetro `RTLD_NEXT` y el nombre de la función. De esta forma dlsym retorna la siguiente dirección obtenida para el símbolo (nombre de función), y este se corresponde al de la función provista por Libc dado que se especificó con LD_PRELOAD que al resolver los símbolos se usase primero la librería que contiene los stubs.

En los casos que se especifica inyección esta debe ser interpretada. Esto puede implicar acciones previas a la invocación de la función como pueden ser cambios de parámetros de entrada, o acciones a realizar luego de llamar a la operación como cambios en el retorno o parámetros de salida. Cuando se especifica que se debe retornar un error solo se asigna un valor a la variable errno y se retorna código de error.

La implementación de los stubs se encuentra en library/new_libc.c

Defectos implementados

Todos los defectos implementados extienden de la clase Fault. Esta contiene parámetros de configuración y estado que permite decidir si el defecto debe ser inyectado. Los parámetros de configuración incluyen variables que indican cuando debe aplicarse la inyección y el código de error o inyección retornado. El estado incluye la cantidad de veces que fue llamada la función a la que aplica el defecto. Para cada defecto en particular se agregan parámetros adicionales necesarios.

Todos los defectos cuentan con las funciones privadas:

- `bool isInyect(Event* e)`: La función isInyect encapsula la condición de disparo del inyección, y retorna true solo si se debe aplicar el defecto para el evento e.
- `void createResponse(Event* e, Response *r)`: Función encargada de construir la respuesta r correspondiente al evento e.

Y la función pública:

- bool notify(Event* e, Response* r): Esta operación es utilizada para notificar al defecto de la ocurrencia de un evento, realizando la actualización del estado del defecto y generando una respuesta de error o inyección adecuada de ser necesario.

Los tipos de defectos que se implementaron para este prototipo son sobre file-system y sobre conexiones de red. Los errores sobre file-system comprenden retornar errores de entrada salida al intentar abrir un archivo y modificar una cadena en las operaciones de escritura o lectura.

Los errores sobre conexiones comprenden modificar una cadena en las operaciones de escritura o lectura de un socket. Estos defectos pueden ser configurados de manera que se apliquen a cualquier conexión/archivo o a uno en particular, en todas las llamadas o a partir de cierto número de llamadas.

El primer inconveniente que se presentó al definir los defectos fue que Libc ofrece varias funciones que posibilitan cumplir con tareas de apertura o cerrado de archivos y lectura o escritura de archivos y sockets (por ej. para lectura `read`, `fread`, `fscanf`, `fgets`, `etc`). Esto hace que realizando un defecto por función, no sea posible tener control sobre, por ejemplo, la lectura de archivos de forma genérica. Para ello se implementaron para este prototipo defectos que aplican a varias funciones.

A continuación se detallan las clases implementadas para cubrir los defectos antes mencionados:

- Fault: clase abstracta que representa un defecto.
- FaultFuncion: clase abstracta que representa un defecto que aplica a una función específica. Extiende a Fault. No se implementaron este tipo de defectos por considerarlos un caso particular de las siguientes categorías.
- FaultKindFunction: clase abstracta que representa un defecto que se aplica sobre un conjunto de operaciones. Extiende Fault.
- FaultOpen: clase que representa un defecto sobre las operaciones de apertura de archivos. La implementación de la misma aplica a las operaciones `open()`, `fopen()` y `fdopen()` sobre archivos.
- FaultClose: clase que representa un defecto sobre las operaciones de cierre de archivos. La implementación de esta aplica a las operaciones `close` y `fclose` sobre archivos.
- FaultRead: clase que representa un defecto sobre la lectura de archivos. La implementación de esta aplica a las operaciones `read` y `fread` sobre archivos.
- FaultWrite: clase que representa un defecto sobre la escritura de archivos. La implementación de esta aplica a las operaciones `write()` y `fwrite()` sobre archivos.
- FaultReadSocket: clase que representa un defecto sobre la lectura de sockets. La implementación de esta aplica a las operaciones `read()` y `fread()` sobre sockets.
- FaultWriteSocket: clase que representa un defecto sobre la escritura de sockets. La implementación de esta aplica a las operaciones `write()` y `fwrite()` sobre sockets.

En la Figura 4.8 se muestra la jerarquía de defectos implementada para el prototipo.

Estos defectos pueden ser configurados para generen una respuesta de error o que indique una acción que va a ser aplicada dentro del stub.

En la implementación de los defectos que aplican a más de una función, el primer problema a solucionar es que los stubs deben generar el mensaje con

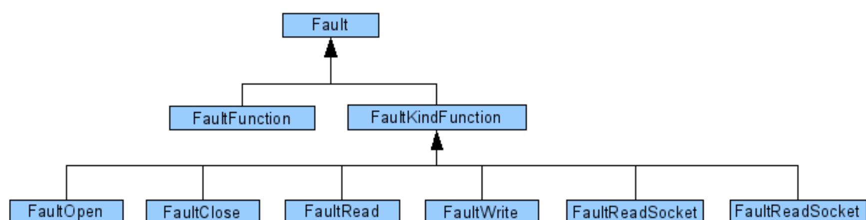

Figura 4.8: Jerarquía de defectos

la información requerida por el defecto, y la respuesta generada por el defecto debe poder ser interpretada por los stubs de todas las funciones que el defecto abarque. Esto no es un problema para operaciones como `close()` y `fclose()` por su parecido, pero se vuelve una tarea compleja si quisiéramos por ejemplo agregar a las faltas interceptadas por FaultRead la función `fscanf()`, dado que tiene una lógica muy diferente a la que presenta `read()` y `fread()`.

Por lo anterior, para poder abarcar mayor cantidad de funciones con un mismo defecto se deben sortear las dificultades que significan las diferencias entre parámetros, retorno y accionar entre funciones. Algunas medidas que se pueden tomar para ello son:

- agregar lógica extra al stub, primero para crear el mensaje genérico que el defecto pueda interpretar y luego para traducir la respuesta genérica a una inyección aplicable a la función.
- diferenciar la acción a realizar según la función interceptada dentro del defecto, agregando más lógica al defecto.
- limitar las acciones posibles del defecto a aquellas que son comunes a todas las funciones a las que aplica.

Al intentar implementar defectos más genéricos también nos encontramos con el inconveniente de que muchas funciones provistas por Libc realizan varias operaciones diferentes. Por ejemplo la función `freopen()` cierra el stream pasado por parámetro, abre un fichero y asigna el stream del fichero abierto al pasado por parámetro; en esta operación se realizan operaciones de apertura y cierre de ficheros.

Cuando operamos con ficheros o conexiones de red nos encontramos con otro problema, este es el de identificar si el elemento sobre el que se está operando representa un archivo o un socket. Para ello se implementaron operaciones que buscan en el sistema de archivos virtuales /proc del proceso interceptado para identificar si se trata de un archivo o socket. En el caso de archivos además se obtiene la ruta de este, esto posibilita la definición de defectos que se aplican a operaciones que afecten archivos específicos. En el caso de sockets se obtiene la información del mismo referente a hosts y puertos de entrada y salida, esto permite definir inyecciones sobre direcciones especificas.

Evaluación del prototipo LD_PRELOAD

Posibles mejoras para este prototipo son:

- Incluir un módulo de monitoreo y registro de eventos.
- Carga de la campaña de inyección por medio de archivo de configuración.
- Modificación de la campaña de defectos en tiempo de ejecución.
- Seguimiento de los hilos: En este prototipo si el proceso objetivo de inyección crea nuevos hilos de ejecución estos últimos también serán inyectados con los defectos definidos para el proceso original. Se podrían brindar más opciones sobre inyección sobre hilos mediante la implementación de alguna estructura que lleve el control de los hilos y más opciones sobre la campaña de inyección o especificación de defectos.
- Redefinición de la jerarquía de defectos que permita una mayor reutilización de código. Se puede observar por ejemplo que las faltas de escritura sobre sockets y sobre filesystem son análogas pero con la jerarquía definida requieren de la repetición de mucho código.

Con respecto a FIG, en el prototipo implementado se crearon defectos con mayor potencial de inyección, agregando la posibilidad de cambios de parámetros de entrada y salida o acciones previas o posteriores a la llamada a la función interceptada.

El enfoque utilizado, que separa el software de inyección de la librería donde se definen los stubs, tiene por consecuencia que las estructuras necesarias para mantener la campaña de inyección ya no se encuentran alojadas en el espacio de memoria del proceso a inyectar, siendo en este aspecto menos intrusivo. Por contrapartida la comunicación de los procesos mediante memoria compartida introduce el uso de semáforos lo cual tiene por consecuencia que la atención de las funciones se hace de forma secuencial, imponiendo un orden de ejecución que en un escenario multi-hilo no se corresponde con la ejecución natural de los procesos, siendo en este aspecto mucho más intrusivo que el enfoque de FIG.

Al evaluar la interposición a Libc para inyección de defectos encontramos las siguientes ventajas:

- El código se genera en un nivel alto lo que lo hace menos complejo y más amigable de programar.
- Todos los nuevos procesos abiertos por un proceso que está bajo inyección son también inyectados sin ningún trabajo extra.
- Al implementar stubs de las funciones de Libc, el producto desarrollado es fácilmente portable y aplicable a cualquier sistema que esté implementado con el uso de esta librería.

Como desventajas podemos mencionar:

- No es posible interceptar `syscalls` hechas de forma directa, por medio de bibliotecas que no utilicen Libc, o resueltas de forma estática.
- Debe generarse un stub para cada función a la que se necesite interponerse.
- Si además se desea ofrecer la posibilidad de definir defectos de mayor complejidad, como los que afecten los parámetros de entrada o salida, es necesario agregar lógica extra a cada stub que depende de la especificación de la función que implementa.

- La implementación de defectos más genéricos, que ataquen un determinado servicio ofrecido por el sistema operativo, es una tarea extremadamente compleja. Al observar el conjunto de funciones ofrecidas por Libc encontramos gran cantidad con finalidades similares pero lógica y parámetros diferentes, y otras funciones que utilizan más de un servicio del sistema operativo.

4.3.3. Prototipo Kprobes

Con el objetivo de lograr la interposición a llamadas al sistema mediante alguna de las variantes de KProbes antes mencionadas, se construyó un prototipo que intercepta la llamada al sistema write. Esta llamada es implementada en el kernel de GNU/Linux por la rutina `sys_write()`.

De las alternativas ya vistas para KProbes se eligió KRetProbes por ser la más adecuada para este tipo de tarea, ya que para lograr realizar inyecciones de defectos se requiere interponerse en el momento inmediatamente antes que comience la llamada y en el momento inmediatamente antes que la misma retorne. KRetProbes nos proporciona este tipo de control mediante los punteros a funciones llamados `‘‘entry_handler()’’` y `‘‘handler()’’` de la estructura struct kretprobe.

Como la función `‘‘entry_handler()’’` para la KRetProbes asociada es invocada en el momento inmediatamente antes a que comience la ejecución de `sys_write`, y dado que en GNU/Linux sobre la arquitectura x86 el pasaje de parámetros es por registro, se tiene que entre los registros salvados en la task correspondiente al momento de conmutar el procesador se encuentran los parámetros pasados por el proceso invocador. De esta forma, obteniendo un puntero a la struct task adecuada podemos acceder y manipular los parámetros de la llamada al sistema en cuestión.

Hay dos formas directas de obtener la struct task, una de ellas es a través de la variable global current, que apunta a la struct task actual, la segunda es a través de uno de los parámetros de “`entry_handler`” más precisamente struct `kretprobe_instance` la cual entre sus campos se tiene acceso a la estructura deseada.

Luego de esto es posible leer y modificar la dirección de espacio de usuario que se encuentra alojada en el registro CX, que para la llamada al sistema write representa la dirección donde se encuentra el buffer con los datos a ser escritos. Luego de realizar los cambios deseados se deja continuar la llamada normalmente.

En el momento inmediatamente antes de retornar el mecanismo de KRetProbes invoca a la función `‘‘handler()’’` asociada, en este punto y accediendo de la misma forma a la struct task se procede a restaurar el buffer de manera tal que al retornar sea transparente para el proceso invocador.

La tarea realizada por el prototipo consiste en que ante llamadas a `write` sobre un determinado file descriptor los datos proporcionados en el buffer dado como parámetro son modificados. Con el método descripto la tarea es realizada de manera tal que es transparente para el proceso que invoca la rutina.

4.4. Conclusión sobre inyección de defectos sobre `syscalls`

De las opciones investigadas para la inyección de defectos sobre `syscalls`, tenemos que Ptrace y LD_PRELOAD se presentan como mejores opciones para construir en base a ellas una herramienta de inyección de defectos que Kprobes o realizar modificaciones al Kernel.

La opción que respecta a las modificaciones del kernel se descarta debido a que tiene una muy fuerte dependencia de la arquitectura y de la implementación del kernel.

En el caso de Kprobes lo que la hace una opción poco atractiva es que las pruebas realizadas basan su éxito en un comportamiento que no está especificado por el API de Kprobes. Esta API tiene como objetivo la monitorización y testeo no intrusivo, y no se dan herramientas para el cambio de variables de los procesos o de las llamadas a funciones. La solución propuesta depende de la forma en que se salva el contexto al invocar los métodos definidos con Kprobes y al acceso a la información de variables globales de contexto. Si la implementación del API cambia en una versión futura la solución propuesta se vuelve obsoleta.

En el caso de Ptrace el gran control sobre los procesos, la facilidad de uso y que esté disponible en todos los sistemas UNIX la hacen una opción a tener en cuenta. Aunque Ptrace tiene algunas desventajas es una excelente candidata para construir una herramienta portable, extensible, fácil de programar y que brinde control total sobre las llamadas al sistema de un proceso.

Por su parte el uso de LD_PRELOAD para el desarrollo de una herramienta de inyección tiene por ventajas una gran portabilidad, debido a que su aplicación se da a nivel del API de Libc y el código a generar se hace en un nivel alto lo que lo hace menos complejo y más amigable de programar. Pero LD_PRELOAD también tiene algunas falencias como la complejidad de atacar de forma genérica a los servicios ofrecidos por el sistema operativo y que solo puede aplicarse sobre procesos que usen compilación dinámica.

De esta manera es que de las opciones LD_PRELOAD y Ptrace ninguna se presenta como ampliamente superior a la otra. Debido a esto y al conocimiento ganado a través de la construcción de los prototipos es que no se descarta ninguna de las dos opciones; en vez de eso se plantea la construcción de una herramienta que integre ambas tecnologías, como dos modos alternativos de inyección sobre llamadas al sistema.

Capítulo 5

Inyección de defectos sobre USB

Uno de los objetivos de este proyecto es investigar y aplicar técnicas de inyección de defectos sobre la tecnología USB. A continuación se explican conceptos básicos de dicha tecnología para más adelante discutir las distintas opciones de inyección investigadas.

5.1. Introducción a USB

Para muchos dispositivos el Universal Serial Bus (USB) se ha transformado en el método estándar de conexión. Se utiliza para conectar un host a una serie de dispositivos periféricos. Algunas de las características que han influenciado el creciente uso de la tecnología USB son:

- El conector USB es único. Una interfaz para muchos dispositivos.
- Posibilidad de conectar múltiples dispositivos sobre el mismo bus.
- Conexión de dispositivos en caliente (hot pluggable).
- Configuración automática. Al conectar el dispositivo el sistema operativo carga los drivers adecuados para el mismo. Si dicho dispositivo cumple con el estándar USB no es necesario siquiera instalar los drivers la primera vez que el dispositivo es conectado.
- Posibilidad de reservar ancho de banda para transmisión de datos multimedia.
- En la mayoría de los casos no es necesario una fuente de alimentación suplementaria para los dispositivos.

El estándar USB se encuentra en la versión 3.0 liberada a fines del 2008 aunque no fue hasta fines del 2009 que comenzaron a aparecer placas que adoptan la nueva revisión del estándar [21][20]. La versión 1.0 y 1.1 soportan las velocidades de transferencia de 1.5 Mbps (low speed) y 12 Mbps (full speed), en la versión 2.0 se añade soporte para 480 Mbps (high speed) y es compatible

con 1.0 y 1.1. En su versión 3.0 se añade el soporte para 5Gbps (super speed) conservando compatibilidad con la versión 2.0. La presente investigación se basa en el uso del estándar 2.0 [19].

USB utiliza una estructura de árbol, con el host como la raíz, hubs como nodos interiores y dispositivos como las hojas. El bus es controlado por el Host y no pueden existir dos host, por lo que es una jerarquía maestro/esclavo [34]. En la Figura 5.1 se ejemplifica esta estructura. Los dispositivos que proveen una

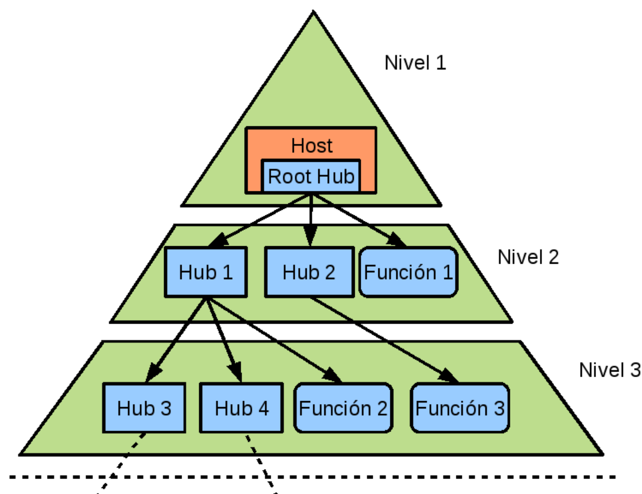

Figura 5.1: Topología USB

funcionalidad son conocidos como funciones. Existen también dispositivos compuestos, estos encapsulan varias funciones y cada una de ellas están conectadas de forma permanente a un hub interno.

Los hubs son conectores que permiten agregar dispositivos al bus. Contienen hasta 8 puertos de los cuales uno conecta con el hub del nivel superior. Solo se permiten 7 niveles y hasta un máximo de 127 dispositivos.

El primer dispositivo de un bus USB es el root hub. Este es el controlador de USB, usualmente contenido en un dispositivo PCI, controla el bus USB conectado a él. No hay un límite de root hubs que puede contener un sistema. La Figura 5.2 ilustra un dispositivo hub.

El sistema operativo en el host es el encargado de administrar los distintos dispositivos, cargar los drivers de los mismos y ofrecer los servicios necesarios para que un driver se comunique con un dispositivo. Los drivers son las piezas de software que se encargan de manejar la lógica propia de los dispositivos para permitir su funcionamiento. En el caso concreto de GNU/Linux el encargado de proporcionar estos servicios es el sub-sistema USB.

En el contexto de este proyecto es de interés encontrar soluciones tecnológicas que permitan realizar inyección de defectos sobre las conexiones USB y los datos transmitidos, para generar una herramienta que permita evaluar el correcto funcionamiento de los dispositivos USB y sus drivers.

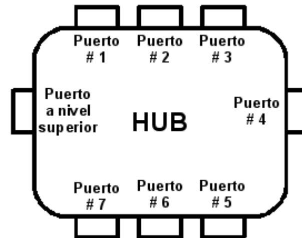

Figura 5.2: Hub

5.1.1. Protocolo USB

El protocolo USB define distintos elementos y procedimientos que hacen posible la comunicación entre los dispositivos y el host.

Un dispositivo USB se compone de un conjunto de endpoints, estos son la interfaz entre el hardware y el firmware del dispositivo. Son los puntos de entrada y salida de datos. Los endpoints se identifican por su dirección, que es un entero entre 0 y 15. También tiene un sentido; los de entrada (IN) proveen datos para ser enviados al host y los de salida (OUT) reciben datos enviados desde el host. Los endpoints son unidireccionales por lo que solo envían o reciben datos, a excepción de los endpoints de control que son bidireccionales.

Existen cuatro tipos de endpoint, uno por cada tipo de transferencia posible [18]:

- Control: usados para configurar, obtener información del dispositivo, envío de comandos y consulta de estado. Cada dispositivo tiene un endpoint 0 utilizado por el host para configurar el dispositivo.
- Bulk: transfieren gran cantidad de datos. Usados en transferencias con el requerimiento de no tener perdida de datos (red, impresora, almacenamiento). No se garantiza la tasa de transacción. Si no hay espacio suficiente en el bus para enviar un paquete bulk se divide en subpaquetes.
- Interrupt: transfieren una pequeña cantidad de datos a tasa fija cada vez que el Host pide datos. Es el método principal de trasferencia de mouses y teclados.
- Isochronous: transfieren grandes cantidades de datos a una tasa fija. No garantizan la no perdida de datos pero mantienen un flujo constante de datos (usados para audio y vídeo).

Los endpoints se agrupan en interfaces. El cometido de una interfaz es manejar un tipo de conexión lógica. Algunos dispositivos USB tienen múltiples interfaces. Por ejemplo una cámara puede tener una interfaz de audio y una de vídeo. Como una interfaz USB representa una funcionalidad básica, cada driver USB controla una interfaz por lo que algunos dispositivos USB pueden necesitar de múltiples drivers.

Cada interfaz USB puede tener una configuración alternativa. El estado inicial de una interfaz es la primera configuración, numerada como 0. Las configuraciones alternativas pueden ser usadas para controlar endpoints individuales de distintas maneras, por ejemplo para reservar diferentes cantidades de ancho de banda.

La Figura 5.3 ejemplifica como se distribuyen endpoint, interfaces y configuraciones de una cámara web USB.

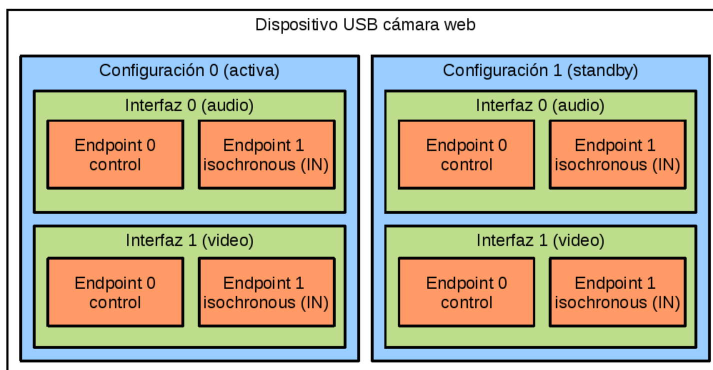

Figura 5.3: Ejemplo de dispositivo USB

Los datos intercambiados entre el sistema y los dispositivos se hace en unidades llamadas URB (USB Request Block). Estos contienen información del dispositivo, el endpoint desde o hacia el cual se envía, información referente a la configuración del endpoint y los datos enviados. Cada endpoint en un dispositivo puede manejar una cola de URBs.

En el kernel de GNU/Linux se definen estructuras que representan cada uno de estos elementos, las mismas son urb, usb_device_descriptor, usb_config_descriptor, usb_interface_descriptor y usb_endpoint_descriptor. Estas estructuras se explican en el apéndice Estructuras USB.

El ciclo de vida típico de un URB en GNU/Linux es el siguiente:

- Creado por un driver de dispositivo USB.
- Asignado a un endpoint de una interfaz de un dispositivo USB específico.
- Enviado al host, por el driver de dispositivo USB.
- Enviado al USB controller driver del dispositivo especificado.
- Procesado por el USB controller driver que hace una transferencia al dispositivo USB.
- Cuando se completa la URB el USB controller driver notifica al driver de dispositivo USB.

Otro proceso de importancia para entender el funcionamiento del USB es la enumeración. Este comienza al conectar un dispositivo al bus USB y termina cuando el dispositivo está correctamente configurado y listo para ser utilizado. Los principales pasos que suceden en la enumeración son los siguientes [34]:

1. Se agrega un dispositivo al sistema.
2. El hub detecta el dispositivo, provee energía al puerto e informa al host. El estado del dispositivo pasa a POWERED y el puerto al que está conectado se encuentra deshabilitado.
3. El host determina la naturaleza del cambio solicitando al hub entre otras cosas un Get Port Status. Habilita el puerto y envía una solicitud de reset.
4. El hub detecta si el dispositivo es low o full speed.
5. El hub reinicia el dispositivo. Mientras esto sucede el dispositivo esta en estado RESET, al culminar el estado del dispositivo es DEFAULT.
6. Si el dispositivo es full-speed el host aprende si soporta high-speed.
7. El hub establece un camino de señal entre el dispositivo y el bus.
8. El host envía un pedido de get descriptor para aprender el tamaño máximo de paquete del pipe por defecto.
9. El host asigna una dirección al dispositivo y este pasa al estado ADDRESS. Desde este punto la comunicación se hace a esa dirección.
10. El host obtiene la descripción y configuraciones del dispositivo haciendo una petición Get Descriptor.
11. El host busca y carga el driver adecuado para el dispositivo. El dispositivo pasa al estado CONFIGURED.
12. El driver selecciona una configuración.

5.1.2. USB y el Kernel de GNU/Linux

Como se mencionó antes en el Kernel el sub-sistema USB es el encargado de encapsular los servicios ofrecidos a los drivers de dispositivos USB. Este se implementa como un módulo del kernel denominado **USB Core** [32]. Su objetivo es abstraer el control del hardware y dispositivos para evitar dependencias a implementaciones específicas mediante la definición de estructuras, macros y funciones.

Presenta dos interfaces, una superior para los drivers de los dispositivos, y una inferior con el USB Host Controller Driver (HCD). La interfaz entre el USB Core y el controlador de host depende de la definición de hardware del controlador de host. Actualmente el controlador de host USB está integrado en los chipsets de las placas base. Las que no cuentan con el controlador pueden ser actualizadas agregándoles una tarjeta PCI con controlador de host. Las interfaces más comunes provistas por los HCD son compatibles con el estándar Open Host Controller Interface (OHCI de Compaq, Microsoft y National Semiconductor) o el estándar Universal Host Controller Interface (UHCI de Intel). También existe el estándar Extended Host Controller Interface (EHCI) para dispositivos high-speed.

En la Figura 5.4 se presenta al USB Core entre las distintas capas del kernel que intervienen en el control de los dispositivos USB.

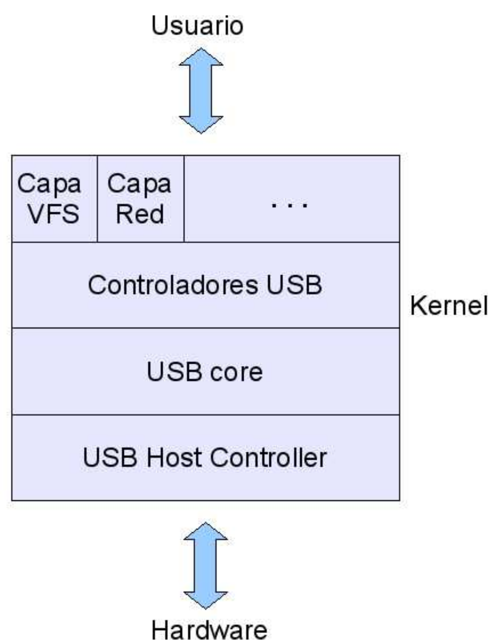

Figura 5.4: USB Core

5.2. Opciones de inyección sobre USB

Examinando el protocolo USB y los distintos procesos antes descriptos se pueden señalar distintos tipos de defectos de interés para las inyecciones.

- Cancelación de URBs.
- Modificación de configuración de interfaces y endpoints en la enumeración. Esto incluye modificación de parámetros como el tamaño máximo de paquete y tipo de los endpoints.
- Alteraciones en los URBs. Incluye la modificación de datos transferidos dentro de los URBs.

A continuación se detallan varias opciones que fueron investigadas en la búsqueda de una solución que permitiese la inyección de los defectos antes mencionados. Entre ellas modificar el USB Core, el uso de librería de alto nivel libusb, estudio de USBMon y su funcionamiento e investigación de la aplicación USB/IP.

5.2.1. Modificación del USB Core

Esta opción consiste en la modificación del USB Core para incluir en él código que permita la comunicación con una herramienta de inyección. Para

ello es necesario investigar su funcionamiento. El objetivo de la investigación sobre el funcionamiento del USB Core es detectar los puntos ideales para la inyección de defectos. Esto refiere a donde se reciben o envían los URBs para poder interceptarlos y cambiar su contenido y a la detección del proceso de enumeración para cambiar allí la configuración de los dispositivos.

A lo largo de todo el proceso de investigación se utiliza openSUSE 11.1 siendo 2.6.27.7-9-pae la versión del kernel.

El API del USB Core proporciona varias operaciones para la creación, manipulación y envío de URBs. Las más importantes son:

- creación y destrucción de urbs: usb_alloc_urb, usb_free_urb
- inicialización de urbs: usb_fill_int_urb, usb_fill_bulk_urb, usb_fill_control_urb
- envío de urbs: usb_submit_urb
- cancelación de urbs: usb_unlink_urb, usb_kill_urb

De las operaciones anteriores la de mayor interés para la inyección es la de envío dado que en este punto sería posible cambiar los valores del URB a enviar. Esta es implementada en el archivo urb.c, dentro del código de esta operación se realizan varios chequeos que terminan con la llamada a la operación usb_hcd_submit_urb. Esta última es implementada en el archivo hcd.c y en ella previamente a encolar el URB para su procesamiento y envío al destino se envía el mismo a través de la operación USBMon_urb_submit a un módulo que permite monitorizar el tráfico USB (el USBMon) que se explicará más adelante.

Al observar todo el camino que los URBs transitan para ser enviados, puede observarse la complejidad de los chequeos, estructuras y funciones que son utilizadas en las operaciones antes descriptas. Además vale la pena mencionar que la documentación disponible a este nivel de "profundidad" en el kernel es sumamente escasa si se la compara con la documentación disponible del API que este ofrece para el desarrollo de drivers. Otro aspecto a tener en cuenta es la variabilidad del código del USB Core entre versiones del kernel, en contraste con las pocas variaciones que presenta su API. En la versión 2.6.27.7 del kernel el USB Core tiene 7534 lineas de código, en la versión 2.6.31.5 8116. Comparando ambas versiones en la 2.6.31.5 hay 688 nuevas lineas, 255 eliminadas y 277 modificadas; esto representa más de un 15 % de modificaciones respecto al código de la versión 2.6.27.7 [1]. Por estas razones una solución que se base a modificar el USB Core implica una pérdida de portabilidad, dado que para implantar en otro kernel la solución se debe estudiar las diferencias entre la implementación de las distintas versiones.

Con respecto al proceso de enumeración el mismo esta a cargo de un hilo del kernel llamado khubd e implementado en la operación hub_thread en el archivo hub.c. A continuación describimos los pasos que realiza en el proceso de enumeración para un pendrive[33].

1. El root hub reporta un cambio en los puertos dado por la inserción de un dispositivo. El hub driver detecta el cambio de status, llamado USB_PORT_STAT_C_CONNECTION y despierta el khubd.

[1] Datos obtenidos mediante la utilización de la herramienta UCC v.2009.10 comparando el kernel presente en las versiones 11.1 y 11.2 de openSUSE

2. Khubd descifra la identidad de puerto USB al que corresponde el cambio de estatus.

3. khubd elige una dirección para el dispositivo entre 1 y 127 y asigna esta al endpoint bulk del pendrive usando un URB de control adjuntado al endpoint 0.

4. Khubd usa el mencionado URB de control para obtener el descriptor del dispositivo del pendrive. Después solicita las configuraciones del dispositivo y selecciona una.

5. Khubd solicita al USB Core un driver para el dispositivo.

La única documentación encontrada sobre este hilo solo hacía referencia a los pasos que este realiza pero no a detalles de su implementación como para hacer viable su modificación. Por las razones antes mencionadas se decidió continuar investigando como realizar la inyección de defectos mediante la utilización del mecanismo provisto para el USBMon.

5.2.2. USBMon

El USB Core provee soporte para la monitorización del tráfico USB, esto se logra mediante la notificación de los URBs que se transmiten a través de las operaciones en la estructura **struct usb_mon_operations** previamente registrada. Esta facilidad está presente en el kernel desde su versión 2.6.11 y fue incluida para soportar la herramienta USBMon. El módulo USBMon aprovecha esta facilidad registrando dicha estructura con el USB Core e implementando las operaciones que esta provee [31].

```
struct usb_mon_operations {
    void (*urb_submit)(struct usb_bus *bus, struct URB *urb);
    void (*urb_submit_error)(struct usb_bus *bus, struct URB *urb,
          int err);
    void (*urb_complete)(struct usb_bus *bus, struct URB *urb);
    void (*bus_add)(struct usb_bus *bus);
    void (*bus_remove)(struct usb_bus *bus);
};
```

- **urb_submit:** A través de esta operación se puede capturar todo el tráfico de URBs que está siendo transmitido en el sistema. Los parámetros indican el URB y el bus al cual pertenece el mismo.

- **urb_submit_error:** En caso de error se indica a través de esta operación el error que ha ocurrido. Los parámetros indican el URB y el bus al cual pertenece así como el status del mismo.

- **urb_complete:** Una vez procesado el URB, se invoca esta operación indicando que el procesamiento del mismo ha culminado. Los parámetros indican el URB y el bus al cual pertenece así como el status del mismo.

- **bus_add:** No se encuentra implementada en la versión del kernel utilizada.

- **bus_remove:** No se encuentra implementada en la versión del kernel utilizada.

Dado que el USBMon es un módulo del kernel que es registrado para ser notificado de los envíos y recepciones de URBs, abre la posibilidad de realizar en dicho punto la captura del tráfico de URBs sin modificar el USB Core.

Tomando la idea del USBMon es que se construye un módulo que captura el tráfico de URBs y de esta forma realizar pruebas de inyección de defectos sobre los mismos. Las pruebas realizadas son las siguientes:

- filtrado de URBs: los URBs pueden ser cancelados en cualquier momento por el driver del dispositivo o por el USB Core. La prueba consiste en verificar si dicha cancelación puede ser realizada desde el módulo implementado. Esto permitiría simular la perdida de URBs en la comunicación.

- modificación de datos transmitidos: en esta categoría se realiza dos tipos de pruebas; modificar los datos referentes a la posición del cursor que son enviados por un mouse USB y cambiar los datos enviados desde y hacia un dispositivo de almacenamiento USB.

- asignación del callback de los URBs: los URBs contienen un puntero a una función que es llamada al completar el procesamiento de los mismos. Esta es utilizada por los drivers generalmente para verificar el estado de los envíos. En esta prueba se intenta cambiar los callbacks por una operación propia, de esta forma en ésta última se intenta cambiar el estado de los URBs y posteriormente llamar al callback original.

- cambio de estado en el complete: la operación urb_complete de la estructura registrada al igual que los callback es llamada al terminar el proceso del URB. Las pruebas realizadas usando esta función apuntan al cambio de estado de los URBs.

- modificación de datos en la enumeración: capturar los URBs correspondientes al proceso de enumeración para un dispositivo y cambiar la información referente a la configuración del mismo.

El detalle de las pruebas realizadas se encuentra en el apéndice Pruebas de prototipo USB.

Evaluación de las pruebas con módulo del kernel

Resumen de las pruebas realizadas para cada caso:

- filtrado de URBs: Al realizar unlink de URBs no hubo efecto ninguno en las transferencias ni en el comportamiento del sistema. En las pruebas que se realizó el kill de los URBs se paralizaron todos los dispositivos conectados en un mismo bus.

- modificación de datos transmitidos: Fue posible modificar los datos transmitidos en transferencias INTERRUPT, y en valores propios de los URBs como son parámetros de la transferencia y hasta el propio endpoint al que se envía la información. Se constataron diferentes efectos sobre el funcionamiento del sistema.

- asignación de callbacks: estas pruebas se pudieron realizar con éxito, asignando distintos valores al status del URB en los callbacks. Esto permitió apreciar el distinto tratamiento de error hecho por los drivers de los dispositivos que se usaron en las pruebas.

- cambio de estado en el complete: luego de inspeccionar el código del USB Core se constató que luego de llamada la operación complete del módulo el valor del status de los URBs era restaurado por lo que estas pruebas no tienen ningún efecto.
- modificación de datos en la enumeración: En las distintas pruebas fue posible cambiar la información obtenida por el USB Core en la enumeración y apreciar diferentes efectos en el comportamiento del sistema.

Como mayor ventaja de este enfoque para realizar las inyecciones de defectos podemos indicar que el mismo permite realizar las pruebas que fueron planteadas. Esto es posible debido a que el mecanismo proporcionado para el USBMon permite capturar los URBs en puntos claves de la transmisión de los mismos, permitiendo su modificación antes de que estos lleguen a ser procesados por el USB Core en la función submit, y antes de la llegada al driver del dispositivo en la función complete.

Otra ventaja es la portabilidad de la solución. Esta viene dada porque utiliza el mecanismo proporcionado por el kernel para la monitorización desde su versión 2.6.11 (año 2005) y se basa en la modificación de estructuras definidas en el estándar USB como son las estructuras URB y descriptores de dispositivo y endpoint.

Como desventaja de se puede mencionar la necesidad de tener un profundo conocimiento del estándar USB, sus estructuras y hasta de los propios dispositivos a testear antes de decidir que pruebas realizar, y para la posterior interpretación de los resultados obtenidos.

5.2.3. Libusb

En la búsqueda de una solución a la inyección de defectos sobre USB que permitiese realizar las mismas desde espacio de usuario es que nos encontramos con libusb. Esta es una biblioteca open source que proporciona funcionalidades para acceder a dispositivos USB desde el espacio de usuario [35]. La versión estable es la 1.0. Entre sus características se encuentra:

- Soporta todos los tipos de transferencias.
- Dos interfaces de transferencias; sincrónicas y asincrónicas.
- Seguridad en hilos (Thread safe)
- Una API reducida y liviana.
- Soporte para USB 1.1 y 2.0.
- Provee estructuras abstractas para los tipos de datos de forma de mantener la portabilidad.

La librería maneja varias estructuras internas que deben ser inicializadas, para ello se debe seguir los siguientes pasos:

1. inicializar libusb mediante la función usb_init
2. buscar los buses en el sistema mediante la función usb_find_busses

3. buscar los dispositivos de cada bus mediante la función usb_find_devices

En este punto se pueden obtener los buses del sistema mediante usb_get_buses. Cada bus tiene una lista con los dispositivos para acceder a la información de los mismos. Para acceder a un dispositivo el mismo debe ser abierto mediante la función usb_open. Si el dispositivo ya tiene asignado un driver debe desasociarse del mismo mediante la función usb_detach_kernel_driver_np.

Al estudiar el API de esta librería nos encontramos con que no presenta ninguna función que permita la manipulación directa de los URBs, la modificación de la información de los dispositivos, ni al intercepción del flujo de datos desde o hacia un driver.

5.2.4. USB/IP

USB/IP es un proyecto desarrollado por Takahiro Hirofuchi, Eiji Kawai, Kazutoshi Fujikawa y Hideki Sunahara en el Nara Institute of Science and Technology. Se trata de una aplicación que permite compartir a través de una red dispositivos USB conectados en un equipo para que los pueda gestionar otro [36].

Dado el objetivo antes planteado de inyectar defectos sobre el tráfico USB, este proyecto abre la posibilidad de poder realizar dichas inyecciones interceptando en algún punto el tráfico sobre la red y realizar allí la inyección. Para poder realizar dicha tarea es necesario investigar el funcionamiento de esta herramienta.

La arquitectura de esta aplicación es distribuida entre un servidor al cual se conectan los dispositivos USB a ser compartidos y los clientes que hacen uso de dichos dispositivos. Esta arquitectura se presenta en la Figura 5.5.

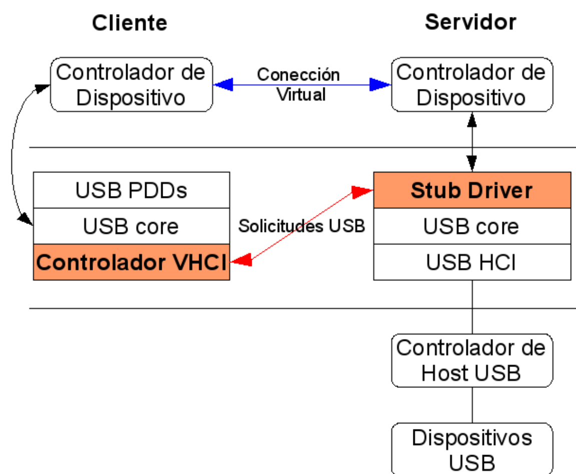

Figura 5.5: Esquema USB/IP

En el cliente se implementa el manejador VHCI (Virtual Host Controller Interface) que emula al controlador USB del host. Esto implica emular la enu-

meración e inicialización de forma remota de dispositivos y encapsular los URBs enviados hacia el dispositivo conectado en el servidor y desencapsular los que provienen del mismo. El servidor es donde está conectado efectivamente el dispositivo y allí se implementa el Stub driver. Este se encarga de administrar los dispositivos compartidos y de encapsular los URBs con destino al cliente y desencapsular los que llegan hacia los dispositivos compartidos. En lo que refiere a la investigación de esta herramienta surgen dos objetivos:

1. Instalar y comprobar el correcto funcionamiento de esta aplicación.
2. Investigar la posibilidad de utilizar la aplicación con la finalidad de realizar inyecciones de defectos o datos en el tráfico USB.

Instalar y comprobar el correcto funcionamiento de USB/IP

Los elementos necesarios para correr la aplicación se dividen en dos partes, los módulos que implementan el VHCI driver y el Stub Driver y los binarios que implementan los comandos por los cuales se comparten los dispositivos. En las versiones de openSUSE 11.1 y 11.2 los módulos ya se encuentran compilados y disponibles para ser usados. Los binarios deben ser instalados aparte, vienen dentro del paquete usbip y la versión que se instala desde el yast es la 0.1.6-4.1.

Es importante considerar que dado que es indispensable tener la facultad de modificar la aplicación es necesario correr la misma desde los módulos y binarios compilados a partir del código fuente. Los fuentes del proyecto USB/IP puede ser descargados desde la página del proyecto USB/IP estos corresponden a la versión 0.1.7 [37].

Los pasos necesarios para poner en marcha la aplicación se resumen así:

- En el servidor:
  - conectar el dispositivo USB.
  - levantar los módulos que dan soporte al Stub driver.
  - levantar el demonio usbip.
  - con los comandos de la aplicación listar los dispositivos disponibles para compartir y posteriormente indicar el dispositivo a compartir. Esto hace que el dispositivo ya no esté disponible en el servidor. utilizar los comandos para dejar de compartir el dispositivo cuando esto se desee. Esto retorna el control del dispositivo al servidor, lo que es posible si en el cliente el dispositivo fue desconectado.
- En el cliente:
  - levantar los módulos del manejador VHCI
  - con los comandos de la aplicación y dada la IP del servidor listar los dispositivos disponibles para ser utilizados y luego seleccionar uno para ser utilizado. Esto conecta de forma remota el dispositivo, realizando la enumeración y dejando el dispositivo disponible para su uso en el cliente.
  - usando los comandos de la aplicación indicar que el dispositivo sea desconectado del cliente cuando así se desee.

Varias pruebas fueron realizadas sobre la compilación, instalación y puesta en marcha de USB/IP. En todas las pruebas realizadas se constataron irregularidades de funcionamiento y fallas. Las pruebas se detallan en el apéndice Pruebas USB/IP.

Evaluación de USB/IP

En el tiempo dedicado a la investigación de USB/IP no fue posible detectar las causas por las que dicha aplicación fallaba luego de seguir todos los pasos indicados para su correcto funcionamiento. Luego de varias pruebas de instalación y puesta en funcionamiento concluimos que la herramienta USB/IP carece de la estabilidad y robustez que es de esperar para construir en base a ella la solución buscada a la inyección de defectos sobre USB. Debido a esto el objetivo antes planteado referente a investigar las posibilidades de utilizar esta herramienta fue abandonado.

5.3. Conclusión sobre inyección de defectos sobre USB

Se encontró una solución al problema planteado de inyectar defectos sobre la tecnología USB. Esto fue posible capturando el tráfico de URBs entre drivers y dispositivos mediante la creación de un módulo que utiliza el mecanismo provisto por el kernel para USBMon. Se logró modificar los URBs transmitidos, y en particular entre ellos los destinados a obtener información sobre la configuración de los dispositivos, interfaces y endpoints en la enumeración.

Se descartaron las demás opciones investigadas:

- Modificar el USB Core debido a su complejidad y poca portabilidad.
- El uso de libusb dado que no ofrece funcionalidades que permitan modificaciones al tráfico de URBs.
- El uso de la aplicación USB/IP, debido a la poca estabilidad que presentó en las pruebas de uso.

Capítulo 6

Producto

Luego de investigadas las distintas opciones de inyección y habiendo realizado prototipos para evaluar las mismas, se eligieron un conjunto de estas para construir una herramienta de inyección. Para las inyecciones de defectos sobre `syscalls` se integraron las tecnologías Ptrace y LD_PRELOAD dentro del producto como dos modos de funcionamiento. Para la inyección de defectos sobre USB se utilizó el mecanismo provisto por el kernel para USBMon.

El ingreso de las campañas de inyección se puede realizar especificando cada defecto desde línea de comandos o incluyéndolos en un archivo que posteriormente es cargado por la aplicación.

A continuación se describe la arquitectura del producto desarrollado y las funcionalidades que provee. En esta sección no se explican detalles del funcionamiento de cada una de las tecnologías dado que este tema fue abordado en secciones anteriores.

6.1. Arquitectura

La aplicación se divide en dos módulos. El primero de ellos se denomina FIK (Fault Injection into the Kernel) y ejecuta en espacio de usuario. FIK es el encargado de la inyección de defectos sobre `syscalls` y de recibir la especificación de los defectos que se desea inyectar. FIK tiene dos modos de ejecución (Ptrace y LD_PRELOAD) entre los cuales debe elegirse al iniciar el mismo. Esto se explicará en detalle más adelante.

El segundo módulo se denomina USBFI (USB Fault Injection) y está implementado como un módulo del kernel. Es el encargado de la inyección de defectos sobre USB.

Si bien ambos módulos realizan inyección de defectos, trabajan de forma independiente. Esto se debe a que la naturaleza de la inyección de defectos sobre Syscalls difiere en mucho de la inyección sobre el protocolo USB. La única interacción entre ambos módulos esta dada por la especificación de los defectos a inyectar. Dicha especificación es ingresada en el módulo FIK que trabaja sobre espacio de usuario, este al recibir comandos referentes a la inyección sobre sobre USB los trasmite al módulo USBFI mediante la escritura en directorios virtuales creados para dicho fin.

Otra pieza importante del producto creado es la librería que contiene los

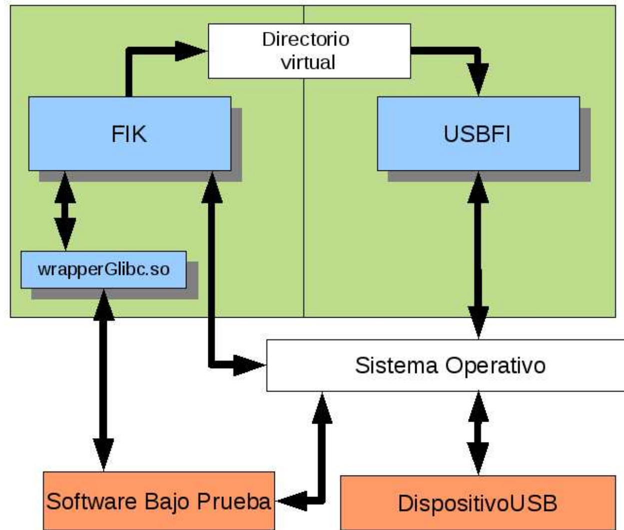

Figura 6.1: Descomposición en módulos del producto

wrappers de funciones de Libc (wrapperGlibc.so). Para realizar inyección de defectos sobre `syscalls` en el modo LD_PRELOAD la ruta a dicha librería debe estar precargada en la variable LD_PRELOAD antes de comenzar a ejecutar la aplicación sobre la que se quiere inyectar.

En la Figura 6.1 se muestra la arquitectura de la aplicación.

A continuación se describen los distintos elementos que constituyen la herramienta construida y su funcionamiento.

6.1.1. Módulo FIK

El módulo FIK es el encargado de recibir la especificación de la campaña de inyección y de realizar la inyección de defectos sobre `syscalls`.

En la Figura 6.2 se muestran los componentes del módulo y como se relacionan. A continuación se describe cada uno de ellos y su funcionamiento.

Componentes

Parser

Este componente es el punto de entrada para la especificación de la campaña de inyección, se encarga de interpretar los distintos comandos aceptados por la aplicación. Está implementado en la clase Parser.

Si el comando corresponde a la especificación de un defecto a ser inyectado sobre `syscalls` se crea la especificación del defecto y el mismo se pasa al componente Controller. Si el comando especificado refiere a la inyección de defectos sobre USB se escribe en el directorio virtual correspondiente.

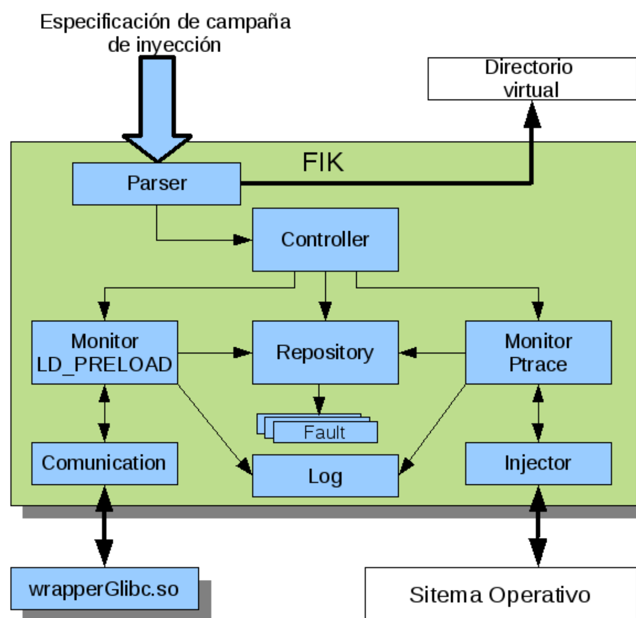

Figura 6.2: Módulo FIK

La lista completa de comandos así como otras especificaciones para correr la aplicación se encuentran en el apéndice Manual del Producto.

Controller

Este componente es el encargado de crear el repositorio, cargar los defectos en él, e inicializar el monitor. Se implementa en la clase Controller.

Como se explico con anterioridad hay dos modos de ejecución, cada uno corresponde a una tecnología de inyección diferente. Por este motivo existen dos clases monitor, cada una contiene la lógica necesaria para la inyección en una u otra tecnología. Dependiendo el modo de ejecución será el tipo de monitor que se instanciará.

Repository

Es el contenedor de los defectos de la campaña de inyección. Brinda primitivas de mutuo exclusión para acceder a los defectos que contiene. Esto permite agregar nuevos defectos al repositorio luego de comenzada la campaña de inyección. Se implementa en la clase Repository.

Monitor LD_PRELOAD

Este monitor es el utilizado cuando el modo de uso corresponde a LD_PRELOAD. Se implementa en la clase MonitorLD.

Se encarga de inicializar la comunicación y quedar a la espera de mensajes provenientes de los wrappers de las funciones de Glibc. Ante un nuevo mensaje

recorre los defectos del repositorio notificando a los mismos de lo ocurrido y retornando la respuesta correspondiente.

Comunication

Componente que encapsula lo referente a la comunicación con los wrappers de Glibc en el modo LD_PRELOAD. Esto se realiza a través de un espacio en memoria compartida que es mutuo excluida por el uso de semáforos. Esta implementado en la clase Comunication.

Monitor Ptrace

Este monitor es el utilizado cuando el modo de uso corresponde a Ptrace. Se implementa en la clase Monitor.

Al ser inicializado se encarga de acoplar con ptrace el proceso sobre el cual se debe inyectar. También se encarga de monitorizar las llamadas al sistema que el proceso realice notificando a los defectos del repositorio para que decidan si deben inyectar. Si es así utiliza los servicios del Injector para realizar la inyección.

Injector

Encapsula los servicios que ofrece ptrace, proporcionando un API más amplia y fácil de utilizar. Se implementa en la clase FIKlib.

Faults

Bajo el concepto de Faults es que se hace referencia a los defectos. Hay dos tipos de estos, los defectos que corresponden al modo Ptrace y los defectos del modo LD_PRELOAD.

Ambos tipos de defectos difieren de gran manera en su implementación debido a la dependencia con la tecnología, pero se basan en la misma idea. Encapsulan la lógica de cuando el defecto que representan debe activarse y generan una respuesta acorde a ello.

En el caso de los defectos implementados para Ptrace, el mismo defecto es el encargado de realizar las modificaciones necesarias para activar el defecto, mientras que para LD_PRELOAD el defecto genera una respuesta que es transmitida al wrapper de la operación interceptada, siendo el wrapper el que ejecuta la acción correspondiente.

La jerarquía de defectos implementada para Ptrace es la misma que se especificó en el prototipo y que se muestra en la imagen 4.6. Los defectos WriteSocketFault y ReadSocketFault fueron modificados respecto a los implementados en el prototipo para que se interpongan a una mayor cantidad de llamadas al sistema para escritura y lectura sobre sockets. En el caso de WriteSocketFault intercepta las llamadas al sistema write, pwrite64, writev, y las socketcall send, sendto y sendmsg. ReadSocketFault intercepta las llamadas al sistema read, pread64, readv, y las socketcall recv, recvfrom y revmsg.

En el caso de los defectos de LD_PRELOAD se presentan varios cambios respecto a lo implementado para el prototipo:

- En primer lugar no se implementan defectos sobre conjuntos de funciones, solo se interceptan funciones concretas. Las funciones para las que se implementaron defectos fueron close, fclose, open, fopen, write, fwrite, read y fread.
- Todos los defectos pasan a tener el sufijo LD para diferenciarlos de los de Ptrace que pudiesen tener nombre similar.
- Se agrega un nuevo tipo de inyección a los ya definidos para el prototipo LD_PRELOAD. Este aplica a las operaciones write, fwrite, read y fread y consiste en remplazar una cadena de caracteres dada por otra del mismo largo. La cadena es reemplazada en lo que se lee si el defecto es activado en una operación de lectura y se remplaza en lo que debe escribirse si el defecto se activa para una operación de lectura.

La jerarquía de defectos implementados para LD_PRELOAD se muestra en la Figura 6.3.

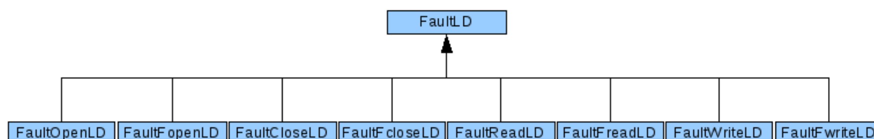

Figura 6.3: Jerarquía de defectos LD_PRELOAD

Log

Componente que encapsula las operaciones de registro de los sucesos ocurridos en la campaña de inyección sobre `syscalls`. Estos refieren principalmente la ocurrencia de inyección.

Es implementado en la clase Log, y simplemente escribe en un archivo los datos que son importantes de registrar para la aplicación.

Ejemplos de log se encuentra en el apéndice Manual del Producto.

6.1.2. WrapperGlibc.so

La librería wrapperGlibc.so contiene el conjunto de wrappers de las distintas funciones de Glibc que se pueden interceptar. Dado que la relación entre tipos de defectos y funciones interceptables es uno a uno la lista de wrappers que fueron implementados corresponden a las funciones close, fclose, open, fopen, write, fwrite, read y fread. Para que sea efectiva la interposición a estas funciones es necesario que donde se ejecuta el proceso objetivo de la inyección la variable de ambiente LD_PRELOAD tenga por valor la ruta a esta librería.

6.1.3. Directorios virtuales

Para la comunicación entre el módulo USBFI y FIK se utilizan directorios virtuales. Mediante debugfs [40] se crean un directorio virtual denominado usbfi bajo /sys/kernel/debug y dos ficheros los cuales se denominan faults y device.

El fichero device contiene las indicaciones necesarias para identificar el dispositivo sobre el cual se realiza la inyección. Los datos necesarios para esto son los ids de vendedor y producto. El fichero faults contiene la especificación de los defectos a inyectar.

Cada vez que el Parser identifica un comando referente a la inyección USB escribe en el fichero correspondiente.

6.1.4. Módulo USBFI

El módulo USBFI es el encargado de la inyección de defectos sobre el protocolo USB. Este está implementado como un módulo del kernel y para poder realizar inyecciones sobre USB este módulo debe ser cargado.

En él se pueden distinguir cuatro componentes principales denominados parser, repository, usbfi y faults. En la Figura 6.4 se muestran los componentes del módulo y como se relacionan.

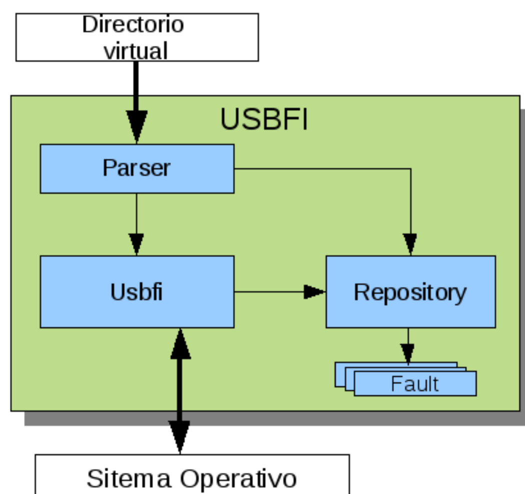

Figura 6.4: Módulo USBFI

Los distintos eventos sucedidos en el transcurso de la ejecución del módulo son registrados en el log del sistema. Un ejemplo de estas salidas se encuentra en el apéndice Manual del Producto.

A continuación se describen cada uno de los componentes del módulo.

Parser

Es el punto de entrada al módulo. Se encarga de crear el directorio virtual usbfi y los ficheros faults y device mediante el uso de debugfs. Luego de crear los mencionados ficheros se encarga de monitorizar cualquier escritura en ellos.

Si el fichero escrito es el device lee del mismo y actualiza la información referente al dispositivo que se debe inyectar, o sea el número de vendedor y producto.

Si el fichero escrito es el faults lee de este interpretando que tipo de defecto es el que se especificó, para posteriormente crearlo y agregarlo al repositorio.

Repository

Este componente es el análogo al Repository de FIK. Es un contenedor de los defectos especificados que provee funciones de mutua exclusión de los elementos que contiene.

Usbfi

Es el análogo al monitor de FIK. Se implementa en el archivo usbfi.c y contiene las funciones que son registradas para monitorizar el tráfico USB. Las funciones que registra utilizadas para la inyección son mon_submit y mon_complete.

- **mon_submit:** ante el envío de un URB hacia un dispositivo chequea primero que el dispositivo al que pertenece el URB sea el mismo sobre el cual se realiza la inyección. Luego se recorre el repositorio notificando a los defectos.
- **mon_complete:** ante el retorno de un URB hacia el host chequea primero que el dispositivo al que pertenece el URB sea el mismo sobre el cual se realiza la inyección. Luego se recorre el repositorio notificando a los defectos.

Faults

Corresponde a los defectos que pueden ser inyectados. Estos defectos se implementan en fault_usb.h y se especifican en la estructura fault_usb. En la misma además de los valores de configuración del defecto se tiene un puntero a la función que verifica si el defecto debe ser activado, y de ser así modifica el URB que recibe según corresponda al cometido del defecto.

Se cuenta con dos tipos de defectos:

- **cambio de valor de estado:** Refiere a cambiar el atributo status de un URB. Esta operación debe realizarse en el Callback del URB dado que en la implementación del protocolo USB el status del URB es restaurado luego de la función complete registrada para el monitoreo. Debido a esto para poder cumplir el cometido de cambiar el status, en la operación complete se hace apuntar al callback del URB a una función que realiza el cambio de status. Como no se debe perder la referencia al callback original este es respaldado dentro del defecto junto con el contexto del URB. El defecto es guardado en el lugar del contexto del URB para que cuando se ejecute el callback se pueda restaurar el contexto en la función de cambio de status y llamar al callback original.

  Hay cuatro parámetros de configuración para este defecto:

  - When: desde que vez se comienza a inyectar.
  - Frecuency: indica la cantidad de URBs que se inyectan de forma consecutiva antes de dejar pasar uno sin inyectar.

    - Allways: indica si la inyección debe realizarse una única vez o si queda activada por siempre.
    - FaultValue: valor de error que será asignado al status. Los posibles valores se explican en el apéndice Estructuras USB.

- **cambio de valor en la enumeración:** Este defecto también se realiza en la función complete de monitoreo. Si el URB corresponde a una respuesta a una solicitud de configuración (utilizados en la etapa de enumeración) se asigna al parámetro especificado el valor especificado. En esta versión del producto los parámetros que pueden ser cambiados son bInterval, bEndpointAddress y bDescriptorType de un endpoint. El EndpointDescriptor a modificar debe especificarse con el número de configuración, interfaz, altSetting y endpoint.

  Hay cuatro parámetros de configuración para este defecto:

    - Configuración, Interface, AltSetting, EndPoint: Identifica el endpoint en el que se requiere inyectar.
    - Atributo a cambiar: indica el atributo de configuración a ser cambiado, los soportados hasta el momento son bEndpointAddress, bInterval y bmAttributes. Los tres son atributos de un EndpointDescriptor.
    - FaultValue: valor que será asignado en el campo especificado.

6.2. Evaluación del producto

La herramienta construida integra los prototipos realizados basados en Ptrace, LD_PRELOAD y el módulo basado en el mecanismo de USBMon, ofreciendo dos modos de inyección de defectos sobre `syscalls` y un módulo de inyección USB. La integración de los prototipos en el producto final implicó relativamente poco esfuerzo y cambios en implementación.

Desde la construcción de los prototipos iniciales se pudo apreciar el potencial de las técnicas investigadas para evaluar la fiabilidad de los sistemas mediante la introducción de defectos y la observación de las respuestas a los mismos.

Las `syscalls` son un punto estratégico para la inyección de defectos por ser la interfaz entre el sistema operativo y los procesos. Las aplicaciones son sumamente sensibles a los errores que se producen al intentar utilizar los servicios del sistema operativo dado que dependen de ello para cumplir sus objetivos. En este punto es que el producto desarrollado se presenta como una herramienta con gran potencial para la evaluación de los manejos de errores realizados por las aplicaciones.

En lo que respecta a la inyección sobre USB, las posibilidades de inyección provistas permiten evaluar el correcto funcionamiento de los Drivers y del propio subsistema USB tanto al momento de la enumeración o ante errores que puedan ocurrir en el envío de los URBs.

Debido a que las inyecciones se realizan en puntos comunes a las aplicaciones y al protocolo USB es que la herramienta desarrollada presenta un gran grado de abstracción del sistema o dispositivo objetivo de la inyección. Esto hace posible inyectar defectos sobre la comunicación con cualquier dispositivo USB y sobre llamadas al sistema o a Libc de cualquier proceso sin necesidad de conocer la implementación de los mismos. La única salvedad a esto es que en modo

LD_PRELOAD no es posible inyectar sobre procesos que son compilados de forma estática.

Una de las características más atractivas del producto desarrollado tiene que ver con la facilidad de extender la aplicación mediante la creación de nuevos defectos. En el caso de los defectos implementados para Ptrace solo se requiere crear una nueva clase de defectos dentro de la jerarquía la cual contiene su propia lógica de activación e inyección. En el caso de los defectos implementados para LD_PRELOAD es análogo al caso Ptrace pero también se requiere crear un wrapper de la función a inyectar.

En el caso de inyección sobre USB hay dos posibilidades de extender las funcionalidades. Una es implementar la modificación de parámetros en los URBs de tipo OUT y la otra es permitir la modificación de cualquier parámetro dentro de la configuración obtenida en la enumeración. Ambas extensiones son fáciles de realizar y no impactan sobre las funcionalidades ya ofrecidas.

Otra característica de la herramienta desarrollada es la portabilidad. Los elementos que son la base de ésta son Ptrace, USBMon,
LD_PRELOAD y la librería Libc. Los cuatro están presentes hace tiempo en todos los sistemas GNU/Linux y con una API estable, esto hace que el producto desarrollado tenga un gran potencial de adaptación a otros entornos, con un bajo o inclusive nulo costo de adaptación[1].

[1] La herramienta fue probada en los sistemas operativos openSUSE 11.1, 11.2 y 11.3 con las versiones del kernel 2.6.27.7, 2.6.31.5 y 2.6.34.7 respectivamente, sin necesidad de realizar adaptaciones

Capítulo 7

Conclusiones y Trabajo Futuro

En este capítulo se presentan las conclusiones del proyecto. Para ello se exponen algunas de las dificultades encontradas en el desarrollo del trabajo, los resultados y aportes obtenidos respecto a los objetivos planteados y posibles trabajos a futuro.

7.1. Conclusiones

Como primer aporte de este trabajo se destaca la información recabada referente a los distintos tipos de defectos y técnicas de inyección de defectos. Estos conocimientos son el punto de partida ineludible de cualquier investigación o desarrollo en torno a la inyección de defectos en sistemas informáticos.

Si bien la inyección de defectos hace tiempo ya que es investigada, no es de las áreas de la tecnología de la información con mayor desarrollo. En especial en lo que refiere al testing de drivers y dispositivos USB no se cuenta con herramientas que permitan realizar inyección de defectos.

Gran parte de la duración del proyecto fue invertida en adquirir conocimientos sobre GNU/Linux, manejo de `syscalls` por el kernel, el protocolo USB y programación en el kernel. Cabe mencionar que si bien abunda la documentación sobre el API ofrecida por el kernel para trabajar con el protocolo USB, no es así en lo que refiere a la documentación del funcionamiento interno del mismo. Una de las dificultades mayores en el trabajo estuvo al momento de investigar como maneja el USB Core el protocolo USB. Este aspecto fue fundamental al buscar una opción de inyección sobre el protocolo USB, pero debido a la escasa bibliografía encontrada insumió mucho tiempo.

En el transcurso del trabajo se logró identificar la importancia de la inyección de defectos como herramienta para la evaluación de los sistemas informáticos. Estas técnicas posibilitan probar el comportamiento del sistema ante escenarios que a menudo no son tenidos en cuenta, algunas veces debido a la baja probabilidad de la ocurrencia de los mismos y otras porque no es posible reproducirlos por otros medios.

Se identificaron las `syscalls` como un punto estratégico para la inyección de defectos y se encontraron varias soluciones a este problema. Los prototipos

realizados para evaluarlas permitieron determinar que las basadas en el uso de Ptrace y LD_PRELOAD son las más adecuadas entre las opciones investigadas.

En lo que refiere a la inyección sobre el protocolo USB, luego de probar con distintas opciones de forma infructuosa se logró encontrar una solución utilizando el mecanismo provisto por el kernel para la monitorización del tráfico de URBs. Las características de esta solución son análogas a la encontrada para `syscalls`, dado que interceptar el tráfico de URBs es el punto ideal para la inyección de defectos. Esto porque permite modificar cualquier dato que se transmita entre el host y cualquier dispositivo USB.

Otro aporte es la herramienta de inyección construida, que implementa un conjunto de funcionalidades básicas que muestra la viabilidad de construir una aplicación más completa con las técnicas elegidas para inyectar defectos sobre software y sobre el protocolo USB.

Esta herramienta cuenta con fuertes características de portabilidad y extensibilidad. La portabilidad viene dada porque tanto Ptrace, USBMon, LD_PRELOAD y la librería Libc son elementos que están presentes hace tiempo en todos los sistemas GNU/Linux y con una API estable. La extensibilidad es una característica del diseño de la aplicación; la modularidad de la misma y la forma en que se diseñaron los defectos permiten agregar nuevas funcionalidades con poco esfuerzo y programación. Estas características dan lugar a tomar la herramienta desarrollada como base para la construcción de un producto más completo para la inyección de defectos.

Para terminar, se concluye que en líneas generales se cumplió con los objetivos planteados. El relevamiento del estado del arte, la investigación de diferentes tecnologías en busca de una solución y la construcción de una herramienta que permite la inyección de defectos sobre procesos y protocolo USB. El presente proyecto se presenta como una buena base para futuros trabajos en esta área.

7.2. Trabajo Futuro

Extender las posibilidades de inyección

Son varias las posibilidades de extender el producto, aquí se presentan algunas de ellas.

- **Aumentar la cantidad de defectos inyectables:** En cada uno de los modos de ejecución pueden agregarse nuevos defectos con facilidad. En el caso de Ptrace para cubrir la mayor cantidad de llamadas al sistema. En el caso de LD_PRELOAD, para cubrir una mayor cantidad de las funciones ofrecidas por Glibc. En el caso de inyección sobre USB existen dos posibilidades de aumentar la oferta de defectos inyectables; una es implementar la modificación de parámetros en los URBs de tipo OUT y la otra es permitir la modificación de cualquier parámetro dentro de la configuración obtenida en la enumeración.

- **Modo de inyección de cambio de cadena para Ptrace:** Esta funcionalidad fue ofrecida para LD_PRELOAD, pero se desistió de hacerlo para PTRACE debido al poco aporte extra que significa para el proyecto en relación con el esfuerzo demandado y teniendo en cuenta que ya se

logro implementar un producto que muestra la viabilidad de las técnicas estudiadas. Por este motivo también cae dentro del trabajo futuro.

- **Defectos activados por tiempo:** En este producto los defectos son activados por conteo. En el caso de `syscalls` la activación de un defecto se da por cantidad de llamadas a la misma `syscall` o función bajo las mismas condiciones. En el caso de USB basado en el conteo cantidad de URBs enviados o recibidos que pertenecen a un dispositivo especificado. Una posible extensión a la herramienta sería agregar la posibilidad de activar los defectos por tiempo. Esto si bien tiene un costo mayor que las anteriores mejoras mencionadas, puede ser logrado mediante la introducción de un Planificador que maneje los defectos activados por tiempo.

- **Inyección sobre sub procesos en Ptrace:** En la inyección sobre `syscalls` mediante Ptrace solo se inyecta sobre el proceso cuyo pid fue especificado, una posibilidad de extensión sería hacer el seguimiento de las llamadas a `fork` registrando los subprocesos abiertos para permitir inyectar en ellos también.

- **Discriminación de sub procesos en LD_PRELOAD:** Al contrario de lo que sucede con Ptrace, para LD_PRELOAD todo subproceso es interceptado dado que comparten las mismas variables de entorno que el proceso padre. En este caso la posibilidad de extensión sería hacer el seguimiento de las llamadas a `fork` registrando los subprocesos abiertos para permitir excluir los procesos sobre los que no se quiere inyectar.

Implementación de un módulo de cache

Uno de los inconvenientes en lo que refiere a la performance que se puede encontrar en la implementación de los defectos elegidos tanto para PTRACE como para LD_PRELOAD, es la necesidad constante de recurrir a la información brindada en los directorios virtuales encontrados bajo "/proc", esto conlleva a una penalización importante de tiempo al tener que realizar repetidas llamadas al sistema para obtener información asociada al proceso, esto es necesario realizarlo por cada llamada al sistema que se está monitorizando.

Una posible mitigación al inconveniente planteado es la de implementar un módulo de cache, este se encarga de proporcionar la información en caso de que la posea, o acceder al directorio virtual adecuado en caso de necesitarlo para posteriormente guardar esa información internamente.

También, es necesario que éste monitoree los archivos que se van abriendo y cerrando por parte del proceso monitorizado para de esa forma invalidar o no la información que se encuentra "cacheada".

Interfaz de usuario y amigabilidad

La interfaz de usuario que presenta el producto es de línea de comandos y muy básica. Mejoras a la misma implicarían implementar una interfaz gráfica en la que se pudiese crear la campaña de inyección y seleccionar el objetivo de la misma, ya sea un proceso o un dispositivo USB.

También se podrían incluir herramientas para visualizar los logs generados permitiendo un mejor análisis de lo ocurrido en las campañas de inyección.

Investigación

En lo que refiere a investigación, una tarea pendiente es realizar pruebas con distintos sistemas para evaluar la respuesta de estos a distintos defectos y posteriormente proponer técnicas de recuperación y/o mitigación.

Capítulo 8

Glosario

Alpha Testing: Actividad en la que se realiza un conjunto de pruebas de funcionamiento a un sistema, hechas por los desarrolladores o testers en el lugar de desarrollo [41].
API: Aplication Programing Interface. Refiere a una interfaz de comunicación entre componentes de software.
Beta Testing: Actividad en la que se realiza un conjunto de pruebas de funcionamiento a un sistema, realizadas por un número limitado de usuarios finales antes de la entrega. En esta etapa pueden solucionarse los defectos detectados por los usuarios [41].
Best Effort: Refiere a modelos de servicio de red donde no se garantiza la correcta transmisión o envío de los paquetes, ni el orden o demora en las transmisiones [43].
Bus: Conjunto de líneas conductoras de hardware utilizadas para la transmisión de datos entre los componentes de un sistema informático.
CPU: Central Processing Unit. Circuito que interpreta y ejecuta instrucciones, encargado del control y el proceso de datos en las computadoras.
Chip: Circuito integrado, montado sobre una placa de silicio, que realiza varias funciones en los dispositivos electrónicos.
COTS: Commercial off-the-shelf, item personalizable utilizado con propósitos no gubernamentales que ha sido vendido alquilado o licenciado para el público general. Termino utilizado para designar distintos productos de software [41].
Debugfs: Sistema de archivos virtual disponible en el kernel de GNU/Linux desde la versión 2.6.10. Permite el intercambio de datos entre espacio de usuario y espacio de kernel [39].
Debugging (Depuración): Proceso que tiene como fin localizar y reducir el número de errores o defectos en un programa [41].
Dlsym: Función del kernel de linux que permite obtener la dirección de un símbolo definido dentro de un objeto hecho accesible mediante la llamada a dlopen() [25].
Driver: Controlador, refiere a los controladores de dispositivos de hardware.
ELF: Executable and Linkable Format, es un formato de archivo para ejecutables, código objeto, bibliotecas compartidas y volcados de memoria utilizado en sistemas UNIX
Excepción: Situación de error detectada por la CPU mientras ejecuta una ins-

trucción, que requiere tratamiento por parte del Sistema Operativo.
Gamma Testing: Actividad en la que se realiza un conjunto de pruebas a un sistema cuando este esta prácticamente listo para ser liberado.
Hacker: Término utilizado para llamar a una persona con grandes conocimientos en informática y telecomunicaciones y que los utiliza con un determinado objetivo. Este objetivo puede o no se maligno o ilegal. La acción de usar sus conocimientos se denomina hacking o hackeo.
Handler: Rutina de software que realiza una determinada tarea.
HDL: Hardware Description Lenguaje. Son lenguajes de programación para la descripción formal de lógica digital y circuitos electrónicos [44].
Host: Usado para referirse a un computador conectado a una red, y que proveen o utilizan servicios de ella.
Hub: Concentrador, usado para hacer referencia a los dispositivos que permite tener varios puertos USB a partir de uno sólo [18].
HWIFI: Hardware Implemented Fault Injection. Hace referencia a herramientas de inyección de defectos implementadas en hardware.
IDT: Interrupt Descriptor Table. Es una estructura de datos usada en la arquitectura x86 para implementar el vector de interrupciones.
IEEE: Institute of Electrical and Electronics Engineers. Asociación de profesionales involucrada en numerosas áreas técnicas y responsable de la promoción de variados e importantes estándares [38].
Interrupción: Señal recibida por la CPU, indicando que debe interrumpir el curso de la ejecución y pasar a ejecutar código específico para tratar esa situación.
Ion pesado: Átomo ionizado de masa mayor a la del helio.
Kernel: Núcleo, parte fundamental de un sistema operativo encargada de gestionar los recursos del sistema.
Lenguaje Ensamblador: Lenguaje de programación de medio nivel, el cual es traducible directamente a lenguaje de máquina.
Lenguaje de Máquina: instrucciones entendibles directamente por la CPU.
Linux: Sistema operativo tipo Unix creado en 1992 y nombre de un kernel creado por Linus Torvalds en 1991.
Stack (Pila): Zona reservada de la memoria o registros hardware donde se almacena temporalmente el estado o información de un programa.
Pin/Pines: Terminal de cada uno de los contactos de un conector o de un componente.
Plataforma: Es el conjunto de hardware y/o software, sobre el cual un programa puede ejecutarse. Incluye la arquitectura de hardware, sistema operativo, lenguajes de programación y sus bibliotecas.
Performance (rendimiento): Medida de unidades de ejecución por unidad de tiempo usada para cuantificar la capacidad de procesamiento de un computador.
POSIX: Portable Operating System Interface. Familia de estándares relacionados especificados por la IEEE para definir APIs para la compatibilidad de software entre los diferentes sistemas operativos Unix [46].
Rootkit: Herramienta de seguridad que captura contraseñas y tráfico desde o hacia una computadora. Es una colección de herramientas que permiten a un hacker crear backdoors en un sistema, recolectando información sobre otros sistemas en la red.
Runtime (Tiempo de Ejecución): período en que un programa es ejecuta-

do.
Sistema operativo: software encargado de controlar y coordinar el uso del hardware entre diferentes programas de aplicación y los diferentes usuarios.
Socket: punto final de un enlace de comunicación de entre dos programas que se ejecutan a través de una red.
Software de base: Software que controla e interactúa con el sistema, proporcionando control sobre el hardware y dando soporte a otros programas.
Strong Symbols: las direcciones de los distintos elementos de un programa en linux son traducidas como símbolos. En particular los strong symbols son únicos en el sistema entre las bibliotecas linkeadas y existentes. De esta manera este único símbolo puede ser usado por todas las bibliotecas que lo necesiten.
Stub: utilizado para hacer referencia a las piezas de software cuya finalidad es remplazar otra ya existente y de esta manera proveer funcionalidades diferentes.
SWIFI: Software Implemented Fault Injection. Hace referencia a herramientas de inyección de defectos implementadas en software.
Syscalls: Las llamadas al sistema son la interfaz fundamental entre una aplicación y el kernel de Linux [26].
Tiempo de compilación: período de tiempo en el que un compilador transforma el código del programa a código que pueda ejecutar una computadora.
Trap (Trampa): Interrupción generada por software.
Unix: Sistema operativo portable, multitarea y multiusuario, desarrollado en 1969 por un grupo de empleados de los laboratorios Bell de AT&T [45].
URB: USB Request Block, (Petición en Bloque USB), estructuras usadas en la transmisión de datos entre el sistema operativo y los dispositivos USB [18].
USB: Universal Serial Bus (Bus Universal en Serie), interfaz que provee un estándar de bus serie para conectar dispositivos a un ordenador [18].
Verilog: Es un lenguaje de descripción de hardware, utilizado en la construcción de modelos de sistemas electrónicos [42].
VHDL: Very high speed integrated circuit Hardware Description. Es un lenguaje de descripción de hardware, utilizado en la construcción de modelos de hardware [44].
Weak Symbols: A diferencia de los Strong Symbols los Weak Symbols pueden existir muchas veces. En caso de que se encuentren muchos símbolos el primero encontrado es devuelto. Esto permite fácilmente sobrescribir entradas en bibliotecas al proveer otra implementación para un mismo símbolo.
x86: Nombre dado al grupo de microprocesadores de la familia de Intel y a la arquitectura que comparten estos procesadores.
Zócalo: Espacio o ranura en una placa base donde se insertan diferentes componentes.

Bibliografía

[1] Becker, James; Flick, Glenn. A practical approach to failure mode,effects and criticality analysis (FMECA) for computing systems.Procedings of the 1996 High-Assurance Systems Engineering Workshop.IEEE 1997.

[2] Mei-Chen, Hsueh; Tsai, T.K.; Iyer, R.K. Fault injection techniques and tools, Computer, Páginas:75 - 82. Abril 1997

[3] Benso, Alfredo; Prinetto, P. Fault Injection Techniques and Tools for Embedded Systems Reliability Evaluation, Páginas 7-47. 2003

[4] Gray, Jim. Talk at ISAT, Redmond, WA on Heisenbugs: A Probabilistic Approach to Availabilty. 20 de abril de 1999

[5] Gray, Jim. Why do computers stop and what can be done about them?, TR-85.7. Junio 1985

[6] Leme, Nelson; Martins, Eliane; Rubira, Cecília. A Software Fault Injection Pattern System- 8th Conference on Pattern Languages of Programs. Setiembre 2001

[7] Vaidyanathan , Kalyanaraman; Trivedi, Kishor S. Extended Classification of Software Faults Based on Aging – ISSRE FastAbstracts. 29 de noviembre 2001

[8] Masum, S.M.; Vanteru, B.C.; Yeasin, M. Fault Injection: A Method for Validating Fault-tolerant System - Workshop on Advances and Innovations in Systems Testing, Memphis. Mayo 2007.

[9] Fault-Tolerant System Design and Verification for Safety-Critical Applications - TOSCA - Project of the ALFA Programme. http://www.cad.polito.it/cooperations/TOSCA/TOSCA.htm. Último acceso enero 2010

[10] Avizienis, Algirdas; Laprie, Jean-Claude; Randell, Brian; Landwehr, Carl. Basic Concepts and Taxonomy of Dependable and Secure Computing. 2004.

[11] Leveson, Nancy; Turner, Clark. An Investigation of the Therac-25 Accidents. 7 de Julio de 1993. http://courses.cs.vt.edu/~ cs3604/lib/Therac_25/Therac_1.html. Último acceso marzo 2010.

[12] Lions, J. L. ARIANE 5 - Flight 501 Failure. 19 de julio de 1996. http://www.di.unito.it/~ damiani/ariane5rep.html. Último acceso marzo 2010.

[13] Love, Robert. Linux Kernel Development Second Edition. Enero 2005. Capítulo 5: System Calls.

[14] Manjarrez, Jorge. Implementing Linux System Calls. Diciembre 1999. http://www.linuxjournal.com/article/3326. Último acceso junio 2010.

[15] Clark, Jeffrey A.; Pradhan, Dhiraj K. Fault injection: A method for validating computer system dependability. Volume 28, Issue 6, Páginas 47 - 56. Enero 1995.

[16] Keniston, Jim; S Panchamukhi, Prasanna; Hiramatsu, Masami. Kernel Probes (Kprobes). Documentación de Kprobes incorporada a la documentación del kernel de Linux versión 2.6.35.4

[17] Goswami, Sudhanshu. An introduction to KProbes. Artículo publicado en lwn.net el 18 de Abril de 2005. http://lwn.net/Articles/132196/.

[18] Universal Serial Bus Specification Revision 2.0 - Compaq Computer Corporation, Hewlett-Packard Company, Intel Corporation, Lucent Technologies Inc, Microsoft Corporation, NEC Corporation, Koninklijke Philips Electronics N.V. 27 de Abril de 2000.

[19] USB Implementers Forum. http://www.usb.org/developers/ssusb. Último acceso setiembre 2010.

[20] ASUS. Unveils First Motherboards to Feature True USB 3.0 and SATA 6Gb/s Performance. 28 de octubre de 2009. http://www.asus.com/News.aspx?N_ID=oyMjcbYR98UtjHD0. Último acceso setiembre 2010.

[21] GIGABYTE. Offers info on GIGABYTE USB 3.0 motherboards and 3rd party SuperSpeed products. 23 de diciembre de 2009. http://www.gigabyte.com/press-center/news-page.aspx?nid=835. Último acceso setiembre 2010.

[22] Linux Device Drivers, third edition - Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman - capítulo 13 USB Drivers. Febrero 2005.

[23] Manual GNU ld 2.19 (GNU Binutils openSUSE 11.1) - 10 de Noviembre de 2008.

[24] Manual ld.so de Linux man-pages 3.13 - 27 de Octubre de 2008.

[25] Manual dlopen de Linux man-pages 3.13 - 27 de Octubre de 2008.

[26] Manual `syscalls` de Linux man-pages 3.13 - 27 de Octubre de 2008.

[27] Manual ptrace de Linux man-pages 3.13 - 27 de Octubre de 2008.

[28] Sieh, Volkmar. Fault-Injector using UNIX ptrace Interface. 30 de Setiembre de 1993.

[29] Broadwell, Pete; Sastry, Naveen; Traupman, Jonathan. Fault Injection in Glibc (FIG). 10 de diciembre de 2001.

[30] The Berkeley/Stanford Recovery-Oriented Computing (ROC) Project. http://roc.cs.berkeley.edu/. Último acceso setiembre 2010.

[31] Zaitcev, Pete. The USBMon: USB monitoring framework - presentado en el Linux Symposium 2005.

[32] Linux USB Project. http://www.linux-usb.org/. Último acceso setiembre 2010.

[33] Venkateswaran, Sreekrishnan. Essential Linux Device Drivers, capitulo 11 - Universal Serial Bus. 2008

[34] Axelson, Jan. USB Complete: The Developer's Guide, Fourth Edition - capítulo 4 Enumeration: How the host learns about devices - 2009.

[35] Libusb. http://libusb.sourceforge.net. Último acceso mayo 2010.

[36] Hirofuchi, Takahiro; Kawai, Eiji; Fujikawa, Kazutoshi; Sunahara, Hideki. USB/IP—A Peripheral Bus Extension for Device Sharing over IP Network. 2005 USENIX Annual Technical Conference.

[37] USB/IP Project. http://usbip.sourceforge.net/. Último acceso setiembre 2010.

[38] IEEE. http://www.ieee.org. Último acceso julio 2010.

[39] Kroah-Hartman, Greg. Debugfs - yet another in-kernel file system. Artículo publicado en lwn.net el 9 de Diciembre de 2004. http://lwn.net/Articles/115282/.

[40] Corbet, Jonathan. Debugfs. Documentación de Debugfs incorporada a la documentación del kernel de linux versión 2.6.35.4.

[41] Sommerville, Ian. Software Engineering 8a Edición, Páginas: 79, 81, 408. 2007.

[42] Verilog. http://www.verilog.com. Último acceso marzo 2010.

[43] Kurose, James; Ross, Keith. Computer Networking A Top-Down Approach Featuring the Internet. 2000.

[44] Tutorial vhdl online. http://www.vhdl-online.de/tutorial/. Último acceso setiembre 2010.

[45] Sitio de unix. http://www.unix.org/what_is_unix/history_timeline.html. Último acceso setiembre 2010.

[46] The Open Group - POSIX.1-2008. http://www.opengroup.org/onlinepubs/9699919799/. Último acceso setiembre 2010.

[47] OpenSUSE. http://es.opensuse.org/Bienvenidos_a_openSUSE.org. Último acceso setiembre 2010.

Apéndice A

Pruebas de prototipo Ptrace

Para el prototipo de inyección de defectos basado en Ptrace fueron realizadas varias pruebas. A continuación se describen las mismas y los resultados obtenido.

A.1. Pruebas sobre File System

Conjunto de pruebas de intercepción de llamadas al sistema que refieren al manejo de archivos, más específicamente a la apertura, cierre, lectura y escritura de los mismos.

A.1.1. Pruebas sobre apertura de archivos

Objetivo

Mediante la utilización del prototipo realizado con ptrace, realizar la inyección de defectos en la apertura de archivos mediante la intercepción de la llamada al sistema open.

Pruebas

Se construye un programa sencillo que abre y cierra el archivo /tmp/test.txt esperando cuatro segundos entre cada acción e imprimiendo éxito o error del intento. En caso de error el código de error obtenido.
Los defectos inyectados en la prueba se describen a continuación:

1. Retorno de el código de error correspondiente a ENOENT (No such file or directory) la primera vez que se invoca a open del archivo especificado.

2. Retorno de el código de error correspondiente a ENOENT tercera vez que se invoca a open del archivo especificado.

3. Retorno del código de error correspondiente a ENOENT desde la segunda vez que se invoca a open del archivo especificado y para todas las llamadas subsecuentes.

Fueron realizadas 3 pruebas en cada una de las cuales se inyecta cada uno de los defectos antes descriptos.

Resultados

En cada una de las pruebas se logra inyectar con éxito el defecto correspondiente en el momento indicado. El programa inyectado al llamar a open recibe el código de error especificado y se ve imposibilitado de abrir el archivo. En los dos primeros casos el error se ejecuta solo en la vez indicada mientras que el en tercero se ejecuto en esa y todas las llamadas subsecuentes a open.

A.1.2. Pruebas sobre cierre de archivos

Objetivo

Mediante la utilización del prototipo realizado con ptrace, realizar la inyección de defectos en el cierre de archivos mediante la intercepción de la llamada al sistema close.

Prueba

Sobre el mismo programa de la prueba de apertura de archivos se inyecta un defecto que retorna el código de error correspondiente a EIO (I/O error) la primera vez que se invoca a close del archivo especificado.

Resultado

Se inyecta con éxito el defecto correspondiente en la primera llamada a close luego de comenzada la inyección. El programa inyectado al llamar a close recibe el código de error especificado y se ve imposibilitado de cerrar el archivo en el primer intento. En los intentos subsecuentes lo logra sin problemas.

A.1.3. Pruebas sobre lectura de archivos

Objetivo

Mediante la utilización del prototipo realizado con ptrace realizar la inyección de defectos en la lectura de archivos mediante la intercepción de la llamada al sistema read.

Prueba

Se construye un programa sencillo que escribe una cadena de caracteres en el archivo /tmp/test.txt (cadena “1234567890”) y luego lee los caracteres escritos, esperando cuatro segundos entre cada acción e imprimiendo lo que lee y escribe en el archivo.

Se definió un defecto de lectura que sustituye los caracteres leídos desde una posición dada por una cadena indicada.

En el caso de la prueba se sustituye lo leído por la cadena “hola” desde la posición 0 la primera vez que se ejecuta el read sobre el archivo.

Resultado

Al realizar la primera llamada a read luego de comenzada la inyección se obtuvo la cadena "hola567890". Cabe mencionar que en el archivo dicha cadena no está presente. En las subsecuentes lecturas el valor obtenido es "1234567890".

A.1.4. Pruebas sobre escritura de archivos

Objetivo

Mediante la utilización del prototipo realizado con ptrace realizar la inyección de defectos en la escritura de archivos mediante la intercepción de la llamada al sistema write.

Prueba

Se utiliza el programa descripto para las pruebas de lectura de archivos, se define un defecto de escritura que sustituye los caracteres de la cadena que debe ser escrita desde una posición dada por una cadena indicada.

En el caso de la prueba se sustituye por la cadena "hola" desde la posición 0 en la cadena que debe ser escrita, en la primera vez que se ejecuta el write sobre el archivo luego de comenzada la inyección.

Resultado

Al realizar la primera llamada a write luego de comenzada la inyección se escribió en el archivo la cadena "hola567890". En las subsecuentes escrituras el valor escrito es "1234567890".

A.2. Pruebas sobre Red

Conjunto de pruebas de intercepción de llamadas al sistema que refieren al uso de funciones de red, más específicamente a la lectura y escritura sobre socket.

A.2.1. Pruebas sobre lectura de sockets

Objetivo

Mediante la utilización del prototipo realizado con ptrace realizar la inyección de defectos en la lectura de socket mediante la intercepción de la llamada al sistema read.

Prueba

Usando el comando netcat en una consola se abre una conexión sobre un puerto de la siguiente manera:

```
netcat -l -p "puerto"
```

Desde otra consola se ejecuta de nuevo el comando netcat para conectar al puerto sobre el que escucha la primera ejecución del comando:

```
netcat "ip" "puerto"
```

Posteriormente se define un defecto para inyectar sobre la llamada al sistema read. Esta llamada es a través de la que netcat realiza las lecturas sobre el socket abierto. El defecto consiste en cambiar los primeros 4 caracteres de cada cadena recibida por la cadena "hola". La inyección puede ser realizada sobre cualquiera de los procesos antes iniciados.

Resultado

Cada vez que desde la consola de la ejecución de netcat no inyectada se envía una cadena, el proceso que es inyectado muestra en pantalla dicha cadena pero con los primeros cuatro caracteres remplazados por la cadena "hola". Si el largo del envío es menor a 4 se sustituyen los caracteres enviados por el substring de "hola" que corresponda.

A.2.2. Pruebas sobre escritura de sockets

Objetivo

Mediante la utilización del prototipo realizado con ptrace realizar la inyección de defectos en la escritura de socket mediante la intercepción de la llamada al sistema write.

Prueba

Usando el comando netcat en una consola se abre una conexión sobre un puerto de la siguiente manera:

```
netcat -l -p "puerto"
```

Desde otra consola se ejecuta de nuevo el comando netcat para conectar al puerto sobre el que escucha la primera ejecución del comando:

```
netcat "ip" "puerto"
```

Posteriormente se define un defecto para inyectar sobre la llamada al sistema write. Esta llamada es a través de la que netcat realiza las escrituras sobre el socket abierto. El defecto consiste en cambiar los primeros 4 caracteres de cada cadena enviada por la cadena "hola". La inyección puede ser realizada sobre cualquiera de los procesos antes iniciados.

Resultado

Cada vez que desde la consola de la ejecución de netcat inyectada se envía una cadena, el proceso que no es inyectado muestra en pantalla dicha cadena pero con los primeros cuatro caracteres remplazados por la cadena "hola". Si el largo del envío es menor a 4 se sustituyen los caracteres enviados por el substring de "hola" que corresponda.

Apéndice B

Pruebas de prototipo LD_PRELOAD

Para el prototipo de inyección de defectos basado en LD_PRELOAD se realizaron varias pruebas. A continuación se describen las mismas y los resultado obtenidos.

La utilización de LD_PRELOAD está enfocada a interceptar llamadas a la librería Glibc. Para interceptar funcionalidades ofrecidas por el sistema de forma más general se definieron defectos a inyectar que atacan funcionalidades del sistema mediante la intercepción de varias funciones de Glibc. Por ejemplo para interceptar apertura de archivos se interceptan las funciones open y fopen de Glibc.

El caso de interceptar funcionalidades de Glibc de forma individual es un caso particular del que se muestra aquí, y por ende más sencillo de implementar.

B.1. Pruebas sobre File System

Conjunto de pruebas de intercepción de llamadas al sistema que refieren al manejo de archivos, más específicamente a la apertura, cierre, lectura y escritura de los mismos.

B.1.1. Pruebas sobre apertura de archivos

Objetivo

Mediante la utilización del prototipo realizado con LD_PRELOAD, realizar la inyección de defectos en la apertura de archivos mediante la intercepción de las funciones de Glibc open, fopen y fdopen.

Pruebas

Se crea un programa sencillo que abre y cierra el archivo /tmp/test.txt esperando cuatro segundos entre cada acción e imprimiendo éxito o error del intento y en caso de error el código de error obtenido. Se definen varios defectos que se describen a continuación:

1. Retorno del código de error correspondiente a ENOENT (No such file or directory) la primera vez que se invoca a las funciones open, fopen o fdopen sobre el archivo especificado.
2. Retorno de el código de error correspondiente a ENOENT tercera vez que se invoca a las funciones open, fopen o fdopen sobre el archivo especificado.
3. Retorno del código de error correspondiente a ENOENT desde la segunda vez que se invoca a las funciones open, fopen o fdopen sobre el archivo especificado y para todas las llamadas subsecuentes.

Se realizaron nueve pruebas, en cada una de las cuales se inyectaba cada uno de los defectos antes descriptos variando el programa para que use el open, fopen y fdopen para abrir el archivo.

Resultados

En cada una de las pruebas se logró inyectar con éxito el defecto correspondiente en el momento indicado. El programa es inyectado al llamar a la función para abrir el archivo, en ese momento recibe el código de error especificado y se ve imposibilitado de abrir el archivo. En los dos primeros casos el error se ejecuta solo en la vez indicada mientras que el en tercero se ejecuta en esa y todas las llamadas subsecuentes a la función de apertura.

B.1.2. Pruebas sobre cierre de archivos

Objetivo

Mediante la utilización del prototipo realizado con LD_PRELOAD, realizar la inyección de defectos en el cierre de archivos mediante la intercepción de las funciones de Glibc close y fclose.

Prueba

Sobre el mismo programa de la prueba de apertura de archivos se define un defecto que retorna de el código de error correspondiente a EIO (I/O error) la primera vez que se llama a la función close o fclose sobre el archivo especificado.

Se realizan dos pruebas inyectando el defecto descripto variando el programa para que use close y fclose.

Resultado

Se logra inyectar con éxito el defecto correspondiente en la primera llamada a la función de cierre de archivos luego de comenzada la inyección. El programa inyectado al llamar a la función recibe el código de error especificado y se ve imposibilitado de cerrar el archivo en el primer intento. En los intentos subsecuentes lo logra sin problemas.

B.1.3. Pruebas sobre lectura de archivos

Objetivo

Mediante la utilización del prototipo realizado con LD_PRELOAD realizar la inyección de defectos en la lectura de archivos mediante la intercepción de las funciones de Glibc read y fread.

Prueba

Se crea un programa sencillo que escribe una cadena de caracteres en el archivo /tmp/test.txt (cadena “1234567890”) y luego lee los caracteres escritos, esperando cuatro segundos entre cada acción e imprimiendo lo que lee y escribe en el archivo.

Se define un defecto de lectura que sustituye los caracteres leídos desde una posición dada por una cadena indicada.

En el caso de la prueba se sustituye lo leído por la cadena “hola” desde la posición 0 la primera vez que se ejecuta la función de lectura sobre el archivo. La prueba se ejecuto dos veces, una con el programa usando read y otra con el programa usando fread.

Resultado

En ambas pruebas al realizar la primera llamada a lectura se obtiene la cadena “hola567890”. Cabe mencionar que en el archivo dicha cadena no está presente. En las subsecuentes lecturas el valor obtenido es “1234567890”.

B.1.4. Pruebas sobre escritura de archivos

Objetivo

Mediante la utilización del prototipo realizado con LD_PRELOAD realizar la inyección de defectos en la escritura de archivos mediante la intercepción de la funciones de Glibc write y fwrite.

Prueba

Se utiliza el programa descripto para las pruebas de lectura de archivos.
Se define un defecto de escritura que sustituye los caracteres de la cadena que debe ser escrita desde una posición dada por una cadena indicada.

En el caso de la prueba se sustituye por la cadena “hola” desde la posición 0 en la cadena que debe ser escrita, en la primera vez que se ejecuta la función de escritura sobre el archivo. La prueba se ejecutó dos veces, una con el programa usando write y otra con el programa usando fwrite.

Resultado

En ambas pruebas al realizar la primera llamada a escritura se escribe en el archivo la cadena “hola567890”. En las subsecuentes escrituras el valor escrito es “1234567890”.

B.2. Pruebas sobre Red

Conjunto de pruebas de intercepción de llamadas al sistema que refieren al uso de funciones de red, más específicamente a la lectura y escritura sobre socket.

B.2.1. Pruebas sobre lectura de sockets

Objetivo

Mediante la utilización del prototipo realizado con LD_PRELOAD realizar la inyección de defectos en la lectura de socket mediante la intercepción de las funciones de Glibc read y fread sobre un socket.

Prueba

Se crearon dos programas sencillos, uno llamado cliente y otro servidor. El cliente se conecta al servidor implementado mediante una conexión tcp, envía 10 mensajes desplegando lo enviado y la respuesta recibida. El servidor implementa un servidor simple que recibe mensajes por un socket tcp. Tanto servidor como cliente despliegan en pantalla el mensaje y la respuesta. El mensaje enviado desde el cliente es siempre "1234567890" mientras que la respuesta del servidor cada vez que lee un mensaje es "El mensaje fue recibido".

Las pruebas consisten en inyectar al servidor de forma que al leer del socket se sustituya la cadena "hola" desde la posición 0 en la cadena leída con la siguiente configuración:

1. la primera vez que se invoca a la función read o fread sobre un socket cualquiera.
2. la segunda vez que se invoca a la función read o fread sobre un socket cualquiera.
3. la tercera vez que se invoca a la función read o fread sobre un socket cualquiera y todas las subsecuentes llamadas.

Se realizaron seis pruebas, para cada uno de los defectos antes mencionados con el servidor usando read y luego fread.

Resultado

En cada una de las pruebas se logra inyectar con éxito el defecto correspondiente en el momento indicado. El servidor es inyectado al llamar a la función para leer del socket mostrando "hola567890". En los dos primeros casos el defecto se introduce solo en la vez indicada mientras que el en tercero se ejecuta en esa y todas las llamadas subsecuentes a la función de lectura.

B.2.2. Pruebas sobre escritura de sockets

Objetivo

Mediante la utilización del prototipo realizado con ptrace realizar la inyección de defectos en la escritura de socket mediante la intercepción de las funciones de Glibc write y fwrite sobre un socket.

Prueba

Se utilizaron los programas servidor y cliente especificados para las pruebas de lectura de socket.

Los defectos consisten en inyectar al servidor de forma que al escribir la respuesta en el socket se sustituya la cadena "hola" desde la posición 0 en la cadena de respuesta con la siguiente configuración:

1. la primera vez que el proceso invoca a la función write o fwrite sobre un socket cualquiera.

2. la segunda vez que el proceso invoca a la función write o fwrite sobre un socket cualquiera.

3. la tercera vez que el proceso invoca a la función write o fwrite sobre un socket cualquiera y todas las subsecuentes llamadas.

Se realizaron seis pruebas, para cada uno de los defectos antes mencionados con el servidor usando write y luego fwrite.

Resultado

En cada una de las pruebas se logró inyectar con éxito el defecto correspondiente en el momento indicado. El servidor es inyectado al llamar a la función para escribir en el socket enviando la respuesta "holaensaje fue recibido". En los dos primeros casos el defecto se introdujo solo en la vez indicada mientras que el en tercero se ejecuto en esa y todas las llamadas subsecuentes a la función de escritura.

Apéndice C

Pruebas de prototipo KProbes

Para probar el prototipo basado en KProbes se utiliza un proceso de usuario que realiza repetidamente un read y write sobre un determinado archivo.

El proceso test escribe continuamente la cadena “0123456789” por lo que en su próxima lectura de 10 bytes procederá a leer dicha cadena. El modulo inyector, realiza la inyección de la cadena “hola” a partir de la posición 0 del buffer, por lo que en su próxima lectura el proceso de usuario deberá leer “hola456789”.

Además el buffer debe ser inalterado, por lo que al retornar de la invocación a la llamada al sistema debe contener la misma cadena, es decir “0123456789”.

Luego de ejecutar el test se procede a correr el modulo del kernel a través de la instrucción insmod kretprobesfi.ko pid=xxx donde xxx representa el pid del proceso test.

Los resultados de la salida del proceso test son los siguientes:

```
antes open
despues open fd=3 errno=0 ENOENT=2
*************************************
antes sleep read
despues sleep read
antes read
despues read fd=4 errno=0 leidos =0
caracteres leidos:
antes sleep write
despues sleep write
antes write
despues write fd = errno=0 escritos =10
caracteres escritos: 0123456789
*************************************

*************************************
antes sleep read
despues sleep read
antes read
despues read fd=4 errno=0 leidos =10
caracteres leidos: 0123456789
antes sleep write
```

```
despues sleep write
antes write
despues write fd = errno=0 escritos =10
caracteres escritos: 0123456789
*************************************

*************************************
antes sleep read
despues sleep read
antes read
despues read fd=4 errno=0 leidos =10
caracteres leidos: 0123456789
antes sleep write
despues sleep write
antes write
despues write fd = errno=0 escritos =10
caracteres escritos: 0123456789
*************************************

*************************************
antes sleep read
despues sleep read
antes read
despues read fd=4 errno=0 leidos =10
caracteres leidos: hola456789
antes sleep write
despues sleep write
antes write
despues write fd = errno=0 escritos =10
caracteres escritos: 0123456789
*************************************
```

En esta última salida se puede ver que los caracteres leídos son "hola456789" y que cuando se imprime el buffer escrito se imprime "0123456789". Por lo tanto se puede observar que la prueba realizada funcionó correctamente.

Apéndice D

Estructuras USB

D.1. usb_device_descriptor

Estructura que contiene la descripción del dispositivo.

```
struct usb_device_descriptor {
  __u8  bLength;
  __u8  bDescriptorType;
  __le16 bcdUSB;
  __u8  bDeviceClass;
  __u8  bDeviceSubClass;
  __u8  bDeviceProtocol;
  __u8  bMaxPacketSize0;
  __le16 idVendor;
  __le16 idProduct;
  __le16 bcdDevice;
  __u8  iManufacturer;
  __u8  iProduct;
  __u8  iSerialNumber;
  __u8  bNumConfigurations;
} __attribute__ ((packed));
```

Campos:

- bLength: largo en bytes del descriptor.
- bDescriptorType: especifica el tipo del descriptor. El valor de este campo es USB_DT_DEVICE.
- bcdUSB: versión del estándar USB que cumple el dispositivo y descriptor.
- bDeviceClass: código de clase del dispositivo (asignado por USB Implementers Forum). Los valores entre 01h y FEh son reservados para las clases definidas por el estándar. Las clases definidas por el vendedor usan FFh.
- bDeviceSubClass: especifica la subclase del dispositivo. Esto permite dar soporte a características adicionales compartidas por un grupo de funciones en una clase.
- bDeviceProtocol: indica el protocolo para la clase y subclase.

- bMaxPacketSize0: indica el máximo largo de paquete del endpoint 0.
- idVendor: identificador del vendedor (asignado por USB Implementers Forum).
- idProduct: identificador del producto. Es asignado por el vendedor y junto con el idVendor puede ayudar a seleccionar el driver adecuado para el dispositivo.
- bcdDevice: número de liberación del producto en BCD.
- iManufacturer: índice que apunta al string que describe el fabricante, o 0 si no existe la descripción de fabricante.
- iProduct: índice que apunta al string que describe el producto, o 0 si no existe la descripción de producto.
- iSerialNumber: es un índice que apunta al string que contiene el número de serie del producto o 00h si no tiene número de serie. Este es útil si se tiene más de un dispositivo idéntico en el bus y el host necesita diferenciarlos luego de un reinicio.
- bNumConfigurations: el número de configuraciones que el dispositivo soporta a la velocidad que está operando.

bDeviceClass	**Descripción**
00h	El descriptor de interfaz especifica la clase y la función no usa un descriptor de asociación de interfaz.
02h	Dispositivo de comunicación.
09h	Hub
0Fh	Dispositivo de cuidado de salud.
DCh	Dispositivo de diagnostico. bDeviceSubclass = 01h, bDeviceProtocol = 01h: para USB2.
E0h	Wireless Controller (puede declararse a nivel de interfaz) bDeviceSubclass = 01h: Bluetooth programming interface
EFh	Misceláneo bDeviceSubclass = 01h bDeviceProtocol = 01h: active sync bDeviceProtocol = 02h: Palm sync bDeviceSubclass = 02h bDeviceProtocol = 01h: descriptor de interfaz de asociación bDeviceProtocol = 02h: adaptador multifuncional de periférico (Wireless USB)
FFh	Vendor-specific (puede declararse a nivel de interfaz)

D.2. usb_config_descriptor

Todo dispositivo tiene al menos una configuración que especifica las características del dispositivo. Típicamente basta con una única configuración, pero dispositivos con múltiples funcionalidades u opciones de alimentación pueden usar múltiples configuraciones. Solo una de estas puede estar activa.

```
struct usb_config_descriptor {
  __u8  bLength;
  __u8  bDescriptorType;
  __le16 wTotalLength;
  __u8  bNumInterfaces;
  __u8  bConfigurationValue;
  __u8  iConfiguration;
  __u8  bmAttributes;
  __u8  bMaxPower;
} __attribute__ ((packed));
```

Campos:

- bLength: largo en bytes del descriptor.
- bDescriptorType: especifica el tipo del descriptor. El valor de este campo es USB_DT_CONFIG
- wTotalLength: largo en bytes del descriptor de configuración y todos sus descriptores subordinados.
- bNumInterfaces: número de interfaces en la configuración. El mínimo es 01h.
- bConfigurationValue: identifica la configuración para las llamadas a usb_get_configuration y usb_set_configuration. Un usb_set_configuration con valor cero provoca que el dispositivo quede en estado Not_Configured.
- iConfiguration: índice del string que describe la configuración, o 0 si no existe la descripción.
- bmAttributes: el bit 6 vale 1 si el dispositivo tiene alimentación propia y 0 si necesita alimentado por el bus. El bit 5 vale 1 si el dispositivo soporta remote wakeup, con lo que habilita a un que un dispositivo USB en estado suspendido indique al host que el dispositivo quiere comunicarse. Los bits 4 al 0 deben valer 0. El bit 7 debe valer uno por compatibilidad con USB 1.0.
- bMaxPower: alimentación del bus requerida por el dispositivo, en unidades de 2 mA (USB 2.0) u 8 mA (SuperSpeed). El máximo que puede ser solicitado es de 500 mA para USB 2.0 y 900 mA para Super-Speed. Si la cantidad solicitada no puede ser provista el host puede rechazar la configuración del dispositivo y el driver debe solicitar una configuración alternativa si esta existe.

D.3. usb_interface_descriptor

Una interfaz refiere a una función implementada por el dispositivo. Muchos dispositivos especifican la clase al nivel de la interfaz. Asignar funciones a interfaces permite que una configuración soporte múltiples funciones.

```
struct usb_interface_descriptor {
  __u8  bLength;
```

```
  __u8  bDescriptorType;
  __u8  bInterfaceNumber;
  __u8  bAlternateSetting;
  __u8  bNumEndpoints;
  __u8  bInterfaceClass;
  __u8  bInterfaceSubClass;
  __u8  bInterfaceProtocol;
  __u8  iInterface;
} __attribute__ ((packed));
```

Campos:

- bLength: largo en bytes del descriptor.
- bDescriptorType: especifica el tipo del descriptor. El valor de este campo es USB_DT_INTERFACE.
- bInterfaceNumber: número que identifica la interfaz.
- bAlternateSetting: número que identifica una configuración alternativa para la interfaz identificada por bInterfaceNumber.
- bNumEndpoints: número de endpoints soportados por esta interfaz adicionalmente al 0. Si la interfaz solo cuenta con el endpoint 0 el valor de este campo es 0.
- bInterfaceClass: tiene el mismo significado que el bDeviceClass del usb_device_descriptor pero para dispositivos donde la clase se especifica en la interfaz.
- bInterfaceSubClass: tiene el mismo significado que el bDeviceSubClass del usb_device_descriptor pero para dispositivos donde la clase se especifica en la interfaz.
- bInterfaceProtocol: tiene el mismo significado que el bDeviceProtocol del usb_device_descriptor pero para dispositivos donde la clase se especifica en la interfaz.
- iInterface: índice del string que describe la interfaz, o 0 si no existe la descripción.

Valores posibles para bInterfaceClass de usb_interface_descriptor	
bInterfaceClass	**Descripción**
00	Reservado
01	Audio
02	Clase de dispositivo de comunicaciones: interfaz de comunicaciones
03	Dispositivo de interfaz humana (HID: Human interface device)
05	Físico
06	Imagen
07	Impresión
08	Almacenamiento
09	Hub
0A	Clase de dispositivo de comunicaciones: interfaz de datos
0B	Tarjeta inteligente
0D	Contenido de seguridad
0E	Vídeo
0F	Dispositivos de cuidado de salud (pueden declararse a nivel de dispositivo)
DC	Dispositivos de diagnóstico pueden declararse a nivel de dispositivo bInterfaceSubclass= 01h, bInterfaceProtocol = 01h. Dispositivos USB 2.0
E0	controlador Wireless bInterfaceSubclass = 01h bInterfaceProtocol = 01h: interfaz de programación Bluetooth (pueden declararse a nivel de dispositivo) bInterfaceProtocol = 02h: UWB Radio control interface (Wireless USB) bInterfaceProtocol = 03h: NDIS remoto bInterfaceSubclass = 02h. Adaptadores para Host y dispositivo (Wireless USB)
EF	Miscellaneous bInterfaceSubclass = 01h bInterfaceProtocol = 01h: active sync bInterfaceProtocol = 02h: Palm sync bInterfaceSubclass = 03h. Cable based association framework (Wireless USB)
FE	Aplicación específica bInterfaceSubclass = 01h. dispositivo de actualización de firmware bInterfaceSubclass = 02h. puente IrDA bInterfaceSubclass = 03h. Test y mediciones
FF	De vendedor específico (pueden declararse a nivel de dispositivo)

D.4. usb_endpoint_descriptor

Cada endpoint especificado en una interfaz tiene un descriptor. El endpoint 0 no tiene un descriptor dado que cada dispositivo debe tenerlo. Como se mencionó antes el usb_device_descriptor contiene el tamaño máximo de paquete del endpoint 0. El host obtiene los descriptores de los endpoints en el retorno de un usb_get_configuration junto con el descriptor de la configuración.

Para los dispositivos de la clase audio 1.0 se extiende el descriptor del endpoint con dos bytes específicos para información de audio.

```
struct usb_endpoint_descriptor {
  __u8  bLength;
  __u8  bDescriptorType;
  __u8  bEndpointAddress;
  __u8  bmAttributes;
  __le16 wMaxPacketSize;
  __u8  bInterval;
  /* NOTE:  these two are _only_ in audio endpoints. */
  /* use USB_DT_ENDPOINT*_SIZE in bLength, not sizeof. */
  __u8  bRefresh;
  __u8  bSynchAddress;
} __attribute__ ((packed));
```

Campos:

- bLength: largo en bytes del descriptor.
- bDescriptorType: especifica el tipo del descriptor. El valor de este campo es USB_DT_ENDPOINT.
- bEndpointAddress: especifica el número y dirección del endpoint. Los bits 3..0 son el número del endpoint. Los dispositivos Low-speed tiene un máximo de 3 endpoints, los full y high-speed tienen un máximo de 16. El bit 7 is la direction, OUT = 0 e IN = 1. Los bits 6..4 no son usados y deben ser cero.

```
bEndpointAddress
  Bit 3...0: The endpoint number
  Bit 6...4: Reserved, reset to zero
  Bit 7: Direction, ignored for control endpoints
          0 = OUT endpoint
          1 = IN endpoint
```

- bmAttributes: los bits 1..0 especifican el tipo de transferencia que soporta el endpoint: 00=control, 01=isochronous, 10=bulk, 11=interrupt. Los bits 7..6 están reservados, deben valer 0. El significado de los bits siguientes varían según el tipo de endpoint y velocidad. Para endpoints isochronous los bits 5..2 pueden indicar el tipo de sincronización y el uso de datos y las respuestas.
- wMaxPacketSize: tamaño máximo en bytes de paquete que soporta el endpoint. Los valores permitidos varían entre los distintos tipos de endpoints y velocidades de dispositivo. Para USB 2.0, los bits 10..0 son el máximo tamaño de paquete con un rango entre 0–1024, para USB 1.x, el rango es 0–1023. En USB 2.0, los bits 12..11 indican cuantas transacciones adicionales por microframe soporta un endpoint interrupt o isochronous en high-speed: 00 = ninguna (total de 1 / microframe), 01 = una adicional (total de 2/microframe), 10 = 2 adicionales (total de 3/microframe), 11 = reservado. En USB 1.x, estos bits son reservados valen y cero. Los bits 15..13 son reservados y valen 0.

```
bmAttributes
  Bits 1..0: Transfer Type
    00 = Control
    01 = Isochronous
```

```
  10 = Bulk
  11 = Interrupt
If not an isochronous endpoint, bits 5..2
 are reserved and must be set to zero. If
 isochronous, they are defined as follows:
Bits 3..2: Synchronization Type
  00 = No Synchronization
  01 = Asynchronous
  10 = Adaptive
  11 = Synchronous
Bits 5..4: Usage Type
  00 = Data endpoint
  01 = Feedback endpoint
  10 = Implicit feedback Data endpoint
  11 = Reserved
All other bits are reserved and must be
 reset to zero. Reserved bits must be
 ignored by the host.
```

- bInterval: si el endpoint es isochronous o interrupt indica el intervalo de servicio. Este es un período que el host debe reservar para las transacciones al endpoint. Es un número de frames para low o full speed, o microframes para high speed. Los valores varían según los tipos de endpoints, las velocidades y la versión de USB.
  - Para endpoints interrupt en low-speed, es la máxima latencia en ms en un rango de 10–255.
  - Para endpoints interrupt e isochronous en full-speed en dispositivos USB 1.x, el intervalo es igual a bInterval en ms. Para endpoints interrupt el valor está en el rango 1–255. Para isochronous en dispositivos USB 1.x, el valor debe ser 1.
  - Para endpoints isochronous en dispositivos full-speed USB 2.0, el rango está entre 1–16, y el intervalo es 2bInterval-1 en ms, por lo que va desde 1 ms a 32,768 segundos.
  - Para high-speed el valor es en unidades de 125 μs. El valor para endpoints interrupt e isochronous está en el rango 1–16, y el intervalo se calcula como 2bInterval-1, por lo que va desde 125 μs a 4,096 segundos.

D.5. urb

La estructura URB es utilizada para intercambiar datos entre el sistema operativo y un endpoint de un dispositivo USB.

```
struct URB {
  /* private: USB core and host controller only fields in the urb */
  struct kref kref;   /* reference count of the URB */
  void *hcpriv;       /* private data for host controller */
  atomic_t use_count; /* concurrent submissions counter */
  atomic_t reject;    /* submissions will fail */
```

```
  int unlinked;       /* unlink error code */
  /* public: documented fields in the URB that can be used by drivers */
  struct list_head urb_list; /* list head for use by the urb's
    * current owner */
  struct list_head anchor_list; /* the URB may be anchored */
  struct usb_anchor *anchor;
  struct usb_device *dev;       /* (in) pointer to associated device */
  struct usb_host_endpoint *ep; /* (internal) pointer to endpoint */
  unsigned int pipe;            /* (in) pipe information */
  int status;                   /* (return) non-ISO status */
  unsigned int transfer_flags;  /* (in) URB_SHORT_NOT_OK | ..*/
  void *transfer_buffer;        /* (in) associated data buffer*/
  dma_addr_t transfer_dma;      /* (in) dma addr for transfer_buffer */
  struct usb_sg_request *sg;    /* (in) scatter gather buffer list */
  int num_sgs;                  /* (in) number of entries in the sg list */
  u32 transfer_buffer_length;   /* (in) data buffer length */
  u32 actual_length;            /* (return) actual transfer length */
  unsigned char *setup_packet;  /* (in) setup packet (control only) */
  dma_addr_t setup_dma;         /* (in) dma addr for setup_packet */
  int start_frame;              /* (modify) start frame (ISO) */
  int number_of_packets;        /* (in) number of ISO packets */
  int interval;                 /* (modify) transfer interval
/* (INT/ISO) */
  int error_count;              /* (return) number of ISO errors */
  void *context;                /* (in) context for completion*/
  usb_complete_t complete;      /* (in) completion routine */
  struct usb_iso_packet_descriptor iso_frame_desc[0]; /* (in)
ISO ONLY */
}
```

Campos privados de uso interno del HCD y USB Core:

- kref: contador de referencias del URB.
- hcpriv: datos privados para el controlador de host.
- reject: para indicar que el envío fallará. /* submissions will fail */
- unlinked: código de error de unlink

Campos públicos que pueden ser usados por los drivers:

- dev: Puntero a la estructura usb_device con la información del dispositivo al que se envía el urb.
- ep: De uso interno. Puntero a la estructura endpoint a la cual se envía el URB.
- pipe: Información del pipe. El mismo esta formado de la siguiente manera:

```
direction:      bit 7      0 = Host-to-Device [Out],
  1 = Device-to-Host [In]
device address: bits 8-14  bit positions
  known to uhci-hcd
endpoint:       bits 15-18 bit positions
  known to uhci-hcd
```

```
pipe type:      bits 30-31 00 = isochronous,
  01 = interrupt
  10 = control, 11 = bulk
```

- status: Estatus del urb. El único momento en que se puede acceder a esta variable de forma segura es en la función de complete.
  Toma los valores:
  - 0: transferencia exitosa.
  - ENOENT: detenido por usb_kill_urb.
  - ECONNRESET: desligado por llamada a usb_unlink_urb, el atributo transfer_flags se inicializa en URB_ASYNC_UNLINK.
  - EINPROGRESS: siendo porcesado por el host controller.
  - EPROTO: se dio un error bitstuff durante la transferencia o el paquete de respuesta no se recibió a tiempo.
  - EILSEQ: CRC en transferencia del urb.
  - EPIPE: endpoint estancado. Si no es endpoint de control puede limpiarse con usb_clear_halt.
  - ECOMM: los datos se recibieron más rápido de lo que pueden escribirse. Solo para IN.
  - ENOSR: no puede proveerse la información desde la memoria del sistema a la velocidad necesaria para cumplir con la tasa requerida.
  - EOVERFLOW: probable error, el endpoint recibe paquetes mayores al máximo tamaño definido.
  - EREMOTEIO : solo si URB_SHORT_NOT_OK es asignada a la transfer_flags del urb's y significa que la cantidad total de datos solicitados en el URB no fue recibida.
  - ENODEV: el dispositivo USB desapareció del sistema.
  - EXDEV: solo en isochronous urb, la transferencia fue completa solo de forma parcial.
  - EINVAL: indica un problema con el urb. "ISO madness, if this happens: Log off and go home." Puede pasar al dar un valor erroneo a las variables del urb.
  - ESHUTDOWN: error severo con el USB host controller driver; esta deshabilitado o el dispositivo fue desconectado. Puede suceder al cambiar la configuración para el dispositivo mientras el URB es enviado

  Generalmente -EPROTO, -EILSEQ, y -EOVERFLOW indican problemas de HW, firmware o cable del dispositivo.

- transfer_flags: mapa de bits donde cada bit toma distintos valores según lo que el driver necesite que suceda.
  - URB_SHORT_NOT_OK: especifica que cualquier lectura corta en un endpoint IN debe tratarse como error.

- URB_ISO_ASAP: en transferencias isochronous indica que el URB debe ser despachado tan pronto como la utilización de ancho de banda lo permita y que se asigne la variable start_frame en el URB en este punto.
- URB_NO_TRANSFER_DMA_MAP: indica que el URB contiene un buffer DMA para ser transmitido para que el USB Core utilice el buffer apuntado por transfer_dma en lugar del apuntado por transfer_buffer.
- URB_NO_SETUP_DMA_MAP, URB_NO_TRANSFER_DMA_MAP: indica que el URB de control tienen un buffer DMA, para que el USB Core use el buffer apuntado por setup_dma en lugar del apuntado por setup_packet.
- URB_ASYNC_UNLINK: indica que la llamada a usb_unlink_urb para este URB debe retornar inmediatamente y el URB se deslinkea en segundo plano. En caso contrario la función espera hasta terminar el deslinkeo del URB antes de retornar.
- URB_NO_FSBR: Usado solo en USB Host controller drivers UHCI. Indica que no se intente de hacer de forma lógica un FSB (Front Side Bus Reclamation).
- URB_ZERO_PACKET: indica que los URB BULK OUT terminen enviando un paquete corto sin datos.
- URB_NO_INTERRUPT: indica que el hardware no debe generar una interrupción cuando se culmina con el urb. Solo debe usarse cuando se encolan varios URBs en un endpoint. El USB Core usa esta modalidad para transferencias de buffer DMA.

- transfer_buffer: buffer de datos asociado al URB.
- transfer_dma: dirección para el transfer_buffer usado en transferencias DMA.
- sg: lista de buffers Scatter/gather usado para transferencias DMA.
- num_sgs: cantidad de entradas en sg.
- transfer_buffer_length: largo del buffer apuntado por transfer_buffer o transfer_dma.
- actual_length: largo de la transferencia actual, asignado al final de la misma.
- setup_packet: paquete setup, solo para trasferencias control. Se mapea con la estructura usb_ctrlrequest.
- setup_dma: dirección del setup_packet para transferencias de control hechas con dma.
- start_frame: número del frame inicial de las transferencias isochronous a usar.
- number_of_packets: especifica el número de paquetes de transferencia isochronous para manejar el urb.

- interval: intervalo de polling para URBs isochronous o de interrupción. Depende de la velocidad del dispositivo.
- error_count: asignado por el USB Core en URBs isochronous. Especifica la cantidad de transferencias erróneas.
- context: puntero a datos de contexto que pueden ser asignados por el driver y utilizados en la función de completación.
- complete: puntero a la rutina de completación definida por el driver y que es llamada por el USB Core cuando el URB es completamente transferido o cuando se produce un error. Dentro de esta función el driver puede inspeccionar el urb, liberarlo, o volver a enviarlo en otra transferencia. Se define como: typedef void (* usb_complete_t) (struct URB *, struct pt_regs *);
- iso_frame_desc[0]: Válido para URBs isochronous. Es un arreglo de struct usb_iso_packet_descriptor que componen el urb. Esta estructura se compone de los siguientes campos:

```
unsigned int offset
unsigned int length
unsigned int actual_length
unsigned int status
```

D.6. usb_ctrlrequest

Estructura de pedidos de control con la que se mapea el campo setup_packet de los urb.

```
struct usb_ctrlrequest {
  __u8 bRequestType;
  __u8 bRequest;
  __le16 wValue;
  __le16 wIndex;
  __le16 wLength;
} __attribute__ ((packed));
```

Campos:

- bRequestType: Tipo de pedido. Es un mapa de bits con el siguiente significado:
  - bit 7: Dirección de la transferencia. 0 = Host-to-device, 1 = Device-to-host
  - bits 6-5: Tipo del pedido. 0 = Standard, 1 = Class, 2 = Vendor, 3 = Reserved
  - bits 4-0: Destinatario. 0 = Device, 1 = Interface, 2 = Endpoint, 3 = Other, 4...31 = Reserved
- bRequest: Especifica el pedido.
- wValue: Parámetro que varía según el pedido.

- wIndex: Parámetro que varía según el pedido típicamente usado para pasar un índice o desplazamiento.
- wLength: Número de bytes a transferir si es una operación de transferencia de datos.

Solicitudes estándar					
bRequestType	**bRequest**	**wValue**	**wIndex**	**wLength**	**Data**
00000000B 00000001B 00000010B	CLEAR_FEATURE	Feature Selector	0 Interface Endpoint	0	No
10000000B	GET_CONFIGURATION	0	0	1	Configuración
10000000B	GET_DESCRIPTOR	Tipo de descriptor e	0 ó id de lenguaje	Largo de descriptor	Descriptor
10000001B	GET_INTERFACE	0	Interfaz	1	Interfaz
10000000B 10000001B 10000010B	GET_STATUS	0 Interfaz Endpoint	0	2	Dispositivo interfaz o endpoint
00000000B	SET_ADDRESS	Dirección	0	0	
00000000B	SET_CONFIGURATION	Configuración	0	0	No
00000000B	SET_DESCRIPTOR	tipo e índice de descriptor	0 ó id de lenguaje	largo de descriptor	Descriptor
00000000B 00000001B 00000010B	SET_FEATURE	feature	0 interfaz endpoint	0	No
00000001B	SET_INTERFACE	configuración	interfaz alternativa	0	No
00000010B	SYNCH_FRAME	0	Endpoint	2	Número de frame

Valores de bRequest	
bRequest	**valor**
GET_STATUS	0
CLEAR_FEATURE	1
Reserved for future use	2
SET_FEATURE	3
Reserved for future use	4
SET_ADDRESS	5
GET_DESCRIPTOR	6
SET_DESCRIPTOR	7
GET_CONFIGURATION	8
SET_CONFIGURATION	9
GET_INTERFACE	10
SET_INTERFACE	11
SYNCH_FRAME	12

Valores de tipos de descriptores	
tipo de descriptor	**valor**
DEVICE	1
CONFIGURATION	2
STRING	3
INTERFACE	4
ENDPOINT	5
DEVICE_QUALIFIER	6
OTHER_SPEED_CONFIGURATION	7
INTERFACE_POWER	8

Apéndice E

Pruebas de prototipo USB

E.1. Filtrado de URBs

E.1.1. Primer conjunto de pruebas

Objetivo

Realizar unlink y kill de los URBs enviados hacia un endpoint bulk con dirección in para intenta simular el filtrado de paquetes. Las pruebas se fueron realizadas sobre un pendrive kingston de 4GB. Este tiene 2 endpoints bulk, el endpoint 1 es in y el endpoint 2 es out. Las pruebas fueron realizadas copiando pequeños archivos desde y hacia el dispositivo. En las pruebas sobre endpoint out se copia hacia el dispositivo y en las pruebas sobre endpoint in se copia hacia el host.

Prueba 1:

Realizar el unlink del URB en el submit de URBs enviados a un endpoint bulk in.

Prueba 2:

Realizar el unlink en el complete de URBs enviados a un endpoint bulk in.

Prueba 3:

Realizar el unlink en el submit de URBs enviados a un endpoint bulk out.

Prueba 4:

Realizar el unlink en el complete de URBs enviados a un endpoint bulk out.

Resultados pruebas 1 a 4:

Se copió el archivo de forma correcta por lo que el intento de inyección no tuvo ningún efecto.

Prueba 5:

Realizar el kill del URB en el submit de URBs enviados a un endpoint bulk in.

Prueba 6:

Realizar el kill en el complete de URBs enviados a un endpoint bulk in.

Prueba 7:

Realizar el kill en el submit de URBs enviados a un endpoint bulk out.

Prueba 8:

Realizar el kill en el complete de URBs enviados a un endpoint bulk out.

Resultados pruebas 2 a 8

En las pruebas de kill no es posible copiar el archivo y todos los dispositivos conectados al mismo bus en el cual esta conectado el dispositivo bajo prueba dejan de funcionar.

E.1.2. Segundo conjunto de pruebas

Objetivo

Cambiar dirección de endpoint al que se envía el URB para intenta simular el filtrado de paquetes.

Prueba 1:

Modificar en el submit la dirección del endpoint del URB. Esta dirección esta contenida en un byte formado de la siguiente manera: del bit 0 al 3 se guarda la dirección del endpoint, del 4 al 6 bits reservados y el bit 7 es la dirección (1 si es in)

El cambio que se hizo en la dirección corresponde al endpoint al cual se envía el URB. Para ello se hace un and con el valor 0xF0 y luego un or con 0x0F. De esta manera los URBs deberían ser enviados al endpoint 15 que no está presente en el dispositivo. La prueba consiste en cambiar la dirección de endpoint en el submit de URBs enviados a un endpoint bulk in, y la dirección de los endpoints en el submit de URBs enviados a un endpoint bulk out.

Resultados

Al cambiar la dirección del endpoint en el URB, para todas las transferencias subsecuentes el URB permanece con el número de endpoint igual a 15. Sin embargo las copias de archivos se realizan de forma exitosa.

Prueba 2:

Modificar en el submit el pipe contenido en el URB. El pipe es un atributo del URB con los datos del pipe al cual está asignado. De este es que se obtiene el endpoint en el submit de los URBs.

Dada la descripción del pipe para cambiar la dirección del endpoint del URB desde el pipe se realiza la operación: pipe = pipe & 0xFFF83FFF — (FFF00FFF — nueva_dir 15)

Se prueba copiar hacia el host y cambiar pipe de URB in y copiar hacia el dispositivo y cambiar pipe de URB out. Además las pruebas se realizan cambiando el número de endpoint por uno inexistente y luego por uno existente en el dispositivo.

Resultados

Al copiar hacia el host y cambiar en el pipe de URB in el numero de endpoint se produce un error de copia. Un log del sistema obtenido con estas pruebas es el siguiente:

```
kernel: USB 5-7: reset high speed USB device using ehci_hcd and address 9
kernel: sd 7:0:0:0: [sdb] Write Protect is off
kernel: sd 7:0:0:0: [sdb] Mode Sense: 00 00 00 00
kernel: sd 7:0:0:0: [sdb] Assuming drive cache: write through
kernel: USB 5-7: reset high speed USB device using ehci_hcd and address 9
kernel: sd 7:0:0:0: [sdb] 7942144 512-byte hardware sectors: (4.06GB/3.78GiB)
kernel: sd 7:0:0:0: [sdb] Write Protect is off
kernel: sd 7:0:0:0: [sdb] Mode Sense: 23 00 00 00
kernel: sd 7:0:0:0: [sdb] Assuming drive cache: write through
```

Al copiar hacia el dispositivo y cambiar pipe de URB out se cancela la copia. El archivo no pudo ser copiado. En algunas ocasiones el sistema incluso desmonta el dispositivo de forma automática y no pudo volver a ser montado. Un log del sistema de esta prueba es:

```
kernel: USB 5-8: reset high speed USB device using ehci_hcd and address 8
kernel: sd 5:0:0:0: Device offlined - not ready after error recovery
kernel: sd 5:0:0:0: rejecting I/O to offline device
kernel: sd 5:0:0:0: [sdb] Result: hostbyte=DID_NO_CONNECT
    driverbyte=DRIVER_OK,SUGGEST_OK
kernel: end_request: I/O error, dev sdb, sector 963
kernel: FAT: FAT read failed (blocknr 35)
kernel: sd 5:0:0:0: rejecting I/O to offline device
kernel: __ratelimit: 511 callbacks suppressed
kernel: Buffer I/O error on device sdb1, logical block 15512
kernel: lost page write due to I/O error on sdb1
hald: unmounted /dev/sdb1 from '/media/ALVARO' on behalf of uid 0
```

E.2. Modificación de datos transmitidos

E.2.1. Primer conjunto de pruebas

Objetivo

El objetivo de estas pruebas es modificar los datos dentro de los URBs transmitidos en una transferencia interrupt.

Se intenta cambiar los datos enviados desde un mouse relacionados con la posición del puntero y que botones están siendo presionados. En las transferencias interrupt se envían URBs de polling desde el host hacia el dispositivos. Al retornar estos en el complete, se obtienen los datos enviados por el dispositivo. Por ello el punto para realizar el cambio de los datos recuperados del dispositivo debe ser en el complete del módulo creado para las pruebas de inyección sobre USB.

Para el mouse la información es retornada en el campo transfer_buffer del URB. El primer byte contiene la información de que botones están siendo presionados, el segundo y tercer byte la de la posición relativa del mouse en complemento a dos.

Prueba 1:

Borrar toda la información del buffer asignando 0 a todos los valores.

Resultados:

Mientras el módulo se encuentra levantado el mouse no responde ante movimientos o al presionar sus botones.

Prueba 2:

Borrar la información sobre la posición relativa a X asignando 0 al segundo byte.

Resultados:

Al realizar la segunda prueba, mientras el módulo se encuentra levantado el mouse no responde al presionar sus botones, pero si al movimiento.

Prueba 3:

Borrar la información sobre que botones han sido presionados (mediante un and entre el primer byte y 0xF4 = 11111000B)

Resultados:

Al realizar la tercera prueba, mientras el módulo se encuentra levantado el mouse no responde a movimientos en el eje de las X, moviéndose solo en dirección vertical. Si responde al presionar los botones.

E.2.2. Segundo conjunto de pruebas

Objetivo

Modificar el parámetro interval de los URBs transmitidos en una transferencia interrupt. Este tipo de transferencias son periódicas, y suceden en intervalos en unidades que son potencia de dos. Para dispositivos full y low speed una unidad es un mili-segundo y para high speed son microsegundos. La llamada a usb_submit_urb modifica el parámetro interval por uno que es menor o igual al intervalo solicitado.

Por ello el punto para realizar esta prueba debe cambiarse el valor del parámetro interval en la función submit del módulo creado para las pruebas de inyección sobre USB.

Prueba 1:

Cambiar el valor interval de los URBs enviados hacia el mouse. Los valores probados fueron 64, 32, 16, 8, 4, 2, 1 y 0.

Resultados:

Al comenzar las pruebas el valor que tienen los URBs es 8. Cuando se intenta cambiar dicho valor por uno mayor no hay efecto, y al capturar en el complete el valor presente en dicho campo es igual al que tiene antes de cambiarlo.

Al cambiar el valor por uno menor este es cambiado con éxito, capturando en el complete el valor asignado. Sin embargo no se observa ningún efecto. Esto es coherente si pensamos que una mayor frecuencia de interrupción no haría más que obtener más seguido los datos de la posición del cursor y esto difícilmente afectaría la percepción que se tiene del movimiento del mismo, a diferencia de lo que sería poder bajar la frecuencia.

Entre pruebas si el valor del interrupt se cambia por uno menor, al comenzar la siguiente prueba el valor se mantiene en el último cambiado. Este vuelve al valor inicial 8 solo después de desconectar y volver a conectar el dispositivo.

E.3. Cambio de estado en complete

E.3.1. Primer conjunto de pruebas

Objetivo

Realiza una inyección cambiando el campo status en un URB de salia (tipo OUT) para el tipo de transferencia BULK utilizando un pendrive.

Prueba 1:

Copiar un archivo desde el host hacia el pendrive y modificar el campo status de un URB utilizando el callback del mismo, posteriormente se restaura el callback anterior de forma tal de no intervenir en las tareas que este realiza. La inyección es realizada a un URB que no sea de tipo CBW (Command Block Wrapper) de forma que la misma se realice en la transferencia de datos hacia el dispositivo. El valor inyectado en el campo status es el -EILSEQ cuyo significado es el de que ha ocurrido un error de CRC.

Resultados:

Al realizar esta inyección se puede observar que el archivo se ha copiado correctamente, es decir la inyección no ha tenido ningún efecto.

Ejecutando mount usbfs & cat /proc/bus/usb/devices se observa que el driver que se esta utilizando para este dispositivo se llama usb-storage

```
T:  Bus=05 Lev=01 Prnt=01 Port=02 Cnt=01 Dev#=  2 Spd=12  MxCh= 0
D:  Ver= 1.10 Cls=00(>ifc ) Sub=00 Prot=00 MxPS= 8 #Cfgs=  1
P:  Vendor=10d6 ProdID=1100 Rev= 1.00
S:  Product=USB MASS STORAGE CLASS
S:  SerialNumber=USB MASS STORAGE CLASS
C:* #Ifs= 1 Cfg#= 1 Atr=80 MxPwr=100mA
I:* If#= 0 Alt= 0 #EPs= 2 Cls=08(stor.) Sub=05 Prot=50 Driver=usb-storage
E:  Ad=01(O) Atr=02(Bulk) MxPS=  64 Ivl=0ms
E:  Ad=82(I) Atr=02(Bulk) MxPS=  64 Ivl=0ms
```

Luego buscando en los fuentes del kernel que driver tiene ese nombre se encuentra el callback que se utiliza en este driver.

```
/* This is the completion handler which will wake us up when
* an URB completes.
*/
static void usb_stor_blocking_completion(struct URB *urb)
{
    struct completion *urb_done_ptr = urb->context;
    complete(urb_done_ptr);
}

void complete(struct completion *x)
{
    unsigned long flags;
    spin_lock_irqsave(&x->wait.lock, flags);
    x->done++;
    __wake_up_common(&x->wait, TASK_NORMAL, 1, 0, NULL);
    spin_unlock_irqrestore(&x->wait.lock, flags);
}
EXPORT_SYMBOL(complete);
```

En el código se puede apreciar que el driver en este caso no toma ninguna medida para el valor devuelto en el status del URB. Si bien en este caso la transferencia fue exitosa ya que el URB no tenia errores de CRC, con la inyección realizada se pone en evidencia fácilmente que el driver no maneja este tipo de situaciones.

E.3.2. Segundo conjunto de pruebas

Objetivo

Realiza una inyección cambiando el campo status en un URB de entrada (tipo IN) para el tipo de transferencia INTERRUPT utilizando un mouse.

Prueba1:

Para el mouse el funcionamiento de envíos y recepción de los URBs es el siguiente: cuando el drivers quiere conocer las coordenadas de la nueva posición envía un URB con tipo de transferencia interrupt, y en caso de que se conozcan nuevas posiciones para el mismo se envía este URB como respuesta. Para el caso de que el mouse este inmóvil, el URB queda a la espera de nuevas coordenadas y el drivers no vuelve a interrogar al dispositivo hasta que se le de una respuesta.

Por lo tanto si bien por ser un tipo de transferencia por interrupción se espera que se realice un polling al dispositivo, el mismo se realiza a "demanda" según se va necesitando.

Por este motivo, al cambiar el callback de un URB en el submit del módulo, si el mouse esta inactivo y se intenta bajar el mismo con insmod, el mouse se "cuelga" al querer invocar el callback del URB que estaba esperando por su nueva posición, ya que el módulo no esta disponible y por ende la operación callback que se sustituye para de este modo realizar la inyección.

Vale la pena mencionar que esta es la única forma de realizar la inyección por este método (interceptando un complete), ya que de utilizar la operación complete que ofrece la estructura usb_mon_operations los cambios no son tenidos en cuenta porque luego que en el USB Core se retorna de la misma se restaura el status de la siguiente forma:

```
void usb_hcd_giveback_urb(struct usb_hcd *hcd, struct URB *urb,
    int status)
{
    urb->hcpriv = NULL;
    if (unlikely(urb->unlinked)) status = urb->unlinked;
    else if (unlikely((urb->transfer_flags & URB_SHORT_NOT_OK) &&
      urb->actual_length < urb->transfer_buffer_length && !status))
    status = -EREMOTEIO;

    unmap_urb_for_dma(hcd, urb);
    //invoco complete del modulo
    USBMon_urb_complete(&hcd->self, urb, status);
    usb_unanchor_urb(urb);

    /* pass ownership to the completion handler */
    //se restaura status antes de devolver el urb
    urb->status = status;
    urb->complete (urb);
    atomic_dec (&urb->use_count);
    if (unlikely(atomic_read(&urb->reject)))
      wake_up (&usb_kill_urb_queue);
    usb_put_urb (urb);
}
```

Como solución a este inconveniente se puede intentar postergar la descarga del módulo hasta que ya no hay un URB inyectado. Otra alternativa es la de dejar de utilizar el dispositivo cuando se terminen de realizar las inyecciones correspondientes.[1]

[1]En el módulo de inyección implementado en el producto final se realiza el cambio de callback en la función complete definida en usb_mon_operations y se guarda en el contexto

Supuestos:

Para que esta prueba en particular funcione correctamente el driver deberá utilizar una única función complete para los URBs que van hacia el endpoint a inyectar.Si bien eventualmente un driver podría utilizar distintos callback para los URBs, de todos los drivers vistos para las pruebas ninguno de ellos utilizaba más de uno.

Resultados:

Al cambiar el status por el valor `-EILSEQ` se obtiene el resultado esperado, es decir, el URB es descartado y se vuelve a consultar al dispositivo por sus coordenadas.

Por lo tanto, a medida que se va aumentando la frecuencia de perdidas de URBs se puede ver como el mouse se enlentece y en el momento de dejar de realizar inyecciones la velocidad del mouse vuelve a ser la habitual.

Al probar con otros valores para el estado, como con `-ENOENT`, que significa que el core ha descartado el URB por algún motivo, se esperaría que suceda lo mismo, pero en este caso por motivos que dependen del driver o el USB Core el resultado no es el esperado; en cambio el dispositivo deja de funcionar provocando en la mayoría de los casos que dejen de funcionar todos los puertos USB hasta que se reinicie el PC.

E.3.3. Tercer conjunto de pruebas

Objetivo

Realiza una inyección cambiando el campo status en un URB de entrada (tipo IN) para el tipo de transferencia ISOCH utilizando una cámara.

Prueba1:

Para esta prueba se utiliza una cámara web y el visor wxcam. La prueba consistió en cambiar el status de un URB enviado al endpoint ISOCHRONOUS de la cámara por el valor -EILSEQ. Para poder apreciar mejor el efecto del experimento se dejo pasar los primero 2000 URBs enviado y se inyectó el error en el 2001.

Resultados:

Cuando se levanta el visor wxcam se comienza a recibir la imagen de forma correcta. Luego de pasados el número de URBs indicado antes al realizar la inyección, la imagen se congela y en la consola donde se ejecuta el visor muestra el siguiente mensaje de error.

```
VIDIOC_DQBUF: Error de entrada/salida
```

En el log del sistema se muestra el siguiente error:

del URB una estructura en la cual se respalda el puntero a la función complete original y el contexto del URB. Esto permite realizar en la función de callback la inyección y restaurar el contexto del URB para luego llamar al complete original. Este mecanismo resuelve las limitaciones que presenta el prototipo.

```
kernel: uvcvideo: Non-zero status (-84) in video completion handler.
```

De esta manera es que al recibir URBs con error la transferencia de vídeo fue detenida. Un comportamiento aceptable en una transferencia ISOCHRONOUS de vídeo puede ser que ante un URB con defectos se descarte este paquete y se siga transmitiendo, pero esto depende del manejo de errores que haga el driver o la propia aplicación visor.

Al observar el complete del driver de la cámara vemos el manejo del error que este realiza.

```
static void uvc_video_complete(struct URB *urb)
{
    struct uvc_video_device *video = urb->context;
    struct uvc_video_queue *queue = &video->queue;
    struct uvc_buffer *buf = NULL;
    unsigned long flags;
    int ret;

    switch (urb->status) {
    case 0:
        break;
    default:
        uvc_printk(KERN_WARNING,"Non-zero status (%d) in video "
                  "completion handler.\n", urb->status);
    case -ENOENT:     /* usb_kill_urb() called. */
        if (video->frozen)
            return;
    case -ECONNRESET:/* usb_unlink_urb() called. */
    case -ESHUTDOWN: /* The endpoint is being disabled. */
        uvc_queue_cancel(queue, urb->status == -ESHUTDOWN);
        return;
    }

    spin_lock_irqsave(&queue->irqlock, flags);
    if (!list_empty(&queue->irqqueue))
        buf = list_first_entry(&queue->irqqueue, struct uvc_buffer,
              queue);
    spin_unlock_irqrestore(&queue->irqlock, flags);

    video->decode(urb, video, buf);

    if ((ret = usb_submit_urb(urb, GFP_ATOMIC)) < 0) {
        uvc_printk(KERN_ERR,
        "Failed to resubmit video URB (%d).\n", ret);
    }
}
```

E.4. Modificación de datos en la enumeración

En este conjunto de pruebas se realizan inyecciones de defectos durante el proceso de enumeración, las opciones de inyección que se presentan en esta etapa son muy amplias, por lo tanto las pruebas realizadas no son más que un intento de mostrar la factibilidad de realizar dichas inyecciones.

E.4.1. Primer conjunto de pruebas

Objetivo

El objetivo de las pruebas es modificar los datos de configuración que obtiene el USB Core de los dispositivos. Esto se logra capturando los URBs enviados al endpoint 0 en la enumeración del dispositivo y modificando la información contenida en los mismos.

Prueba1:

La prueba consiste en modificar la dirección del endpoint por otra dirección que no exista. Para esta primer prueba se utiliza un mouse para realizar la inyección, el mismo dispone de un único endpoint IN de tipo INTERRPUT. El standard USB define que cada dispositivo puede tener como máximo 16 endpoint IN y 16 endpoint OUT, que se direccionan con un entero entre 0 y 15. Esta información se encuentra en el campo bEndpointAddress del descriptor del endpoint que fue explicado con la estructura usb_endpoint_descriptor.

Para el caso del dispositivo utilizado el valor de dicho campo es 0x81 = 10000001b por lo tanto se trata de un endpoint IN cuya dirección es 1. El defecto inyecta el valor 0x8F = 10001111b, que indica que es un endpoint IN cuya dirección es 15.

Resultado:

En el momento que se inserta el mouse en un puerto USB y comienza el proceso de enumeración, se puede observar que al pasar el tiempo este no queda disponible para ser utilizado por el usuario. Inspeccionando el log que deja el módulo de inyección se puede observar que el USB Core sigue solicitando el descriptor de la configuración (USB_DT_CONFIG) ininterrumpidamente (pues de ahí saca el descriptor del endpoint). Mientras en cada reintento de la petición al descriptor se realice esta inyección el USB Core sigue pidiendo el descriptor.

La salida de lsusb -v se en ese momento indica lo siguiente

```
Endpoint Descriptor:
bLength          7
bDescriptorType  5
bEndpointAddress 0x8f  EP 1 IN
bmAttributes     3
    Transfer Type  Interrupt
    Synch Type     None
    Usage Type     Data
wMaxPacketSize   0x0006  1x 6 bytes
bInterval        10
```

Con esto vemos que la inyección fue realizada con éxito. Por lo tanto se concluye que mediante algún mecanismo USB Core se da cuenta que esto no es correcto y por eso intenta obtenerlo nuevamente. En el momento que se deja de hacer la inyección, al pedir nuevamente el descriptor obtiene el correcto y el mouse comienza a funcionar correctamente.

Prueba2:

Para esta prueba se utiliza el mismo mouse que en la prueba anterior. La prueba consiste en modificar el tipo de transferencia del endpoint de este dispositivo (tipo INTERRUPT) por el tipo de transferencia BULK. Esta información se encuentra en el campo bmAttributes del descriptor del endpoint. La composición de este campo se explicó junto con la estructura usb_endpoint_descriptor.

Los dos bits menos significativos indican el tipo de transferencia, por lo tanto para inyectar el defecto se debe sustituir los dos bits menos significativos por el valor 10b (correspondiente a transferencia BULK) del campo anteriormente mencionado.

Resultado:

Si bien la inyección se realiza correctamente, y al visualizar la salida de lsusb -v para ese dispositivo se puede observar el resultado de la inyección, el mouse funciona correctamente y no presenta ningún inconveniente.

Fragmento de la salida de lsusb -v para el endpoint inyectado.

```
Endpoint Descriptor:
bLength           7
bDescriptorType   5
bEndpointAddress  0x81  EP 1 IN
bmAttributes      2
    Transfer Type   Bulk
    Synch Type      None
    Usage Type      Data
wMaxPacketSize    0x0006  1x 6 bytes
bInterval         10
```

Por otra parte, si se ejecuta cat /proc/bus/usb/devices se puede observar más información con respecto al mouse.

```
T:  Bus=06 Lev=01 Prnt=01 Port=00 Cnt=01 Dev#=  4 Spd=1.5 MxCh= 0
D:  Ver= 2.00 Cls=00(>ifc ) Sub=00 Prot=00 MxPS= 8 #Cfgs=  1
P:  Vendor=1bcf ProdID=0007 Rev= 0.10
S:  Product=USB Optical Mouse
C:* #Ifs= 1 Cfg#= 1 Atr=a0 MxPwr= 98mA
I:* If#= 0 Alt= 0 #EPs= 1 Cls=03(HID  ) Sub=01 Prot=02 Driver=usbhid
E:  Ad=81(I) Atr=03(Int.) MxPS=   6 Ivl=1ms
```

En la última linea se puede observar que se indica información sobre el atributo bmAttributes, esta información se indica mediante Atr=03(Int.). Esto es incoherente con la información que lista lsusb y la inyección realizada.

De todas formas, revisando el funcionamiento del driver (usbhid) se puede observar que una vez asignado al dispositivo, este comienza a enviar URBs del tipo INTERRUPT sin consultar de que tipo es el endpoint del mouse. Probablemente a este hecho se deba el buen funcionamiento del dispositivo.

Prueba3:

Para esta prueba se utiliza el mismo mouse que en la prueba anterior. La prueba consiste en modificar el intervalo con que el driver debe consultar el dispositivo para obtener nuevos datos. El standard USB define bInterval del siguiente modo,

```
bInterval
```

```
Interval for polling endpoint for data transfers.
Expressed in frames or microframes depending on the
device operating speed (i.e., either 1 millisecond or 125 units).

For full-/high-speed isochronous endpoints, this value
must be in the range from 1 to 16. The bInterval value
is used as the exponent for a 2^bInterval-1 value; e.g., a
bInterval of 4 means a period of 8 (2^4-1).
For full-/low-speed interrupt endpoints, the value of
this field may be from 1 to 255.

For high-speed interrupt endpoints, the bInterval value
is used as the exponent for a 2^bInterval-1 value; e.g., a
bInterval of 4 means a period of 8 (2^4-1). This value
must be from 1 to 16.

For high-speed bulk/control OUT endpoints, the
bInterval must specify the maximum NAK rate of the
endpoint. A value of 0 indicates the endpoint never
NAKs. Other values indicate at most 1 NAK each
bInterval number of microframes. This value must be
in the range from 0 to 255.
```

Para el caso del dispositivo utilizado el valor de dicho campo es 10 ms. El defecto inyecta el valor 255 indicando que se debe realizar el polling más lentamente.

Ejecutando lsusb -v se puede observar la información correspondiente al mouse (aquí solo mostramos el fragmento que corresponde al endpoint inyectado).

```
Endpoint Descriptor:
bLength           7
bDescriptorType   5
bEndpointAddress  0x81  EP 1 IN
bmAttributes      3
    Transfer Type   Interrupt
    Synch Type      None
    Usage Type      Data
wMaxPacketSize    0x0006  1x 6 bytes
bInterval         255
```

Resultado:

Si bien se puede observar un pequeño comportamiento inusual al mover el puntero del mouse, este no es significativo comparándolo con lo observado al mover el puntero del mouse sin realizar la inyección. Un posible motivo de este comportamiento podría ser que con refrescar la posición del mouse cada 255 ms sea suficiente para el mismo y por lo tanto no se percibe una mayor lentitud.

Prueba4:

En esta última prueba y utilizando el mismo mouse que en las pruebas anteriores se intenta cambiar la clase de la interfaz del dispositivo. Esta información se encuentra en el campo bInterfaceClass de cada descriptor de interfaz de todos los dispositivos.

Observando la salida del comando lsusb -v para el mouse en cuestión podemos ver que el valor del campo mencionado es 0x03, este valor corresponde a un dispositivo HID (Human Interface Device).

La lista completa se puede encontrar en el archivo ch9.h encontrado en el kernel de linux.

```
/*
* Device and/or Interface Class codes
* as found in bDeviceClass or bInterfaceClass
* and defined by www.usb.org documents
*/
#define USB_CLASS_PER_INTERFACE 0 /* for DeviceClass */
#define USB_CLASS_AUDIO        1
#define USB_CLASS_COMM         2
#define USB_CLASS_HID          3
#define USB_CLASS_PHYSICAL     5
#define USB_CLASS_STILL_IMAGE  6
#define USB_CLASS_PRINTER      7
#define USB_CLASS_MASS_STORAGE 8
#define USB_CLASS_HUB          9
#define USB_CLASS_CDC_DATA     0x0a
#define USB_CLASS_CSCID        0x0b /* chip+ smart card */
#define USB_CLASS_CONTENT_SEC  0x0d /* content security */
#define USB_CLASS_VIDEO        0x0e
#define USB_CLASS_WIRELESS_CONTROLLER 0xe0
#define USB_CLASS_MISC         0xef
#define USB_CLASS_APP_SPEC     0xfe
#define USB_CLASS_VENDOR_SPEC  0xff
#define USB_SUBCLASS_VENDOR_SPEC 0xff
```

El valor del defecto inyectado es el 8, correspondiente a Class Mass Storage.

Resultado:

Ejecutando lsusb -v se puede observar la información correspondiente al mouse (aquí solo mostramos el fragmento que corresponde a la interfaz inyectada).

```
Interface Descriptor:
    bLength           9
    bDescriptorType   4
    bInterfaceNumber  0
    bAlternateSetting 0
    bNumEndpoints     1
    bInterfaceClass   8 Mass Storage
    bInterfaceSubClass 1 RBC (typically Flash)
    bInterfaceProtocol 2
    iInterface        0
    ** UNRECOGNIZED:  09 21 10 01 00 01 22 57 00
    Endpoint Descriptor:
      bLength         7
      bDescriptorType 5
      bEndpointAddress 0x81  EP 1 IN
      bmAttributes    3
    Transfer Type     Interrupt
    Synch Type        None
    Usage Type        Data
        wMaxPacketSize  0x0006  1x 6 bytes
        bInterval     10
    Device Status:    0x0000
```

```
(Bus Powered)
```

Podemos ver que el campo bInterfaceClass contiene el valor 8 y 05:03 p.m. 28/05/2010

Resultado:

Una vez insertado el dispositivo y culminado el proceso de enumeración, se observa que el mouse no responde y no esta disponible para ser utilizado.

Al Inspeccionar la información proporcionada por usbfs, al ejecutar cat /proc/bus/usb/devices se observa la siguiente información.

```
T:  Bus=06 Lev=01 Prnt=01 Port=00 Cnt=01 Dev#=  6 Spd=1.5 MxCh= 0
D:  Ver= 2.00 Cls=00(>ifc ) Sub=00 Prot=00 MxPS= 8 #Cfgs=  1
P:  Vendor=1bcf ProdID=0007 Rev= 0.10
S:  Product=USB Optical Mouse
C:* #Ifs= 1 Cfg#= 1 Atr=a0 MxPwr= 98mA
I:* If#= 0 Alt= 0 #EPs= 1 Cls=08(stor.) Sub=01 Prot=02 Driver=(none)
E:  Ad=81(I) Atr=03(Int.) MxPS=   6 Ivl=10ms
```

En la linea correspondiente a I: se puede observar que el valor para el campo inyectado es Cls=08(stor.) y además se puede observar que el motivo por el cual el mouse no responde es que no se le ha asignado ningún driver Driver=(none).

Observando el driver que habitualmente atiende al mouse, es coherente que este no se haga cargo del mismo ya que en su estructura usb_device_id define lo siguiente:

```
static const struct usb_device_id hid_usb_ids[ ] = {
  { .match_flags = USB_DEVICE_ID_MATCH_INT_CLASS,
    .bInterfaceClass = USB_INTERFACE_CLASS_HID },
  { } /* Terminating entry */
};
```

O sea que especifica que entre otras cosas el mouse sea USB_INTERFACE_CLASS_HID, y esto ha sido modificado por la inyección.

Apéndice F

Pruebas USB/IP

Las pruebas de instalación fueron realizadas en varios conjuntos que se explican a continuación.

F.1. Primer conjunto de pruebas

Objetivo:

En un PC se levanta una VM y se intenta compartir entre dicho PC y la VM un dispositivo.

Entorno:

Tanto el PC como la VM tienen openSUSE 11.1 con los módulos que vienen con el SO y los binarios instalados desde el yast. El PC oficia de servidor y la VM de cliente.

Prueba 1:

compartir memoria USB kingston de 4GB siguiendo los pasos antes indicados para compartir el dispositivo.

Resultados:

Esta prueba fue realizada repetidas veces con resultados diferentes. En ocasiones se lograba compartir el dispositivo, este aparecía como disponible en el cliente pudiendo ser utilizado. Posteriormente se procedía a desconectarlo desde el cliente para que el servidor vuelva a tomar control del mismo. Repitiendo los mismos pasos en algunas ocasiones el dispositivo no podía ser retornado al servidor, o no era posible siquiera desconectarlo desde el cliente. En otras pruebas luego de que el cliente solicitaba el dispositivo este no quedaba disponible en el mismo para ser usado, aunque siempre se dio que el dispositivo dejaba de estar disponible en el servidor. En muchas ocasiones además del mal funcionamiento de la aplicación se dio de que todos los dispositivos USB dejaban de funcionar. En ningún caso se desplegó ningún mensaje de error.

Prueba 2:

compartir otros dispositivos, memoria de 16GB y mp4 titan 2GB.

Resultados:

No fue posible compartir ninguno de los dos dispositivos. En el paso en que el cliente reclama dichos dispositivos estos nunca aparecen como disponibles, aunque si se da que dejan de estar disponibles en el servidor. En todos los casos no fue posible recuperar los dispositivos en el servidor. En varias ocasiones el mal funcionamiento trajo aparejado la caída de todos los dispositivos USB. En ningún caso se desplegó ningún mensaje de error.

F.2. Segundo conjunto de pruebas

Objetivo:

Compilar los fuentes de los módulos y de los binarios de la aplicación para posteriormente correrlos en un PC y una VM y compartir un dispositivo.

Entorno:

Tanto el PC como la VM tienen openSUSE 11.1. Los fuentes de módulos y binarios a compilar son los bajados desde la página del proyecto USB/IP correspondientes a la versión 0.1.7. El PC oficia de servidor y la VM de cliente.

Prueba 1:

Compilar los módulos a partir de los fuentes bajados.

Resultados:

Para esto se debe compilar la versión correspondiente al kernel de linux. En este punto se comprobó que entre los fuentes descargados había una versión por cada versión del kernel desde la 2.6.16 a 2.6.25 salteándose inclusive algunas versiones. Luego en las notas se aclaraba que se debía compilar la versión correspondiente al kernel y desde la versión 2.6.28 en adelante se debía "probar" con los fuentes que vienen en el linux-staging. La versión del kernel que viene con openSUSE 11.1 es la 2.6.27 por lo que, para esta prueba, se encontraba en un bache que no cubren ni los fuentes bajados ni los que se encuentran en el staging.

En este punto las opciones eran:

1. compilar con los fuentes de otra versión: esto se intento con los distintos fuentes e inclusive con los del staging obteniendo siempre por resultado el error "invalid module format" al intentar levantar los módulos compilados.
2. instalar el SO con una versión anterior o posterior de kernel: se optó por actualizar a la versión de openSUSE 11.2 que viene con el kernel 2.6.31 y utilizar los fuentes de la carpeta staging.

F.3. Tercer conjunto de pruebas

Objetivo:

Compilar los fuentes de los módulos y de los binarios de la aplicación para posteriormente correrlos en un PC y una VM y compartir un dispositivo.

Entorno:

Tanto el PC como la VM tienen openSUSE 11.2. Los fuentes de módulos son los de la carpeta staging del kernel y los binarios a compilar son los bajados desde la página del proyecto USB/IP correspondientes a la versión 0.1.7. El PC oficia de servidor y la VM de cliente.

Prueba 1:

Compilar los módulos a partir de los fuentes del staging.

Resultado:

Se pudieron compilar los módulos correctamente y estos no daban error al correrlos.

Prueba 2:

Compilar los binarios a partir de los fuentes descargados.

Resultado:

Se intentó la compilación en varias ocasiones encontrando como inconvenientes la falta de bibliotecas que eran necesarias por algunas dependencias que no estaban aclaradas en la documentación del proyecto. Para correr los mismos se tuvo como nuevo contratiempo que los comandos fallaban al ser ejecutados debido a que faltó crear un directorio y un enlace que tampoco estaba especificado en la documentación oficial. Subsanados estos inconvenientes se realizó la compilación e instalación de los binarios y estos eran ejecutados sin que lanzaran errores.

Prueba 3:

Con los elementos compilados anteriormente en la PC y la VM compartir la memoria kingston de 4GB (la que pudo ser compartida en el primer conjunto de pruebas).

Resultado:

Se probó en varias ocasiones compartir dicha memoria. Del lado del servidor siempre se logro compartir el recurso y llegar al punto en el que el cliente listaba los dispositivos que el servidor designó como compartidos. Incluso siempre se pudo pasar el punto en el que en el cliente indicaba el dispositivo que se quería obtener, pero en este momento el dispositivo nunca fue montado en el cliente y no era posible acceder al mismo. Tampoco fue posible realizar la liberación del dispositivo para el servidor.

Al hacer detach da el error:

```
usbip --detach 00
8 ports available
usbip err: vhci_driver.c: 19 (imported_device_init)
sysfs_open_device <NULL>
usbip err: vhci_driver.c: 102 (parse_status) init new device
usbip err: vhci_attach.c: 393 (detach_port ) open vhci_driver
```

Luego al intentar recuperar el dispositivo desde el servidor el proceso queda colgado al ejecutar el comando correspondiente.

Prueba 4:

Análoga a la prueba 3 pero con otros dispositivos (memoria de 16GB y mp4 titan 2GB).

Resultado:

Igual al resultado de prueba 3.

F.4. Cuarto conjunto de pruebas

Objetivo:

Con los fuentes de los módulos y de los binarios de la aplicación compilados anteriormente correr las pruebas entre 2 PCs.

Entorno:

Ambas PCs tienen openSUSE 11.2. Los fuentes de módulos son los de la carpeta staging del kernel y los binarios a compilar son los bajados desde la página del proyecto USB/IP correspondientes a la versión 0.1.7. Los PC están conectados a través de ADSL, un PC oficia de servidor y el otro de cliente.

Prueba 1:

Con los elementos indicados anteriormente compartir la memoria kingston de 4GB.

Resultado:

Del lado del servidor se logro compartir el recurso al igual que en el conjunto anterior de pruebas el error se dio en el punto en que el cliente indicaba el dispositivo que se quería obtener, El dispositivo nunca fue montado en el cliente y no fue posible acceder al mismo. Tampoco fue posible realizar la liberación del dispositivo para el servidor. Luego al intentar recuperar el dispositivo desde el servidor el proceso queda colgado al ejecutar la el comando correspondiente.

F.5. Quinto conjunto de de pruebas

Objetivo:

Como en el primer conjunto de pruebas, en un PC se levanta una VM y se intenta compartir entre dicho PC y la VM un dispositivo, con la diferencia de que los módulos son los que vienen con openSUSE 11.2.

Entorno:

Tanto el PC como la VM tienen openSUSE 11.2 con los módulos que vienen con el SO y los binarios instalados desde el yast. El PC oficia de servidor y la VM de cliente.

Prueba 1:

Compartir algún dispositivo siguiendo los pasos antes indicados para compartir el dispositivo. (memoria USB kingston de 4GB, memoria de 16 GB, mp4 titan 2GB)

Resultado:

Se intento en varias ocasiones logrando siempre los mismos resultados que en el conjunto de pruebas 3. El error obtenido fue muy similar:

```
usbip --detach 00
8 ports available
usbip err: vhci_driver.c: 19 (imported_device_init)
sysfs_open_device 001
usbip err: vhci_driver.c: 102 (parse_status) init new device
usbip err: vhci_attach.c: 393 (detach_port ) open vhci_driver
```

Apéndice G

Manual de Producto

En este apéndice se explica la forma de utilizar el producto y los comandos que este acepta.

G.1. Funcionamiento

El principal componente del producto es el modulo FIK. Este es el encargado de la inyección sobre `syscalls` y ofrece la interfaz de usuario. Se ejecuta mediante el comando fik.

- Usos:
  - fik mode [pid]
- Parámetros:
  - mode: obligatorio, indica el modo de ejecución. Vale 0 para Ptrace y 1 para LD_PRELOAD
  - pid: indica el pid del proceso sobre el cual se inyectarán los defectos. Este parámetro solo se toma en cuenta en modo Ptrace.

En cualquiera de los modos de ejecución la aplicación recibe comandos para campañas de inyección sobre USB. A continuación se describen los pasos a seguir para realizar una campaña de inyección en cada uno de los modos de ejecución.

G.1.1. Modo Ptrace

Para comenzar una campaña de inyección sobre `syscalls` utilizando Ptrace en primer lugar se debe ejecutar la aplicación mediante 'fik 0 pid' donde pid es el pid del proceso sobre el que se realizará la inyección.
Lo siguiente es ingresar la especificación de los defectos mediantes los comandos existentes para dicho fin.

G.1.2. Modo LD_PRELOAD

Para comenzar una campaña de inyección sobre operaciones de Glibc utilizando LD_PRELOAD se debe ejecutar la aplicación mediante 'fik 1'. En donde se ejecute la aplicación que será objetivo de la inyección debe inicializarse la variable de entorno LD_PRELOAD con la ruta a la librería con los wrappers de Glibc (wraperWlibc.so). Estas dos tareas no tienen un orden en particular.

Lo siguiente es ingresar la especificación de los defectos mediantes los comandos existentes para dicho fin.

G.1.3. Inyección USB

Para comenzar una campaña de inyección sobre USB es indistinto el modo en que se ejecuta la aplicación. Si es requisito cargar en el kernel el módulo USBFI dado que este es el encargado de llevar a cabo las tareas necesarias para cumplir con la inyección de defectos sobre USB. Estas dos tareas no tienen un orden en particular.

Lo siguiente es ingresar la especificación de los defectos mediantes los comandos existentes para dicho fin.

Es importante aclarar que hasta que no se especifique un dispositivo sobre el cual inyectar (tarea realizada mediante el comando usbdevice) no se ejecutará ninguno de los defectos especificados en la campaña de inyección.

G.2. Comandos

A continuación se presenta la lista de comandos aceptados por la aplicación y su descripción.

G.2.1. Comandos de uso general

exit

- Usos:
  - exit
- Descripción: Para salir de la aplicación (si se carga desde archivo no se toma en cuenta)

-h

- Usos:
  - -h
- Descripción: Despliega la lista de comandos.

archivo

- Usos:
  - archivo <filename>

- Descripción: Carga defectos desde archivo. Los comandos especificados deben tener el mismo formato con el que son ingresados por línea de comandos. Las líneas que comienzan con "# " se toman como comentarios.
- Parámetros:
  - Filename: Nombre del archivo, si tiene espacios debe ingresarse entre "". Los comandos dentro del archivo deben cumplir el mismo formato que se ingresa en línea de comandos.

G.2.2. Defectos para modo LD_PRELOAD

openld

- Usos:
  - openld <when> <allways> <error> [filename]
- Descripción: Defecto sobre función Open de Glibc.
- Parámetros:
  - when: en qué número de invocación a la función se activa el defecto.
  - allways: si el defecto luego de activado debe ejecutarse siempre. Valores 0=false o 1=true.
  - error: código de error retornado.
  - filename: ruta al archivo para el cual se activa el defecto. Si no se ingresa filename se inyecta en cualquier archivo. Si filename tiene espacios debe ingresarse entre "".

fopenld

- Usos:
  - fopenld <when> <allways> <error> [filename]
- Descripción: Defecto sobre función Fopen de Glibc.
- Parámetros:
  - when: en qué número de invocación a la función se activa el defecto.
  - allways: si el defecto luego de activado debe ejecutarse siempre. Valores 0=false o 1=true.
  - error: código de error retornado.
  - filename: ruta al archivo para el cual se activa el defecto. Si no se ingresa filename se inyecta en cualquier archivo. Si filename tiene espacios debe ingresarse entre ""

closeld

- Usos:
  - closeld <when> <allways> <error> [filename]
- Descripción: Defecto sobre función Close de Glibc.
- Parámetros:
  - when: en qué número de invocación a la función se activa el defecto.
  - allways: si el defecto luego de activado debe ejecutarse siempre. Valores 0=false o 1=true.
  - error: código de error retornado.
  - filename: ruta al archivo para el cual se activa el defecto. Si no se ingresa filename se inyecta en cualquier archivo. Si filename tiene espacios debe ingresarse entre “”

fcloseld

- Usos:
  - fcloseld <when> <allways> <error> [filename]
- Descripción: Defecto sobre función fclose de Glibc.
- Parámetros:
  - when: en qué número de invocación a la función se activa el defecto.
  - allways: si el defecto luego de activado debe ejecutarse siempre. Valores 0=false o 1=true.
  - error: código de error retornado.
  - filename: ruta al archivo para el cual se activa el defecto. Si no se ingresa filename se inyecta en cualquier archivo. Si filename tiene espacios debe ingresarse entre “”

freadld

- Usos:
  - freadld <when> <allways> <injectionType=1> <error> [filename]
  - freadld <when> <allways> <injectionType=2> <faultValue> <offset> [filename]
  - freadld <when> <allways> <injectionType=3> <faultValue> <targetValue> [filename]
- Descripción: Defecto sobre función fread de Glibc.
- Parámetros:
  - when: en qué número de invocación a la función se activa el defecto.

- allways: si el defecto luego de activado debe ejecutarse siempre. Valores 0=false o 1=true.
- injectionType: indica el tipo de error a inyectar. El valor 1 se usa cuando el defecto consta en retornar un error. El valor 2 se usa cuando el defecto consta en modificar lo leído. El valor 3 se usa cuando el defecto consiste en remplazar la cadena targetValue por faultValue.
- error: código de error retornado.
- faultValue: cadena a inyectar. Si contiene espacios debe estar entre “”.
- targetValue: cadena que será remplazada por faultValue. Si contiene espacios debe estar entre “”.
- offset: Byte desde el que se inserta la cadena.
- filename: ruta al archivo para el cual se activa el defecto. Si no se ingresa filename se inyecta en cualquier archivo. Si filename tiene espacios debe ingresarse entre “”

fwriteld

- Usos:
  - fwriteld <when> <allways> <injectionType=1> <error> [filename]
  - fwriteld <when> <allways> <injectionType=2> <faultValue> <offset> [filename]
  - fwriteld <when> <allways> <injectionType=3> <faultValue> <targetValue> [filename]
- Descripción: Defecto sobre función fwrite de Glibc.
- Parámetros:
  - when: en qué número de invocación a la función se activa el defecto.
  - allways: si el defecto luego de activado debe ejecutarse siempre. Valores 0=false o 1=true.
  - injectionType: indica el tipo de error a inyectar. El valor 1 se usa cuando el defecto consta en retornar un error. El valor 2 se usa cuando el defecto consta en modificar lo que se va a escribir. El valor 3 se usa cuando el defecto consiste en remplazar la cadena targetValue por faultValue.
  - error: código de error retornado.
  - faultValue: cadena a inyectar. Si contiene espacios debe estar entre “”.
  - targetValue: cadena que será remplazada por faultValue. Si contiene espacios debe estar entre “”.
  - offset: Byte desde el que se inserta la cadena.
  - filename: ruta al archivo para el cual se activa el defecto. Si no se ingresa filename se inyecta en cualquier archivo. Si filename tiene espacios debe ingresarse entre “”.

readfileld

- Usos:
  - readfileld <when> <allways> <injectionType=1> <error> [filename]
  - readfileld <when> <allways> <injectionType=2> <faultValue> <offset> [filename]
  - readfileld <when> <allways> <injectionType=3> <faultValue> <targetValue> [filename]
- Descripción: Defecto sobre función Read de Glibc cuando se ejecuta sobre un archivo.
- Parámetros:
  - when: en qué número de invocación a la función se activa el defecto.
  - allways: si el defecto luego de activado debe ejecutarse siempre. Valores 0=false o 1=true.
  - injectionType: indica el tipo de error a inyectar. El valor 1 se usa cuando el defecto consta en retornar un error. El valor 2 se usa cuando el defecto consta en modificar lo leído. El valor 3 se usa cuando el defecto consiste en remplazar la cadena targetValue por faultValue.
  - error: código de error retornado.
  - faultValue: cadena a inyectar. Si contiene espacios debe estar entre “”.
  - targetValue: cadena que será remplazada por faultValue. Si contiene espacios debe estar entre “”.
  - offset: Byte desde el que se inserta la cadena.
  - filename: ruta al archivo para el cual se activa el defecto. Si no se ingresa filename se inyecta en cualquier archivo. Si filename tiene espacios debe ingresarse entre “”

readsoketld

- Usos:
  - readsocketld <when> <allways> <injectionType=1> <error> [matchLocalAddress matchLocalPort matchremoteAddress matchremotePort localAddress localPort remoteAddress remotePort]
  - readsocketld <when> <allways> <injectionType=2> <faultValue> <offset> [matchLocalAddress matchLocalPort matchremoteAddress matchremotePort localAddress localPort remoteAddress remotePort]
  - readsocketld <when> <allways> <injectionType=3> <faultValue> <targetValue> [matchLocalAddress matchLocalPort matchremoteAddress matchremotePort localAddress localPort remoteAddress remotePort]

- Descripción: Defecto sobre función Read de Glibc cuando se ejecuta sobre un socket. Si no se especifican los 8 valores entre [ ] se carga el AddNet como Null y el defecto inyecta en cualquier lectura de socket.
- Parámetros:
  - when: en qué número de invocación a la función se activa el defecto.
  - allways: si el defecto luego de activado debe ejecutarse siempre. Valores 0=false o 1=true.
  - injectionType: indica el tipo de error a inyectar. El valor 1 se usa cuando el defecto consta en retornar un error. El valor 2 se usa cuando el defecto consta en modificar lo leído. El valor 3 se usa cuando el defecto consiste en remplazar la cadena targetValue por faultValue.
  - error: código de error retornado.
  - faultValue: cadena a inyectar. Si contiene espacios debe estar entre "".
  - targetValue: cadena que será remplazada por faultValue. Si contiene espacios debe estar entre "".
  - offset: Byte desde el que se inserta la cadena.
  - matchLocalAddress: si para activar el defecto el localAddress debe coincidir. Valores 0=false o 1=true.
  - matchLocalPort: si para activar el defecto el localPort debe coincidir. Valores 0=false o 1=true.
  - matchRemoteAddress: si para activar el defecto el remoteAddress debe coincidir. Valores 0=false o 1=true.
  - matchRemotePort: si para activar el defecto el remotePort debe coincidir. Valores 0=false o 1=true.

writefileld

- Usos:
  - writefileld <when> <allways> <injectionType=1> <error> [filename]
  - writefileld <when> <allways> <injectionType=2> <faultValue> <offset> [filename]
  - writefileld <when> <allways> <injectionType=3> <faultValue> <targetValue> [filename]
- Descripción: Defecto sobre función Write de Glibc cuando se ejecuta sobre un archivo.
- Parámetros:
  - when: en qué número de invocación a la función se activa el defecto.
  - allways: si el defecto luego de activado debe ejecutarse siempre. Valores 0=false o 1=true.

- injectionType: indica el tipo de error a inyectar. El valor 1 se usa cuando el defecto consta en retornar un error. El valor 2 se usa cuando el defecto consta en modificar lo que se va a escribir. El valor 3 se usa cuando el defecto consiste en remplazar la cadena targetValue por faultValue.
- error: código de error retornado.
- faultValue: cadena a inyectar. Si contiene espacios debe estar entre "".
- targetValue: cadena que será remplazada por faultValue. Si contiene espacios debe estar entre "".
- offset: Byte desde el que se inserta la cadena.
- filename: ruta al archivo para el cual se activa el defecto. Si no se ingresa filename se inyecta en cualquier archivo. Si filename tiene espacios debe ingresarse entre "".

writesocketld

- Usos:
  - writesocketld <when> <allways> <injectionType=1> <error> [matchLocalAddress matchLocalPort matchremoteAddress matchremotePort localAddress localPort remoteAddress remotePort]
  - writesocketld <when> <allways> <injectionType=2> <faultValue> <offset> [matchLocalAddress matchLocalPort matchremoteAddress matchremotePort localAddress localPort remoteAddress remotePort]
  - writesocketld <when> <allways> <injectionType=3> <faultValue> <targetValue> [matchLocalAddress matchLocalPort matchremoteAddress matchremotePort localAddress localPort remoteAddress remotePort]
- Descripción: Defecto sobre función Read de Glibc cuando se ejecuta sobre un socket. Si no se especifican los 8 valores entre [ ] se carga el AddNet como Null y el defecto inyecta en cualquier lectura de socket.
- Parámetros:
  - when: en qué número de invocación a la función se activa el defecto.
  - allways: si el defecto luego de activado debe ejecutarse siempre. Valores 0=false o 1=true.
  - injectionType: indica el tipo de error a inyectar. El valor 1 se usa cuando el defecto consta en retornar un error. El valor 2 se usa cuando el defecto consta en modificar lo leído. El valor 3 se usa cuando el defecto consiste en remplazar la cadena targetValue por faultValue.
  - error: código de error retornado.
  - faultValue: cadena a inyectar. Si contiene espacios debe estar entre "".
  - targetValue: cadena que será remplazada por faultValue. Si contiene espacios debe estar entre "".

- offset: Byte desde el que se inserta la cadena.
- matchLocalAddress: si para activar el defecto el localAddress debe coincidir. Valores 0=false o 1=true.
- matchLocalPort: si para activar el defecto el localPort debe coincidir. Valores 0=false o 1=true.
- matchRemoteAddress: si para activar el defecto el remoteAddress debe coincidir. Valores 0=false o 1=true.
- matchRemotePort: si para activar el defecto el remotePort debe coincidir. Valores 0=false o 1=true.

G.2.3. Defectos para modo Ptrace

open

- Usos:
    - open <when> <allways> <error> [filename]
- Descripción: Defecto sobre `syscall` Open.
- Parámetros:
    - when: en qué número de invocación a la función se activa el defecto.
    - allways: si el defecto luego de activado debe ejecutarse siempre. Valores 0=false o 1=true.
    - error: código de error retornado.
    - filename: ruta al archivo para el cual se activa el defecto. Si no se ingresa filename se inyecta en cualquier archivo. Si filename tiene espacios debe ingresarse entre “”

close

- Usos:
    - close <when> <allways> <error> [filename]
- Descripción: Defecto sobre `syscall` Close.
- Parámetros:
    - when: en qué número de invocación a la función se activa el defecto.
    - allways: si el defecto luego de activado debe ejecutarse siempre. Valores 0=false o 1=true.
    - error: código de error retornado.
    - filename: ruta al archivo para el cual se activa el defecto. Si no se ingresa filename se inyecta en cualquier archivo. Si filename tiene espacios debe ingresarse entre “”

readfile

- Usos:
  - readfile <when> <allways> <faultValue> <offset> [filename]
- Descripción: Defecto sobre `syscall` Read cuando se ejecuta sobre un archivo.
- Parámetros:
  - when: en qué número de invocación a la función se activa el defecto.
  - allways: si el defecto luego de activado debe ejecutarse siempre. Valores 0=false o 1=true.
  - faultValue: cadena a inyectar. Si contiene espacios debe estar entre “”.
  - offset: Byte desde el que se inserta la cadena.
  - filename: ruta al archivo para el cual se activa el defecto. Si no se ingresa filename se inyecta en cualquier archivo. Si filename tiene espacios debe ingresarse entre “”

writefile

- Usos:
  - writefile <when> <allways> <faultValue> <offset> [filename]
- Descripción: Defecto sobre `syscall` Write cuando se ejecuta sobre un archivo.
- Parámetros:
  - when: en qué número de invocación a la función se activa el defecto.
  - allways: si el defecto luego de activado debe ejecutarse siempre. Valores 0=false o 1=true.
  - faultValue: cadena a inyectar. Si contiene espacios debe estar entre “”.
  - offset: Byte desde el que se inserta la cadena.
  - filename: ruta al archivo para el cual se activa el defecto. Si no se ingresa filename se inyecta en cualquier archivo. Si filename tiene espacios debe ingresarse entre “”.

readsocket

- Usos:
  - readsocket <when> <allways> <faultValue> <offset> [matchLocalAddress matchLocalPort matchremoteAddress matchremotePort localAddress localPort remoteAddress remotePort]

- Descripción: Defecto sobre `syscall` Read cuando se ejecuta sobre un socket. Si no se especifican los 8 valores entre [ ] se carga el AddNet como Null y el defecto inyecta en cualquier lectura de socket.
- Parámetros:
  - when: en qué número de invocación a la función se activa el defecto.
  - allways: si el defecto luego de activado debe ejecutarse siempre. Valores 0=false o 1=true.
  - faultValue: cadena a inyectar. Si contiene espacios debe estar entre "".
  - offset: Byte desde el que se inserta la cadena.
  - matchLocalAddress: si para activar el defecto el localAddress debe coincidir. Valores 0=false o 1=true.
  - matchLocalPort: si para activar el defecto el localPort debe coincidir. Valores 0=false o 1=true.
  - matchRemoteAddress: si para activar el defecto el remoteAddress debe coincidir. Valores 0=false o 1=true.
  - matchRemotePort: si para activar el defecto el remotePort debe coincidir. Valores 0=false o 1=true.

writesocket

- Usos:
  - writesocket <when> <allways> <faultValue> <offset> [matchLocalAddress matchLocalPort matchremoteAddress matchremotePort localAddress localPort remoteAddress remotePort]
- Descripción: Defecto sobre `syscall` Read cuando se ejecuta sobre un socket. Si no se especifican los 8 valores entre [ ] se carga el AddNet como Null y el defecto inyecta en cualquier lectura de socket.
- Parámetros:
  - when: en qué número de invocación a la función se activa el defecto.
  - allways: si el defecto luego de activado debe ejecutarse siempre. Valores 0=false o 1=true.
  - faultValue: cadena a inyectar. Si contiene espacios debe estar entre "".
  - offset: Byte desde el que se inserta la cadena.
  - matchLocalAddress: si para activar el defecto el localAddress debe coincidir. Valores 0=false o 1=true.
  - matchLocalPort: si para activar el defecto el localPort debe coincidir. Valores 0=false o 1=true.
  - matchRemoteAddress: si para activar el defecto el remoteAddress debe coincidir. Valores 0=false o 1=true.
  - matchRemotePort: si para activar el defecto el remotePort debe coincidir. Valores 0=false o 1=true.

G.2.4. Comandos USB

usbdevice

- Usos: usbdevice <nro_vendor> <nro_product>
- Descripción: Especifica el dispositivo USB sobre el que se realizará la inyección a través del número de vendedor y de producto.
- Parámetros:
  - nro_vendor: número de vendedor del dispositivo.
  - nro_product: número de producto del dispositivo.

usbstop

- Usos: usbstop
- Descripción: Detiene la inyección de defectos sobre USB.

usbstatus

- Usos: usbstatus <FaultType> <when> <frecuency> <allways> <faultValue>
- Descripción: Defecto de cambio de estatus de los URBs.
- Parámetros:
  - When: en qué número de invocación a la función se activa el defecto.
  - Frecuency: numero de URBs inyectados antes de dejar pasar uno sin inyectar.
  - Allways: si el defecto luego de activado debe ejecutarse siempre. Valores 0=false o 1=true.
  - FaultValue: valor por el cual se cambia el status del URB.

usbenumeration

- Usos: usbenumeration <injectionType> <configuration> <interface> <altSetting> <endPoint> <atribute> <faultValue>
- Descripción: Defecto de cambio de estatus de los URBs.
- Parámetros:
  - injectionType: estructura sobre la que se realiza el cambio de atributo. Los valores posibles son 0=Configuration, 1=Interface y 2=Endpoint.
  - frecuency: numero de URBs inyectados antes de dejar pasar uno sin inyectar.
  - allways: si el defecto luego de activado debe ejecutarse siempre. Valores 0=false o 1=true.

- atribute: indica el nombre del atributo a cambiar en la configuración especificada. Los posibles valores son bEndpointAddress, bInterval y bDescriptorType.
- configuration: número de configuración. Usado para especificar la estructura a la que se modificará el atributo.
- interface: número de interfaz. Usado para especificar la estructura a la que se modificará el atributo.
- altSetting: número de altSetting. Usado para especificar la estructura a la que se modificará el atributo.
- endPoint: número de endpoint. Usado para especificar la estructura a la que se modificará el atributo.
- faultValue: valor a asignar al atributo especificado.

G.3. Logs

Los sucesos de mayor importancia acontecidos en la campaña de inyección son registrados. Este registro es realizado en el log del sistema para las inyecciones en el protocolo USB y en archivos en la inyección sobre `syscalls` realizadas en los modos LD_PRELOAD y Ptrace. A continuación se presentan ejemplos de los logs.

Log de inyección sobre `syscalls`

El log de la inyección sobre `syscalls` se realiza en archivos cuyo nombre contiene el cumple el siguiente formato *log_aa-mm-dd_hh-ii-ss.txt*, donde lo que sigue a la palabra log es la fecha de la ejecución del programa de inyección expresada de forma numérica.

Los sucesos que se loguean son:

- inicio de programa de inyección.
- finalización de programa de inyección.
- modo de inyección, 0 indica Ptrace y 1 indica LD_PRELOAD.
- defectos agregados a la campaña de inyección.
- defectos activados y algunos de sus valores.

Un ejemplo de log para el modo Ptrace es el siguiente:

```
[22:08:53] Inicio Log
[22:08:53] Controller: Init mode=0
[22:10:17] Parser: se agrego OpenFault
[22:10:17] Parser: se agrego CloseFault
[22:10:19] Defecto activado - OpenFault, filePath=/tmp/test.txt,
error=-2
[22:10:23] Defecto activado - OpenFault, filePath=/tmp/test.txt,
error=-2
[22:10:27] Defecto activado - OpenFault, filePath=/tmp/test.txt,
error=-2
```

```
[22:15:52] Parser: se agrego ReadFileFault
[22:16:39] Defecto activado - ReadFileFault, filePath=All
[22:18:32] Fin Log
```

Un ejemplo de log para el modo LD_PRELOAD es el siguiente:

```
[16:38:08] Inicio Log
[16:38:08] Controller: Init mode=1
[16:39:13] Parser: se agrego WriteSocketFaultLD
[16:39:13] Parser: se agrego WriteSocketFaultLD
[16:39:32] Defecto activado - function=6, injectionType=3
[16:39:37] Defecto activado - function=6, injectionType=3
[16:39:41] Defecto activado - function=6, injectionType=3
[16:43:02] Parser: se agrego WriteSocketFaultLD
[16:43:02] Parser: se agrego ReadSocketFaultLD
[16:43:09] Defecto activado - function=5, injectionType=3
[16:43:14] Defecto activado - function=5, injectionType=3
[16:48:54] Fin Log
```

Log de inyección sobre USB

El log de la inyección sobre USB se realiza sobre el log del sistema. Por lo tanto para acceder al mismo se debe ejecutar el comando "cat /var/log/messages | grep usbfi".

Los sucesos que se loguean son:

- inicio de modulo de inyección USBFI.
- finalización de modulo de inyección USBFI.
- especificación de dispositivo sobre el que se debe inyectar.
- inyecciones realizadas. Para estas si se trata de inyección de cambio de estatus se especifica el valor que se asociará a dicho atributo. Si es una inyección de enumeración se especifica el atributo modificado y el valor que se le asigna.

Un ejemplo de log es el siguiente:

```
Sep 18 17:05:52 linux-2aih kernel: [ 4658.433793]
usbfi[10964]: Inicio usbfi
Sep 18 17:06:02 linux-2aih kernel: [ 4668.482485]
usbfi[10840]: usbfi vendor=4d8 product=c
Sep 18 17:06:59 linux-2aih kernel: [ 4725.253139]
usbfi[0]: inyección realizada bmAttributes=3
Sep 18 17:06:59 linux-2aih kernel: [ 4725.253167]
usbfi[0]: inyección realizada bInterval=5
Sep 18 17:12:06 linux-2aih kernel: [ 5032.496901]
usbfi[0]: inyección realizada status=-84
Sep 18 17:12:06 linux-2aih kernel: [ 5032.500877]
usbfi[4133]: inyección realizada status=-84
Sep 18 17:12:07 linux-2aih kernel: [ 5033.424851]
usbfi[4133]: inyección realizada status=-84
Sep 18 17:12:07 linux-2aih kernel: [ 5033.456850]
usbfi[1814]: inyección realizada status=-84
Sep 18 17:21:22 linux-2aih kernel: [ 5588.446082]
usbfi[11254]: Fin usbfi
```

Apéndice H

Pruebas del módulo USBFI

En esta sección se describe un conjunto de pruebas realizadas con el producto desarrollado en lo que refiere a inyección de defectos sobre el protocolo USB.

H.1. Pruebas

Con el objetivo de cuantificar el efecto en alguna de las inyecciones que permite realizar el producto sobre dispositivos USB, se han realizado un conjunto de pruebas sobre un dispositivo en particular. En estas pruebas se miden las diferencias en el comportamiento del dispositivo cuando este opera de forma normal y cuando lo hace bajo inyección.

Las pruebas consisten en medir los tiempos de "ida" y "vuelta" (ping) de cada uno de los URBs enviados al dispositivo, sin realizar ninguna modificación y luego de realizar inyecciones de defectos en el momento de la enumeración a través de la aplicación construida.
Para este fin se cuenta con un dispositivo USB cuya funcionalidad es la de devolver cualquier URB que sea enviado a su endpoint. Entre otros, el dispositivo cuenta con dos endpoint BULK, uno IN y otro OUT los cuales son usados para las pruebas.
Para medir los tiempos de cada ping se implementa una aplicación que realiza el envió de un URB al dispositivo luego espera a que este vuelva y registra el tiempo de ida y vuelta. Para cada tipo de prueba se registran los tiempos de 5000 URBs, de forma de contar con una cantidad considerable de medidas al momento de realizar las comparaciones.

El escenario construido consta de un HUB 1.1 conectado a un puerto USB del computador y luego tanto el dispositivo utilizado para las pruebas como dos pendrives son conectados al HUB 1.1.

Las condiciones de prueba para el escenario construido son dos:

1. Realizar medidas de forma tal que solo se transmitan datos entre el computador y el dispositivo utilizado para realizar las mismas, logrando en este caso que los URBs no compitan por los recursos de ancho de banda.

2. Realizar medidas forzando a los URBs a competir por los recursos de ancho de banda, para ello se mantiene transmitiendo datos a los dos pendrives que se tiene conectados.

Dado que el ancho de banda para USB 1.1 es de 12Mbps, se puede comprobar fácilmente que con estos tres dispositivos transmitiendo al mismo tiempo alcanza para lograr saturar el ancho de banda.

A continuación se muestra los resultados obtenidos en las distintas pruebas, en todos los casos el URB enviado es de tipo BULK.

1) Sin competencia por el ancho de banda

inyección realizada	**tiempo total**	**media**	**desviación estándar**	**varianza**
ninguna	19,998	0,0039996	6,63E-05	4,40E-09
transf a interrupt y bInterval a 255	319,996	0,0639964	0,000123248	1,52E-08
transf a interrupt y bInterval a 5	40,007	0,0079976	0,000244144	5,96E-08

2) Con competencia por el ancho de banda

inyección realizada	**tiempo total**	**media**	**desviación estándar**	**varianza**
ninguna	10,510	0,0021020	0,000313713	9,84E-08
transf a interrupt y bInterval a 255	319,994	0,0639952	0,000164871	2,72E-08
transf a interrupt y bInterval a 5	40,043	0,0080082	0,000262577	6,89E-08

H.2. Conclusión

Se pueden realizar varias observaciones de las medidas obtenidas:

Al comparar los tiempos medidos para el dispositivo sin inyectar entre un escenario sin competencia por el ancho de banda y un escenario con competencia, se observa que el escenario con competencia favorece el cumplimiento de los envíos de URBs. Este comportamiento no era el esperado pero para determinar sus causas requiere de un estudio más profundo de la implementación del protocolo USB realizada en el USB Core.

Para los casos de inyección se puede observar que al cambiar el tipo de transferencia a interrupt el tiempo aumenta considerablemente. Esto en primer lugar demuestra que la inyección tuvo efecto en la forma en que es tratado el dispositivo y como se realizan las transferencias. En segundo lugar muestra que para un endpoint interrupt el tráfico de URBs demora más en completar su ciclo. Una posible explicación es que si bien las transferencias interrupt tienen prioridad sobre las bulk, también tienen una cota de tiempo que es la indicada en el binterval. Un URB no es enviado al dispositivo USB hasta que dicho tiempo se cumple, esto sumado a que en cada URB enviado al dispositivo bajo prueba

contiene poca información por lo que es rápidamente despachado, puede ser la causa del enlentecimiento.

Cuando se compara los casos donde se inyecta con valores de binterval de 255, con los casos con binterval de 5, se puede observar que cuanto menor es el binterval menor es el tiempo requerido para el envió y retorno de un URB. Esto demuestra en primer lugar que la inyección de cambio de valor en binterval altera el funcionamiento del dispositivo y como es de esperarse a un menor intervalo de espera de envío se corresponde un menor tiempo total en el envío de los URBs.

De estas observaciones se puede concluir que las inyecciones se están realizando con éxito y están alterando el comportamiento de del dispositivo y como es tratado el mismo por el USB Core. Varias son las posibles interpretaciones de los resultados obtenidos, pero para llegar a conclusiones serias se debe realizar un estudio más profundo del funcionamiento del USB Core que escapa al alcance de este proyecto. De esta manera el producto desarrollado demuestra ser efectivo al momento de alterar el tráfico de URBs y más específicamente en lo que refiere a los cambios de configuración en la enumeración.