Universidad Zaragoza

Trabajo Fin de Grado

DADE: MOTOR DE DETECCIÓN DE DATOS ANÓMALOS

AUTOR

Luis Fueris Martín

DIRECTORES

José Luis Briz Velasco
Darío Suárez Gracia

ESCUELA DE INGENIERÍA Y ARQUITECTURA

2018

TRABAJOS DE FIN DE GRADO / FIN DE MÁSTER

Escuela de Ingeniería y Arquitectura Universidad Zaragoza

DECLARACIÓN DE AUTORÍA Y ORIGINALIDAD

(Este documento debe acompañar al Trabajo Fin de Grado (TFG)/Trabajo Fin de Máster (TFM) cuando sea depositado para su evaluación).

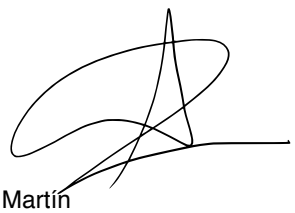
D./Dª. Luis Fueris Martín,

con nº de DNI 73029746P en aplicación de lo dispuesto en el art. 14 (Derechos de autor) del Acuerdo de 11 de septiembre de 2014, del Consejo de Gobierno, por el que se aprueba el Reglamento de los TFG y TFM de la Universidad de Zaragoza,

Declaro que el presente Trabajo de Fin de (Grado/Máster) Grado, (Título del Trabajo) DADE: Motor de Detección de Datos Anómalos,

es de mi autoría y es original, no habiéndose utilizado fuente sin ser citada debidamente.

Zaragoza, 21 de septiembre de 2018



Fdo: Luis Fueris Martín

RESUMEN

Actualmente están muy extendidos en los centros de datos los *low-volume servers*, que albergan máquinas virtuales con aplicaciones accedidas por usuarios de manera concurrente. La seguridad es crucial, sea a nivel de aplicación como a nivel de sistema operativo, para evitar filtración de datos personales. Si el sistema está comprometido y un atacante adquiere permisos de administrador, su próximo movimiento será realizar una escalada de privilegios, ya sea horizontal (dentro de la DMZ) o vertical (red interna). Una aproximación para intentar aumentar los privilegios es instalar código malicioso para, por ejemplo, corromper la filtración de paquetes y acceder a otras máquinas. Existen prototipos como DADE: *A fast Data Anomaly Detection Engine for kernel integrity monitoring* [4] que ayuda a detectar cambios de código o datos dentro del kernel. DADE provee un mecanismo que ayuda a mantener la integridad del sistema operativo Linux. Realiza una detección basada en la monitorización de los objetos del kernel. Los objetos creados en el kernel pueden ser caracterizados por su *stacktrace*, que es la cadena ordenada de invocaciones a funciones hasta llegar al interfaz de asignación de memoria física del kernel. En este trabajo se ha implementado el prototipo DADE introducido en [4], y se han aportado nuevos métodos para la detección de anomalías. Además de reproducir las técnicas de detección de ataques referidas en dicho artículo, se han desarrollado tres pruebas de concepto propias para atacar el código del kernel y comprobar la viabilidad y eficacia del diseño propuesto.

ABSTRACT

Nowadays, low-volume servers prevail in data centers. They host virtual machines which run multiple applications concurrently accessed by the users. Security is of paramount importance at both the application and operating system level, to avoid personal data leaks. If the system is compromised and an eavesdropper acquires rootly powers, her or his next move will be to perform a privilege rampage, either horizontal (inside the DMZ) or vertical (local net). A way to increase the privileges consists in installing malicious code in order to garble packet filtering and to access to other machines. There exist prototypes like DADE: *A fast Data Anomaly Detection Engine for kernel integrity monitoring* [4], which help to detect code or data modification inside the operating system's kernel. DADE provides a mechanism to maintain operating system integrity based on monitoring kernel objects. The created objects inside kernel can be characterized by *stack-trace*, which is the ordered trace of function invocations down to the kernel physical memory allocation interface. We have implemented the DADE prototype introduced in [4], and contributed with new methods for detecting anomalies. Besides reproducing the kernel attack detection techniques exposed in the paper, we have implemented three new kernel attacks as a proof of concept to test the viability and effectiveness of the proposed technique.

The only truly secure system is one that is powered off, cast in a block of concrete and sealed in a lead-lined room with armed guards - and even then I have my doubts.

— Gene Spafford

AGRADECIMIENTOS

En primer lugar, quisiera agradecer a José Luis Briz y Darío Suárez por dirigir este trabajo, además de darme la oportunidad de introducirme a la investigación. *Gracias por enseñarme a pensar*. A mis amigos y compañeros del grado por darme ánimos cuando estaba decaído. A mis padres y hermano por darme todo lo que estaba en su mano. Y por último, a mis amigos Iñigo, Mónica y Alejandro.

Muchas gracias.

ÍNDICE GENERAL

ÍNDICE DE FIGURAS

ÍNDICE DE CUADROS

LISTINGS

ACRÓNIMOS

DMZ	Demilitarized Zone
LOPD	Ley Orgánica de Protección de Datos
DDoS	Distributed Denial-Of-Service
VMM	Virtual Machine Monitor
CP	Control Program
ISA	Instruction Set Architecture
VM	Virtual Machine
IF	Interrupt-enable Flag
KVM	Kernel-based Virtual Machine
LWP	Light-Weight Process
SoC	System on Chip
CPU	Central Processing Unit
ARM	Advanced RISC Machines
GPU	Graphics Processing Unit
QEMU	Quick EMUlator
SMP	Symmetric Multi-Processin
DADE	Data Anomaly Detection Engine
PC	Program Counter
ELF	Executable and Linking Format
COFF	Common Object File Format
MS-DOS	MicroSoft Disk Operating System
BSD	Berkeley Software Distribution
SVM	Support Vector Machine

1 INTRODUCCIÓN

1.1 MOTIVACIÓN

En los últimos años, la seguridad informática está cobrando una importancia relevante debido a la nueva ley de protección de datos en España (LOPD), así también, por recientes ataques producidos en la actualidad. Un ejemplo serían los 6 millones de cuentas expuestas en Instagram (agosto 2017), ataques DDoS a páginas web eslovacas (junio 2018), intrusión a la infraestructura IT de Liberty Seguros (junio 2018). Incluso hay páginas web como `http://www.norse-corp.com` que muestran ataques en tiempo real. Las máquinas comprometidas en su mayoría están virtualizadas, es decir, el sistema operativo no es ejecutado en una máquina física. Además, el uso de máquinas virtuales está muy extendido entre los proveedores de *cloud*. Una de las preocupaciones, sino la mayor, de los proveedores, clientes e investigadores es la seguridad e integridad en el código y datos. Por lo tanto, cuando hay un intruso es necesario mitigar la escalada de privilegios, ya sea horizontal o vertical.

Una escala de privilegios horizontal consiste en conseguir cuentas de usuarios sin elevar el privilegio. Sin embargo, una escalada de privilegios vertical consiste en conseguir usuarios con más privilegios. Se puede aplicar la misma definición a máquinas (DMZ o red interna).

1.2 OBJETIVOS

El principal objetivo es complementar mecanismos de defensa basados en *hash*, como los que se pueden aplicar a los módulos del KERNEL LINUX [5] (Sec. 2.2). Para ello se aplicarán las técnicas propuestas por Hayoon Yi et al. [4] replicando el diseño de DADE. También, se proporcionarán nuevos métodos de detección de modificaciones y diferentes pruebas de concepto.

1.3 ALCANCE

La consecución de los objetivos requiere en primer lugar preparar el laboratorio con las máquinas virtuales, además de realizar un estudio del interfaz de asignación de memoria física en el kernel. Una vez puestas en marcha las máquinas virtuales, se procede a implementar el sistema DADE mediante la técnica *backtrace-naming* diseñada por Hayoon Yi et al. [4]. Para ello, es necesario modificar el kernel de la máquina virtualizada insertando instrucciones que realicen un seguimiento de las funciones ejecutadas al asignar memoria física. Cuando la máquina virtual (huésped) tenga que asignar memoria, el control pasará al *Hypervisor* desde el cual se realizarán las operaciones para extraer los datos que requiere *backtrace-naming*. Si se sigue el

sistema descrito en el artículo seminal de DADE [4], al tener que hacer un cambio de contexto por cada asignación de memoria física, tiene lugar una caída grave del rendimiento de la máquina virtual.

El cambio producido entre Hypervisor-VM y viceversa, recibe el nombre de world switch.

Sin embargo, Linux proporciona mecanismos de depuración para ayudar a los desarrolladores de sistemas a su implementación. El más característico es FTRACE (Sec. 2.2.2), por lo que se ha decidido utilizar este recurso en lugar de recurrir a la técnica descrita en [4]. Se ha estudiado la documentación disponible para encontrar un mecanismo que pueda extraer todos los datos que replique la técnica *backtrace-naming*.

Una vez desarrollado el prototipo de DADE, se han implementado tres pruebas de concepto que demuestran que el mecanismo diseñado detecta correctamente los ataques al kernel. Dichas pruebas, han sido elegidas minuciosamente para mostrar de manera incremental el peligro que puede tener insertar código no legítimo en el sistema operativo. Cada ataque lleva un estudio de las diferentes estructuras del kernel, y su modifcación se ha hecho de manera precisa para cambiar únicamente lo imprescindible.

El diagrama de Gantt del proyecto está en el Apéndice A.1 y las horas se muestran en el Cuadro 1.1

TAREA	HORAS
1. Estudio fundamentos virtualización y kernel	40
2. Estudio, análisis y diseño de DADE	30
3. Laboratorio experimental	25
4. Implementación DADE	60
5. Implementación ataque nº 1. literal pool spoofing	30
6. Implementación ataque nº 2. unreliability binary script format	20
7. Implementación ataque nº 3. garbling Netfilter ipv4	45
8. Escritura memoria y documentación	70
Horas totales:	320

Cuadro 1.1: Horas dedicadas

1.4 DESCRIPCIÓN DEL DOCUMENTO

En el Capítulo 2 están descritos los fundamentos sobre los que se sustenta el trabajo, VIRTUALIZACIÓN (Sec. 2.1) y KERNEL LINUX (Sec. 2.2). En el Capítulo 3 están descritos el HARDWARE (Sec. 3.1) y SOFTWARE (Sec. 3.2) utilizado para desarrollar las pruebas. El Capítulo 4 describe la técnica *backtrace-naming* (Sec. 4.2) y la ARQUITECTURA (Sec. 4.1) de las herramientas desarrolladas.

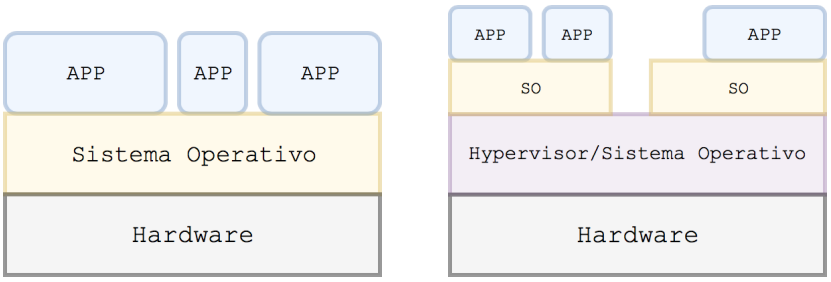
Las pruebas de concepto detalladas se encuentran en el Capítulo 5. Finalmente, el Capítulo 6 contiene las conclusiones y trabajo futuro a realizar. Las explicaciones adicionales, código y trazas están completas en el Capítulo A.

2 FUNDAMENTOS

A continuación se introducen los conceptos relacionados con VIRTUALIZACIÓN (Sec. 2.1) y KERNEL LINUX (Sec. 2.2) dos pilares fundamentales de este trabajo. Gracias al *Hypervisor* que proporciona un entorno de ejecución a las máquinas virutales se podrá monitorizar la actividad del kernel para la detección de ataques.

2.1 VIRTUALIZACIÓN

La virtualización es una técnica que permite la ejecución de un sistema operativo con sus aplicaciones en un hardware no real [8, 10]. En la Figura 2.1 se muestra la diferencia respecto a una máquina física.



(a) Máquina física (b) Máquinas virtuales

Figura 2.1: Diferencia máquina física (anfitrión) y virtual (huésped)

Las máquinas virtuales supusieron un gran cambio para las empresas que tenían aplicaciones ejecutándose en servidores físicos, los cuales no llegaban a un pleno rendimiento. Este cambio permitió ejecutar sobre un mismo servidor físico varias máquinas virtuales con aplicaciones que necesitan entornos específicos. Este proceso se conoce como *consolidación* de servidores, y comporta una serie de ventajas.

- **Aislamiento y protección frente a fallos**. Un sistema operativo es ejecutado de manera aislada en una máquina virtual. Si falla, no tiene por qué afectar a las demás.
- **Desarollo ágil y multiplataforma**. Las aplicaciones necesitan bibliotecas específicas, o incluso versiones diferentes de una misma biblioteca. Además, el proveedor de la aplicación puede exigir un sistema operativo y versión concretos. Esto permite también el desarollo para distintas plataformas de una manera eficaz.

- **Seguridad**. El aislamiento entre máquina física (anfitrión) y virtual (huésped) es crucial en entornos de producción. Una máquina virtual infectada con *malware* no debe permitir la propagación a los demás sistemas del centro de datos.
- **Ejecución de aplicaciones legadas**. Las máquinas virtuales pueden proporcionar un entorno de ejecución legado, lo que permite seguir utilizando aplicaciones que de otro modo quedarían obsoletas aunque fueran perfectamente operativas y útiles.

El software encargado de esta tarea se denomina *Hypervisor*, *Virtual Machine Monitor* (VMM) o *Control Program* (CP). Se utilizará el nombre *Hypervisor* para nombrar al software encargado de proporcionar el entorno de ejecución de las máquinas virtuales. Cada máquina, virtual o no, dispone de unos recursos que el sistema operativo ofrece a las aplicaciones, tales como procesadores, memoria, espacio en disco o interfaz de E/S. Dichos recursos deben ser virtualizados por un *Hypervisor* ejecutándose en la máquina física.

2.1.1 *Hypervisor*

Gerald J. Popek et al. [10] asentaron tres axiomas que debe de cumplir un *Hypervisor* para poder ejecutar máquinas virtuales como si fueran un duplicado eficiente y aislado de la máquina real. Es preciso matizar que, hacen referencia a máquinas de 3ª generación (IBM System/370, Honeywell 6000, Digital PDP-10). Esta categorización en generaciones de computadores fue definida de manera aproximada por Peter J. Denning [1].

Los axiomas que Gerald J. Popek et al. [10] definieron son los siguientes:

A1. Proporcionar un entorno idéntico a las aplicaciones como si estuvieran en la máquina física.

A2. Las aplicaciones alojadas en la máquina virtual, en el peor de los casos, sufren disminuciones en la velocidad de ejecución.

A3. Todos los recursos del sistema virtualizado tienen que estar bajo control del *Hypervisor*.

Los microprocesadores actuales, especialmente los de propósito general, han heredado estos modos de ejecución, explotados por los sistemas operativos. En algunos casos se extienden a más de dos modos, como en el caso de Intel x86-64, que ofrece cuatro.

Una máquina convencional de aquella época, como puede ser la IBM 360, tenía dos modos de ejecución: *supervisor* y *usuario*. En modo *supervisor*, el procesador puede ejecutar todo el repertorio de instrucciones de la arquitectura de lenguaje máquina (ISA). En una máquina física, el sistema operativo se ejecuta típicamente en este modo. En modo *usuario*, el procesador únicamente puede ejecutar un subconjunto de instrucciones *no privilegiadas*: el intento de ejecución de una

instrucción no permitida (*privilegiada*) provoca una excepción (i.e. almacenamiento del estado, cambio a modo *supervisor*, y salto a una rutina específica que puede ser *firmware* o formar parte de un sistema operativo concreto). Las aplicaciones se ejecutan en ese modo.

La ejecución en modo usuario del sistema operativo de la VM supuso un problema con las instrucciones no privilegiadas sensibles. Fue resuelto utilizando traductores binarios (estáticos y dinámicos) o interpretes. Posteriormente, se añadió soporte hardware para virtualización.

Un *Hypervisor* tiene que ejecutarse en modo *supervisor* para tener disponibles todas las instrucciones, mientras que sus respectivas máquinas virtuales correrán en modo *usuario*, incluyendo no sólo las aplicaciones sino el propio sistema operativo.

A su vez, existe un subconjunto dentro de las instrucciones *privilegiadas*, las instrucciones *sensibles* que se dividen en:

- Instrucciones *sensibles de control* [1]. Cambian la configuración de los recursos del sistema.
- Instrucciones *sensibles de comportamiento* [2]. Cambian la configuración de los recursos del sistema pero su comportamiento depende de la configuración del procesador (*supervisor* o *usuario*).

La Figura 2.2 relaciona estos conjuntos de instrucciones. Las *sensibles* cambian el estado arquitectónico del sistema. Según A3 los recursos del sistema virtualizado deben ser controlados por el *Hypervisor*, por lo tanto, cada vez que se produce una instrucción de este tipo el *Hypervisor* debe ser consciente de ello. De esta manera, surge el Teorema 1 formulado por Gerald J. Popek et al. [10].

Teorema 1 *Para cualquier computador convencional de tercera generación, puede construirse un Hypervisor si el conjunto de instrucciones sensibles es un subjconjunto de las instrucciones privilegiadas.*

Por lo tanto, cualquier ISA que cumpla dicho teorema podrá ser virtualizada por un *Hypervisor*. En un principio la arquitectura *x86* no era virtualizable, sin embargo, gracias a las extensiones de virtualización este problema fue solucionado. Para más información ver las Figuras A.2 y A.3 del Apéndice A.2.

2.1.2 *Tipos de Hypervisores*

Actualmente existen múltiples formas en la implementación de *Hypervisores* aunque la taxonomía es la siguiente:

1 **Set CPU Timer** [9] (SPT, IBM System/370). Reemplaza el *timer* por un valor referenciado en memoria si la CPU está en modo *supervisor*, en caso contrario, se produce una interrupción.

2 **Pop Stack into Flags Register** [9] (POPF, Intel IA-32). Escribe en los flags del registro de estado el dato extraído de la pila. En modo *usuario* sobreescribe todos los flags excepto IF. En modo *supervisor* se puede modificar dicho flag.

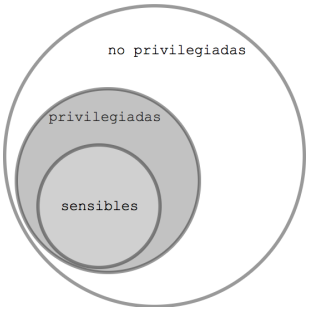
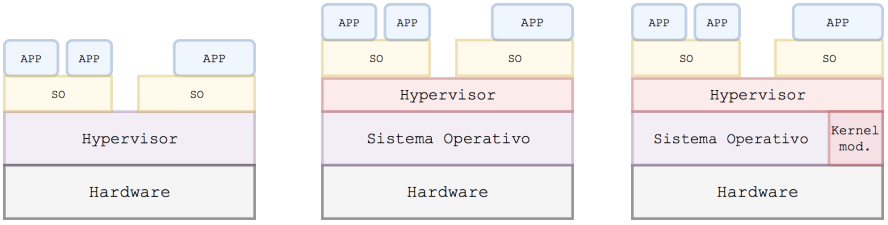


Figura 2.2: Tipos de instrucciones



(a) Tipo 1 (b) Tipo 2 (c) Tipo 1.7

Figura 2.3: Taxonomía de *Hypervisores*

- **Tipo 1**. Corren por encima del hardware real, es decir, en *bare metal*. No poseen un sistema operativo al uso, sino que implementan funcionalidades específicas que necesitan las máquinas virtuales. Las técnicas utilizadas son *Ejecución directa + trap&emulación* (VMware Exsi[3]) y *paravirtualización* (Xen [4]). Una explicación detallada de las técnicas se encuentra en los Apéndices A.2 y A.3 respectivamente.
- **Tipo 2**. Corren por encima del sistema operativo, ejecutando máquinas virtuales con los recursos proporcionados por éste. Un ejemplo sería Bochs IA-32 Emulator[5].
- **Tipo 1.7**. Corren por encima del sistema operativo, sin embargo, la principal diferencia respecto a los de tipo 2 es la utilización de un módulo del kernel para simplificar la comunicación con el hardware real. Un ejemplo sería KVM[6].

La taxonomía de los *Hypervisores* se muestra en la Figura 2.3.

2.2 KERNEL LINUX

Linux[7] fue inicialmente desarrollado por Linus Torvalds en 1991. Este sistema operativo se podía ejecutar en los ordenadores personales

3 Producto VMware Exsi - https://www.vmware.com/es/products/esxi-and-esx.html
4 Proyecto Xen - https://www.xenproject.org
5 Proyecto Bochs IA-32 - http://bochs.sourceforge.net
6 Proyecto KVM - https://www.linux-kvm.org/page/Main_Page
7 Código kernel Linux - https://www.kernel.org

IBM compatibles basados en el microprocesador Intel 80386.

Los PCs IBM compatibles o clones, hace referencia a aquellos ordenadores que son de manera arquitectónica iguales a IBM PC, XT y AT gracias a realizar ingeniería inversa a la BIOS.

Nemeth et al. [2] definieron el núcleo o kernel Linux como el responsable de distribuir recursos y proveer servicios de los cuales dependen los procesos de usuario. Las características principales de Linux son:

- **Kernel monolítico**. Todos los componentes (interfaz de llamadas al sistema, subsistema de ficheros, control de procesos) están integrados en un único ejecutable. Su ejecución es siempre en modo *privilegiado*. Tras el arranque del sistema en ese modo, se ejecutan procesos de usuario en modo *no privilegiado*, y la ejecución de código de kernel únicamente se dispara por excepción. Si la excepción es independiente del proceso en ejecución (como en el caso de las interrupciones hardware) se habla de *ejecución en contexto de interrupción*. Si la excepción ha sido causada por un proceso en ejecución se habla de *ejecución en contexto de proceso*. De este modo, los procesos de usuario pueden ejecutar el código de los servicios del kernel provocando excepciones, pasando a ejecutarse *en modo kernel en contexto de proceso*.

Frente a un kernel monolítico, un microkernel sólo soporta unos servicios mínimos entre los que está la gestión de excepciones, y sobre los que subsistemas pueden implementar interfaces de llamadas al sistema de diferentes sistemas operativos.

- **Soporte para módulos**. Realiza un enlazado (desenlazado) dinámico de partes de código del kernel cuando éste está en ejecución. En MÓDULOS (Sec. 2.2.1) hay una explicación más elaborada.

- **Kernel threading**. Un *kernel thread* se define como un contexto de ejecución ligero dentro del kernel. Son creados y ejecutados para realizar una tarea específica, y pueden estar asociados o no a un proceso de usuario.

Kernel threads: init, keventd, kapmd, kswapd, kblockd.

- **Procesos multithread**. Un proceso puede estar compuesto de varios *threads* (hilos) que comparten los mismos recursos (espacio de direcciones, páginas de memoria, ficheros abiertos). Los hilos creados con la llamada al sistema `clone` están asociados a un *kernel thread* y se denominan a menudo LWP.

- **Modelos de expulsión en el kernel** (**kernel preemption**). Linux permite configurar la expulsión de diferentes maneras, con ligeras variaciones a lo largo de sus versiones, que mejoran la eficiencia según se trate de un servidor, un equipo personal o un sistema tiempo real. Así, en los modelos totalmente expulsivos (*full preemption*), un proceso corriendo en modo *kernel* puede ser expulsado de la CPU y sustituido por otro mediante una interrupción que invoque directamente al planificador. Según las restricciones el modelo *full preemtion* se puede implementar de diferentes maneras. Por ejemplo, en el actual modelo `PREEMPT_DESKTOP`, la expulsión se permite si el proceso no está en una sección crítica. En el caso extremo de un kernel tiempo real como el *patch*, incluso las rutinas de servicio a excepción

y las funciones diferidas (*softirqs*, *tasklets*, *workqueues*) son planificadas, no entrelazadas. En el modelo más restrictivo *partial preemption*, (`PREEMPT_NONE`) un proceso que se ejecuta en modo *kernel* sólo cederá la CPU mediante *invocación directa* al planificador (función `schedule`), o bien invocándolo cuando esté a punto de regresar a modo usuario (*invocación perezosa*). Es decir, una rutina de servicio a interrupción no puede causar un cambio de contexto de alto nivel (un cambio de proceso) ni directamente (invocando la función de planificación) ni indirectamente (causando un fallo de página).

Otro aspecto importante es la asignación de memoria física de la máquina virtual. En la Figura 2.4 se muestra el mapa de memoria virtual[8] de una máquina física. Al lanzar una máquina virtual KVM[9] realiza la asignación de las páginas del kernel de la máquina virtual a partir de la dirección `0x0000004000000000` del espacio de usuario de KVM. Por lo tanto los objetos del kernel huésped estarán mapeados en esas páginas.

El concepto importante aquí es que KVM es un *Hypervisor* tipo 1.7 (Sec. 2.1.2) y que por tanto se ejecuta como un proceso de usuario en el sistema operativo de la máquina física (anfitrión). En consecuencia, el sistema virtualizado (huésped) reside en el espacio de usuario de KVM. Para que el kernel de la máquina virtual siga contando con los mecanismos de protección de los que dispondría al ejecutarse sobre una máquina real -no virtualizada- es preciso un soporte específico a la virtualización.

2.2.1 *Módulos*

Los módulos [7] son componentes de código que se pueden enlazar (y desenlazar) dinámicamente con el kernel en ejecución, interactivamente o mediante un *shell script*. Permiten extender funcionalidades sin necesidad de reconfigurar, recompilar y enlazar estáticamente todos los componentes del kernel, generar una nueva imagen ejecutable, y reiniciar la máquina. Ello permite añadir o retirar *drivers* de dispositivos hardware o virtuales, utilidades de monitorización, soporte a diferentes tipos de sistemas de ficheros.

En este trabajo, todos los ataques se han realizado inyectando módulos modificados para producir comportamientos anómalos dentro del kernel. Para la detección de los ataques se utiliza la funcionalidad trazabilidad que provee el propio kernel, FTRACE (Sec. 2.2.2).

8 Catalin Marinas, Linux on AArch64 ARM 64-bit Architecture - https://events.static.linuxfound.org/images/stories/pdf/lcna_co2012_marinas.pdf

9 <ksrc>/Documentation/arm64/memory.txt.

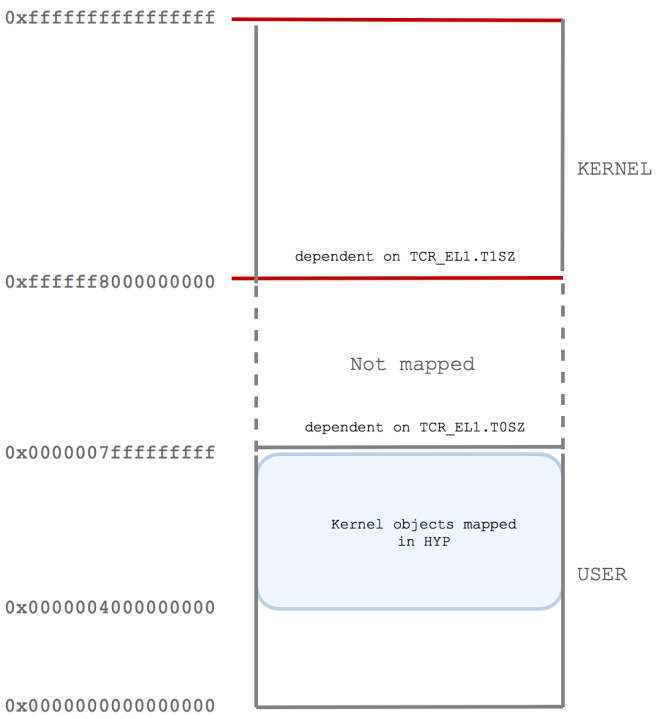


Figura 2.4: Mapa de memoria virtual *AArch64*

2.2.2 *Ftrace*

Ftrace[10] es un depurador que permite a los programadores de sistemas ver lo que ocurre en el kernel. También es utilizado para analizar las latencias en espacio de kernel y su respectivo rendimiento. Todo el control de Ftrace se realiza en el directorio `/sys/kernel/debug/tracing`. Dicho directorio posee el sistema de ficheros `tracefs`[11].

Tracefs es introducido en el kernel a partir de la versión 4.13, antes se utilizaba debugfs descartado por motivos de seguridad.

Para poder usar esta herramienta, es necesario modificar la configuración del kernel. Las opciones necesarias se encuentran en el Apéndice A.5.

10 https://elixir.bootlin.com/linux/v4.18.5/source/Documentation/trace/ftrace.rst
11 tracing: Add new file system tracefs - https://lwn.net/Articles/630526/

3 ENTORNO EXPERIMENTAL

La selección de las herramientas utilizadas en este proyecto se ha enfocado desde un primer momento al ahorro de costes y simplicidad de ejecución de las diferentes PRUEBAS DE CONCEPTO, Sec. 5.

3.1 HARDWARE

El proyecto ha sido implementado sobre la placa Hikey LeMaker[1] de 2 GB. Tiene un SoC Kirin 620 compuesto por una CPU ARM-A53 octa-core de 64 bits a 1,2 GHz y GPU Mali450-MP4. El conjunto de instrucciones es ARMv8.

Se ha elegido esta placa porque proporciona los recursos necesarios para ejecutar máquinas virtuales de manera sencilla y económica.

3.2 SOFTWARE

El binario EXTRACTOR DEL BACKTRACE (Sec. 4.1.1) está implementado en C y el script VERIFICADOR DE INTEGRIDAD (Sec. 4.1.2) en AWK. Para lanzar máquina virtuales de manera sencilla y rápida ha sido utilizado KVMTOOL, Sec. 3.2.1.

*AWK proviene de las iniciales de los apellidos de sus creadores: Alfred **A**ho, Peter **W**einberger y Brian **K**ernighan*

3.2.1 *kvmtool*

La herramienta *kvmtool*[2] permite lanzar instancias de máquinas virtuales en máquinas con pocos recursos. Es un *Hypervisor* mucho más sencillo que QEMU[3]/KVM o Xen, destinado a:

- Depuración del kernel Linux en máquinas virtuales.
- Ejecución de aplicaciones que utilicen SMP.
- Laboratorios experimentales para realizar pruebas de concepto.

Ambos aspectos son interesantes tanto para los desarrolladores de sistemas como para probar aplicaciones que utilizan múltiples CPUs. Soporta máquinas virtuales de la misma arquitectura que la máquina física, así como el parche *Real-Time*[4] del kernel.

kvmtool ha sido desarrollada por Pekka Enberg, Cyrill Gorcunov, Asias He, Sasha Levin y Prasad Josh con la ayuda de Avi Kivity e Ingo Molnar. Actualmente, los últimos commits de git son de Julien Thierry, Jean-Philippe Brucker y Andre Przywara

1 http://www.lemaker.org/product-hikey-index.html
2 Proyecto kvmtool - https://git.kernel.org/pub/scm/linux/kernel/git/will/kvmtool.git
3 Proyecto QEMU - https://www.qemu.org
4 Wiki *Real-Time* Linux - https://rt.wiki.kernel.org/index.php/Main_Page

También puede ejecutar aplicaciones de 32 bits en kernels de 64 bits. Para lanzar una instancia de una máquina virtualizada, primeramente hay que tener compilado un kernel con unas configuraciones que están en el Apéndice A.9.

4 METODOLOGÍA DE DADE

En primer lugar se define la caracterización de un objeto del kernel (Sec. 4.2) para después explicar la ARQUITECTURA implementada (Sec. 4.1) integrable en un *Hypervisor* como KVMTOOL.

DADE pretende detectar anomalías en la trazabilidad de asignación de memoria física a los objetos del kernel, por lo tanto se va a replicar la idea planteada por Hayoon Yi et al. [4] evitando la modificación del kernel de la máquina virtual. Además, se aportarán otros campos para realizar una caracterización más precisa del objeto. Esto ayudará en un futuro a aplicar técnicas de *Machine Learning* (SVM).

4.1 ARQUITECTURA

La Figura 4.1 muestra la arquitectura general del motor de detección de ataques organizado por la ubicación de cada componente. Hay dos elementos claramente diferenciados: EXTRACTOR DEL BACKTRACE (Sec. 4.1.1) que caracteriza el objeto y VERIFICADOR DE INTEGRIDAD (Sec. 4.1.2) que realiza la detección de anomalías entre dos objetos. Para extraer los elementos más importantes que genera FTRACE en el fichero de depuración[1], es necesario un programa *parser* que limpie e inserte los datos en el extractor. De esta manera, el verificador compara objeto a objeto, detectando si hay alguna incongruencia en los *hashes* generados con las direcciones de las funciones.

4.1.1 *Extractor del backtrace*

El extractor está formado por un pequeño *parser* encargado de leer el fichero generado por FTRACE que contiene la lista ordenada de funciones ejecutadas hasta llegar a `kmalloc`. También contiene las direcciones de las funciones, *offsets* y nombre del objeto. Con todos estos datos se puede realizar una caracterización del objeto de manera concisa. Las operaciones ejecutadas son las siguientes:

`kmalloc` reserva memoria física para objetos del kernel menores al tamaño de página

- En la secuencia de arranque se activa FTRACE que empieza a escribir en el fichero `trace`. Paso nº 1.
- **S**e monitoriza periódicamente el sistema de ficheros de la máquina virtual para comprobar si existe el fichero `trace`. Paso nº 2.

1 /sys/kernel/debug/tracing/trace

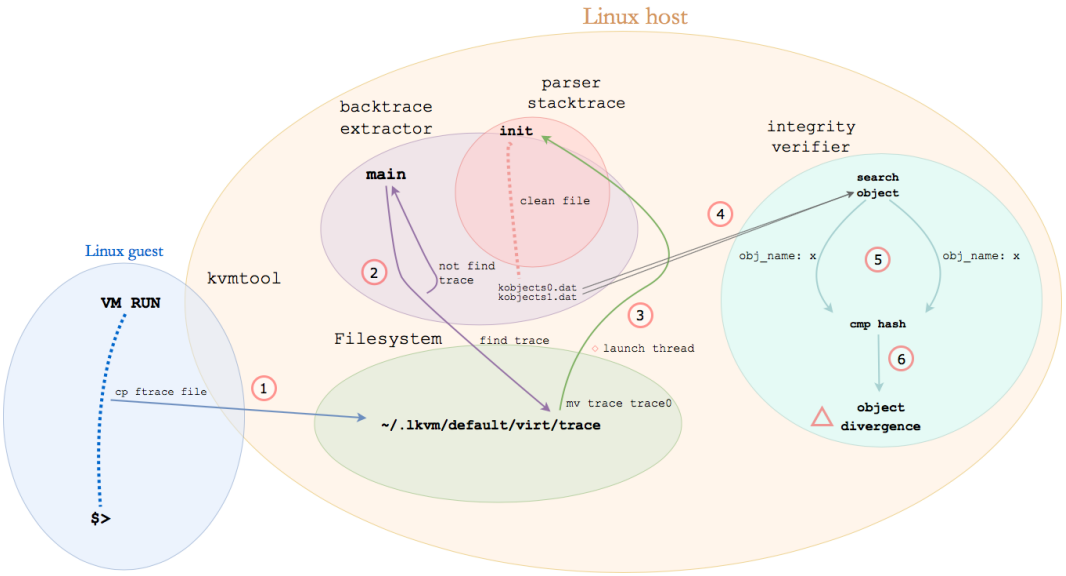


Figura 4.1: Esquema DADE

- Si existe dicho fichero, se lanza un hilo que ejecuta la función inicializadora del binario `parser_stacktrace` para extraer los datos más importantes, como por ejemplo el nombre del objeto, el *stacktrace* con sus direcciones y *offsets* y el valor de la dirección de memoria reservada. Paso nº 3.
- Después de que todos los datos hayan sido extraídos, se entregan al `backtrace_extractor` para que rellene la estructura `backtrace_container` y realice la firma *hash* de las direcciones de las funciones.
- Finalmente, se escribe toda la estructura `backtrace_container` en el fichero `kobjects{i}.data`.

El código completo tanto del extractor, como del *parser* se encuentran en los Apéndices A.6 y A.7 respectivamente.

4.1.2 *Verificador de integridad*

El verificador de integridad compara las firmas *hash* de los objetos de la siguente forma:

- La entrada del verificador son dos ficheros generados por el extractor (`kobjects{i}.data` y `kobjects{i+1}.data`). Paso nº 4.
- Al inicio, toma el nombre del primer fichero para que se puedan distinguir los vectores correspondientes de cada uno. Los vectores utilizarán como índice el nombre del objeto y como valor el *hash*.

- Se lee el nombre del objeto `obj_name` y se almacena en la variable `key`.
- Cuando se encuentra la palabra `hash`, se inserta como valor la firma y como índice el nombre del objeto. Paso nº 5.
- Cuando finaliza la lectura de los dos ficheros, hace un recorrido de los dos vectores buscando el objeto. En el caso que exista, compara sus *hashes* buscando una incongruencia.
- Si los *hashes* difieren entre sí, el ataque habrá sido detectado con éxito. Paso nº 6.

El código completo está en el Apéndice A.8.

4.2 CARACTERIZACIÓN DEL OBJETO

Para caracterizar un objeto, Hayoon Yi et al. [4] recurren a la identificación mediante *backtrace-naming*. Consiste en registrar la cadena ordenada de funciones activas (con sus direcciones) que se encuentran en la *stack* de un proceso o hilo. Para definir el contexto de un objeto, sólo hay que tener en cuenta el *backtrace* correspondiente a la cascada de invocaciones de las funciones de asignación de memoria.

Esta técnica se fundamenta en tres puntos:

1. La mayoría de objetos del kernel se crean a través de un conjunto específico de funciones que reservan memoria.
2. El contexto del kernel en el momento en que se crea un objeto refleja las características de este en tiempo de ejecución.
3. Modificar un objeto durante un ataque supone modificar la traza normal con la que se accede, por lo que si registramos el *nombre* (i.e. la traza de acceso) normal, posteriormente podemos compararla con trazas presuntamente modificadas por un ataque, diagnosticando el hecho.

Para extraer los *backtraces* Hayoon Yi et al. [4] insertan en las funciones de asignación de memoria del kernel una llamada al *Hypervisor* (i.e. una instrucción *hypercall*; vid. paravirtualización A.3). De este modo, cuando el kernel de las máquinas virtuales quiere reservar memoria, provocará una excepción y se saltará al *Hypervisor* para que realice los cambios necesarios y pueda extraer el *backtrace*. La Figura 4.2 ilustra este proceso.

Este mecanismo produce una degradación del sistema virtualizado, debido a que cada vez que se asigna memoria a los objetos del kernel,

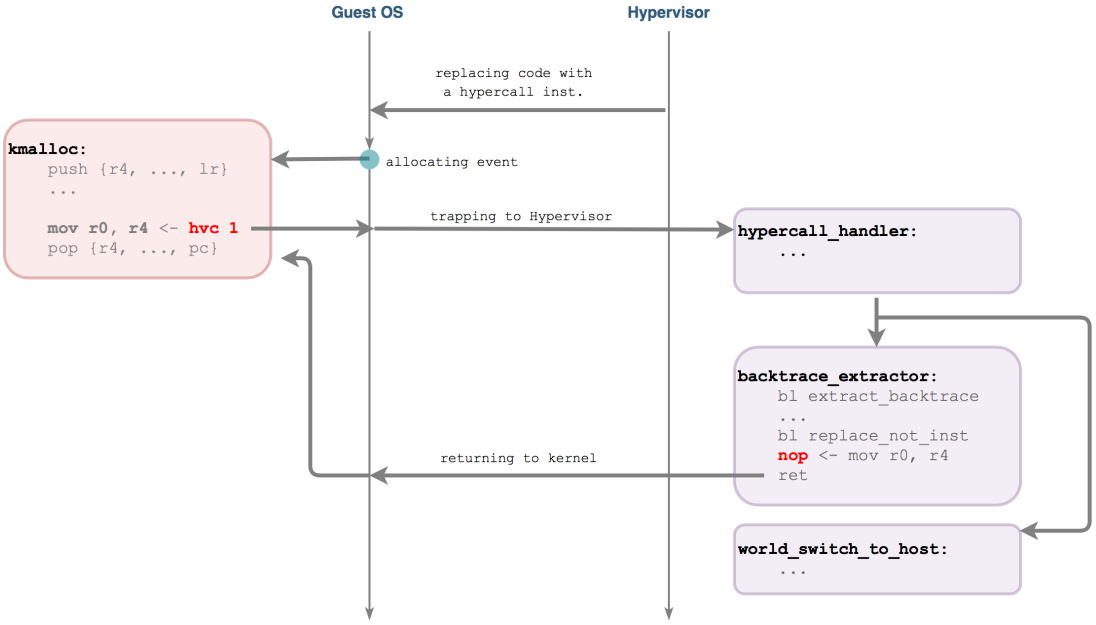


Figura 4.2: Diagrama de la solución con *hypercall*

se realiza un cambio al *Hypervisor*.

Una aportación de este proyecto es utilizar FTRACE (Sec. 2.2.2) para obtener la caracterización del objeto que obtienen modificando el kernel de la máquina virtual. Además, se consigue disminuir de manera sustancial las latencias que producen los cambios de contexto entre *Hypervisor*-VM y viceversa. Con FTRACE se obtiene el *stacktrace*[2], así como todas las direcciones[3] de las funciones. Las siguientes lineas ilustran como ejemplo el *backtrace* de la asignación de memoria para un objeto tipo *inode*:

```
2900   iptables-1283  [000] ....   176.885405: kmalloc: \
       call_site=ffff000089ff5f8 ptr=ffff80001bfb7100 \
       bytes_req=128 bytes_alloc=128 gfp_flags=GFP_KERNEL
2901   iptables-1283  [000] ....   176.885408: <stack trace>
2902   => alloc_inode+0x2c/0xb8 <ffff000082b7114>
2903   => new_inode_pseudo+0x20/0x60 <ffff000082b9398>
2904   => sock_alloc+0x28/0xd8 <ffff000089fe5d8>
2905   => __sock_create+0x5c/0x1b8 <ffff000089feeec>
2906   => SyS_socket+0x58/0xe8 <ffff000008a00420>
```

En primer lugar, ha sido realizado un estudio de los eventos disponibles para depurar y se ha habilitado la trazabilidad de la función `kmalloc`[4], encargada de reservar memoria física para objetos menores al tamaño de página en el kernel. Otra aportación realizada, además del modo de obtener las direcciones de las funciones, ha sido añadir

2 echo stacktrace >/sys/kernel/debug/tracing/trace_options
3 echo sym-addr >/sys/kernel/debug/tracing/trace_options
4 echo 1 >/sys/kernel/debug/tracing/events/kmem/kmalloc/enable

el *offset*[5] desde el cual se llama a la función. De este modo, la caracterización es más determinista (sufre menos variaciones entre ejecuciones) a la hora de detectar la modificación de las funciones.

5 echo sym-offset >/sys/kernel/debug/tracing/trace_options

5 PRUEBAS DE CONCEPTO

Los experimentos descritos en los apartados siguientes muestran cómo puede realizarse la detección de anomalías en las máquinas virtuales a partir de las aplicaciones EXTRACTOR DEL BACKTRACE (Sec. 4.1.1) y VERIFICADOR DE INTEGRIDAD (Sec. 4.1.2). Todas las pruebas de concepto realizadas son propias, originales de este proyecto, debido a que en los estudios previos de Hayoon Yi et al. [4] describen cómo realizan las pruebas pero no adjuntan la implementación.

5.1 ATAQUE NO. 1 LITERAL POOL SPOOFING

En este ataque se modifica el binario KERNEL LINUX (Sec. 2.2) mediante la inserción de una instrucción que carga una constante en un registro. En ARM los valores constantes que superan 16 bits no pueden ser cargados mediante una instrucción `mov`-inmediato[1] y son almacenados en la tabla de literales A.10. Al modificar la tabla de literales de un binario puede cambiarse las direcciones de salto y, por lo tanto, modificar el flujo de ejecución.

La tabla de literales está compuesta de datos codificados en una secuencia de instrucciones y se encuentra al final de cada función. Suele guardar direcciones de bloques básicos.

El módulo del kernel llamado *mutenroshi* A.11 crea un fichero en el directorio `/proc`[2]. Hay que leer dicho fichero para que se ejecute la función `kmalloc` y se guarde el árbol de llamadas mediante FTRACE. Forzamos la ejecución de la función invocante de `kmalloc` del siguiente modo:

```
$> cat /proc/mutenroshi
kamehameha!
```

El código que forma el módulo ilegítimo se implementa añadiendo la instrucción en ensamblador de la línea 28. Dicha instrucción carga el valor `3462381247` en el registro `w0`, por lo tanto el compilador creará la tabla de literales y ejecutará la carga de dicha constante, realizando un direccionamiento relativo a PC que apuntará a la tabla.

1 PC-relative modes (load-literal) - http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.den0024a/CJAIFJII.html

2 En dicho directorio está montado el pseudo-sistema de ficheros `proc` que proporciona información y control sobre el hardware y kernel (procesos, dispositivos E/S, módulos, CPU).

```
26 static int proc_open(struct inode *inode,
                       struct file *file)
27 {
28     __asm__("LDR w0, =3462381247");
29     return single_open(file, proc_show, NULL);
30 }
```

En el Cuadro 5.1 se muestran los dos *stacktraces*: la función del módulo `proc_open` invoca a la función `single_open` que, finalmente, invoca `kmalloc`. Obsérvese que la función `proc_open` del módulo posee un *offset* igual a `0x28/0x40` relativo a PC. Sin embargo, al insertar la instrucción que carga un valor aleatorio en la tabla de literales, el *offset* cambia a `0x2c/0x40`. Los *stacktraces* formateados que genera EXTRACTOR DEL BACKTRACE 4.1.1 se encuentran en los Apéndices A.13 y A.14 respectivamente.

GOOD_MUTENROSHI.KO	BAD_MUTENROSHI.KO
elo_svc_naked+0x34/0x38	elo_svc_naked+0x34/0x38
SyS_openat+0x3c/0x4c	SyS_openat+0x3c/0x4c
do_sys_open+0x14c/0x214	do_sys_open+0x14c/0x214
do_filp_open+0x6c/0xe8	do_filp_open+0x6c/0xe8
path_openat+0x518/0xf78	path_openat+0x518/0xf78
vfs_open+0x5c/0x8c	vfs_open+0x5c/0x8c
do_dentry_open+0x22c/0x330	do_dentry_open+0x22c/0x330
proc_reg_open+0x9c/0x13c	proc_reg_open+0x9c/0x13c
proc_open+0x28/0x40	proc_open+0x2c/0x40
single_open+0x78/0xc0	single_open+0x78/0xc0

Cuadro 5.1: *Stacktrace* literal pool spoofing

Para comprender mejor las implicaciones hay que considerar el código ensamblador del Cuadro 5.3. La instrucción nº 18 del módulo ilegítimo carga en el registro `w0`[3] el literal almacenado en la posición `proc_show+0x48`. Por lo tanto, provoca que la dirección de retorno de `single_open` sea superior en *4* con respecto al módulo legítimo.

Mediante el EXTRACTOR DEL BACKTRACE se obtienen los *hashes* del Cuadro 5.2 correspondientes al objeto `single_open`.

KERNEL OBJECT	HASH MD5
1. single_open	90ffd3626bbdc6a9b964909f2ba0fb8c
2. single_open	7fb1c3bd6212987c57443729982940b4

Cuadro 5.2: *Hashes* objeto single_open

3 `w0` - registro de 32 bits en ARMv8.

```
good_mutenroshi.ko: file format
     elf64-littleaarch64

Disassembly of section .text:

<proc_open>:
0: stp x29, x30, [sp,#-32]!
4: mov x29, sp
8: str x19, [sp,#16]
c: mov x0, x30
10: mov x19, x1
14: bl  0 <_mcount>
18: ldr x1, 38 <proc_open+0x38>
1c: mov x2, #0x0
20: mov x0, x19
24: bl  0 <single_open>
28: ldr x19, [sp,#16]
2c: ldp x29, x30, [sp],#32
30: ret
34: nop
...
```

```
bad_mutenroshi.ko: file format
    elf64-littleaarch64

Disassembly of section .text:
<proc_open>:
0: stp x29, x30, [sp,#-32]!
4: mov x29, sp
8: str x19, [sp,#16]
c: mov x0, x30
10: mov x19, x1
14: bl  0 <_mcount>
18: ldr w0, 88 <proc_show+0x48>
1c: ldr x1, 38 <proc_open+0x38>
20: mov x2, #0x0
24: mov x0, x19
28: bl  0 <single_open>
2c: ldr x19, [sp,#16]
30: ldp x29, x30, [sp],#32
34: ret
...
<proc_show>:
...
88: .word   0xce5fbebf
8c: nop
```

Cuadro 5.3: Código ensamblador módulos

El VERIFICADOR DE INTEGRIDAD detectará el ataque de manera satisfactoria debido a que los *hashes* del objeto `single_open` son incongruentes.

5.2 ATAQUE NO. 2 UNRELIABILITY BINARY SCRIPT FORMAT

Este ataque consiste en la inserción de un procedimiento malicioso en la función `load_script` del módulo encargado de ejecutar el formato *script* en el kernel (`binfmt_script`). La finalidad de este ataque es saltar al procedimiento insertado cada vez que se realice la ejecución de un programa *script*, modificando el flujo de ejecución y causando degradaciones en la máquina.

La función `load_script` realiza la validación de los caracateres #!

Según Daniel P. Bovet et al. [2], el kernel Linux soporta diversos formatos de ficheros ejecutables (ELF, COFF). También permite la ejecución de binarios pertenecientes a otros sistemas operativos (MS-DOS, familia Unix BSD). Los formatos son reconocidos por un *número mágico* codificado en los primeros 128 bytes de todos los ficheros.

La información acerca de los formatos y estructuras del kernel está ampliada en A.15.

El módulo ilegítimo con la llamada al procedimiento se puede observar en la línea 98. El código completo se encuentra en el Apéndice A.17.

```
17 static int load_script(struct linux_binprm *bprm)
18 {
        ...
98      get_max_files();
99      /*
100      * OK, now restart the process with the
           interpreter's dentry.
101      */
102     file = open_exec(i_name);
103     if (IS_ERR(file))
104         return PTR_ERR(file);

        ...
121     return 0;
122 }
```

En el Cuadro 5.4 se recogen los *stacktraces* que corresponden a la ejecución de un programa *script*, tanto del módulo sin modificar como el módulo malicioso. Los *stacktraces* formateados que genera EXTRACTOR DEL BACKTRACE se encuentran en A.16 y A.19.

GOOD_BINFMT_SCRIPT.KO	BAD_BINFMT_SCRIPT.KO
elo_svc_naked+0x34/0x38	elo_svc_naked+0x34/0x38
SyS_execve+0x38/0x4c	SyS_execve+0x38/0x4c
link_path_walk+0x47c/0x4d8	do_execve+0x44/0x54
do_execveat_common.isra.37+0x494/0x63c	do_execveat_common.isra.37+0x494/0x63c
search_binary_handler+0xb0/0x204	search_binary_handler+0xb0/0x204
load_script+0x1fc/0x218	load_script+0x1f8/0x218
search_binary_handler+0xb0/0x204	search_binary_handler+0xb0/0x204

Cuadro 5.4: *Stacktrace* unreliability binary script format

La función `load_script` del módulo legítimo posee un *offset* de `0x1fc/0x218` mientras que en el módulo ilegítimo es `0x1f8/0x218`. Estos *offsets* aparecen de este modo porque al añadir código nuevo y compilar por defecto con `-O2`, el compilador reordena el código, generando un *offset* menor. En el primer ataque, como era únicamente una instrucción de código, el compilador no ha realizado ninguna otra optimización. En los Apéndices A.20 y A.21 están los códigos en ensamblador del módulo legítimo e ilegítimo respectivamente.

Conviene notar que, los *hashes* 5.5 de las direcciones del objeto `search_binary_handler` son dispares.

KERNEL OBJECT	HASH MD5
1. search_binary_handler	8ebd6a46aa779b90cd651b708a475226
2. search_binary_handler	17e39e204a228aff9da4c2d657b34a68

Cuadro 5.5: *Hashes* objeto search_binary_handler

5.3 ATAQUE NO. 3 GARBLING NETFILTER IPV4

El ataque propuesto está basado en la modificación del módulo `ip_tables` perteneciente al *framework* Netfilter. Este módulo es uno de los componentes que realiza el filtrado de los paquetes *ipv4* del sistema. El objetivo del ataque es intentar cambiar la IP que el usuario quiere filtrar. De esta manera una IP maliciosa no sería añadida al *firewall*.

Las consecuencias de introducir este *malware* son graves, debido a que el usuario introduce una IP concreta y el código malicioso cambia dicha IP, realizando una operación `NOT` bit a bit. Para solucionar el problema, en primer lugar, el usuario debería saber las estructuras de Netfilter y qué módulo está fallando. A priori, no puede ver el código, por lo que debería de proceder a un desensamblado del mismo.

Una de las partes de Netfilter [3, 6] es la tabla `FILTER`, utilizada para el filtrado de paquetes. Esta tabla únicamente filtra paquetes, no los modifica. Para poder utilizar `FILTER`, hay que cargar tres módulos, en el siguiente orden debido a la dependencia que hay entre ellos:

También existen otras tablas: `NAT`, `MANGLE`, `CONNTRACK`.

1. **x_tables**[4] - realiza el filtrado génerico basado en tabla independiente del protocolo (*ipv4*, *ipv6*, *arp*, *eb*).
2. **ip_tables**[5] - realiza el registro de las reglas en la tabla *ipv4* para la herramienta `iptables`, encargada de la configuración por la línea de comandos en espacio de usuario.
3. **iptable_filter** [6] - inicializa la función que salta al módulo **ip_tables** para asignar memoria y registrar la tabla. También, inicializa los *hooks* `LOCAL_IN`, `LOCAL_OUT` y `FORWARD` necesarios para que el paquete *ipv4* sea filtrado.

4 <ksrc>/net/netfilter/x_tables.c
5 <ksrc>net/ipv4/netfilter/ip_tables.c
6 <ksrc>/net/ipv4/netfilter/iptables_filter.c

La implementación de la *rootkit* está dentro de la función `translate_table`, concretamente, en las líneas 740-745.

```
734     /* Finally, each sanity check must pass */
735     i = 0;
736     xt_entry_foreach(iter, entry0, newinfo->size) {
737         ret = find_check_entry(iter, net, repl->name,
        repl->size,
738         &alloc_state);
739
740         if (((iter->ip.src.s_addr >> 24U) & 255) != 0 &&
741             ((iter->ip.src.s_addr >> 16U) & 255) != 0 &&
742             ((iter->ip.src.s_addr >> 8U) & 255) != 0 &&
743              (iter->ip.src.s_addr & 255) != 0) {
744             iter->ip.src.s_addr = ~iter->ip.src.s_addr;
745         }
746
747         if (ret != 0)
748             break;
749         ++i;
750     }

751
752     if (ret != 0) {
753         xt_entry_foreach(iter, entry0, newinfo->size) {
754             if (i-- == 0)
755                 break;
756             cleanup_entry(iter, net);
757         }
758         return ret;
759     }
760
761     return ret;
762 out_free:
763     kvfree(offsets);
764     return ret;
765 }
766
```

El principal objetivo de esta función es realizar una sanitización de la información almacenada en la dirección `entry0` mediante la estructura `xt_table_info` apuntada por la variable `newinfo`. En el Cuadro 5.6 están los *stacktraces* obtenidos.

En el caso del módulo ilegítimo el *offset* `0x1ac/0x238` menor que `0x1ac/0x248`, del módulo legítimo. Ocurre lo mismo que en el segundo ataque, al compilar por defecto con `-O2`, el compilador reordena el código aplicando las optimizaciones necesarias. En los Apéndices A.26 y A.27 están los códigos en ensamblador del módulo legítimo e ilegítimo respectivamente. Los *hashes* del objeto `__do_replace` son mostrados en el Cuadro 5.7.

GOOD_IP_TABLES.KO	BAD_IP_TABLES.KO
elo_svc_naked+0x34/0x38	elo_svc_naked+0x34/0x38
SyS_setsockopt+0x74/0xd0	SyS_setsockopt+0x74/0xd0
sock_common_setsockopt+0x54/0x68	sock_common_setsockopt+0x54/0x68
raw_setsockopt+0x70/0xb0	raw_setsockopt+0x70/0xb0
ip_setsockopt+0x7c/0xa8	ip_setsockopt+0x7c/0xa8
nf_setsockopt+0x64/0x88	nf_setsockopt+0x64/0x88
do_ipt_set_ctl+0x1ac/0x248	do_ipt_set_ctl+0x1ac/0x238
__do_replace+0xe4/0x250	__do_replace+0xe4/0x250

Cuadro 5.6: *Stacktrace* garbling Netfilter *ipv4*

KERNEL OBJECT	HASH MD5
1. __do_replace	1f827b15780378d9e4f6cbfb5f4e0455
2. __do_replace	0910c045af680e7f4b5b422288c498f7

Cuadro 5.7: *Hashes* objeto __do_replace

6 CONCLUSIONES Y TRABAJO FUTURO

Para replicar el diseño de DADE se ha tenido que hacer un estudio extenso sobre el funcionamiento de la virtualización y los sistemas operativos. A pesar de que la técnica no estaba explicada de forma precisa, el comportamiento ha sido replicado.

La idea novedosa de utilizar FTRACE (Sec. 2.2.2) para generar *backtraces* a efectos de detección de ataques en kernels de máquinas virtuales ha resultado satisfactoria. Se ha realizado un prototipo explotando esta técnica que consigue extraer los datos para la detección de anomalías sin modificar el kernel de las máquinas virtuales. Adicionalmente, se han preparado tres ataques originales, utilizados como pruebas de concepto que muestran la viabilidad y utilidad de las herramientas diseñadas. Una de ellas, ATAQUE NO. 3 GARBLING NETFILTER IPV4 (Sec. 4.1.1) realmente útil para provocar una degradación importante en la filtración de los paquetes.

El prototipo desarrollado podría ser extendido aplicando las siguientes ideas.

- Realizar una caracterización de la liberación de objetos con `kfree`.
- Contextualizar la asignación de memoria de los objetos del kernel en función de la aplicación (Apache, MongoDB, PostFix, Docker).
- Aplicar técnicas de *Machine Learning* (SVM) a los ficheros formateados con el extractor.
- Estudiar otras funciones de asignación de memoria del kernel: `kmalloc_node`, `kmem_cache_alloc`, `mm_page_alloc`.

ANEXOS

A.1 DIAGRAMA GANTT DEL PROYECTO

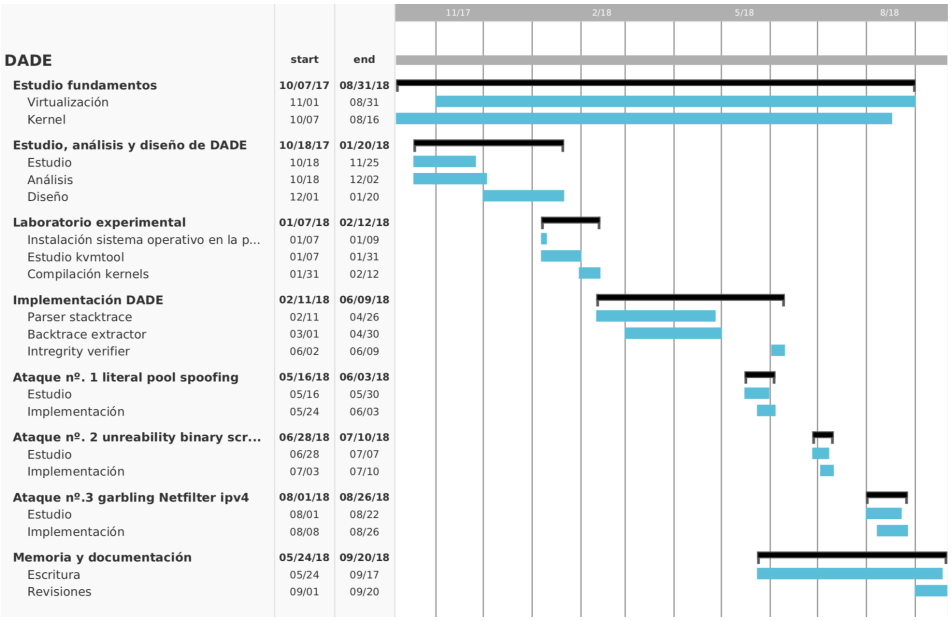


Figura A.1: Diagrama de Gantt

A.2 EJECUCIÓN DIRECTA + TRAP&EMULACIÓN

La técnica de *Ejecución directa + trap&emulación* es utilizada en los *Hypervisores* de tipo 1, como por ejemplo, VMware Exsi. Se basa en la siguiente idea. La máquina virtual se está ejecutando en un modo de privilegio menor que el *Hypervisor*, por lo tanto puede ejecutar un pequeño subconjunto de instrucciones. En el momento en que la máquina virtual quiera ejecutar una instrucción *sensible*, ya sea de *control* o de *comportamiento*, el flujo del programa debería pasar al *Hypervisor* mediante una excepción (*trap*). Sin embargo, aparece un problema con algunas instrucciones que, debido a su ISA, son *sensibles* pero *no privilegiadas* (son instrucciones *duales*). Es decir, puede ocurrir que una instrucción ejecutándose en modo *usuario* cambie recursos del sistema sin que el *Hypervisor* tenga constancia, ya que esa instrucción (no es privilegiada) está dentro del subconjunto de instrucciones que ofrece el modo *usuario*. Esto es un problema de la ISA.

Este comportamiento incumple por tanto el Teorema 1 formulado por Popek et al. [10] para la implementación de un *Hypervisor*. La solución al problema de las instrucciones *duale*s era realizar una traducción (estática o dinámica) del binario, sustituyendo en tiempo de ejecución las instrucciones problemáticas por llamadas al *Hypervisor*. Un ejemplo de instrucción *dual* es:

- **Pop Stack into Flags Register** [9] (POPF, Intel IA-32). Escribe en los flags del registro de estado el dato extraído de la pila. En modo *usuario* sobreescribe todos los flags excepto IF.

La instrucción anterior suponía un problema ya que únicamente, en caso de *x86*, había cuatro anillos (o modos) de privilegio.

- Anillo 3 - Ejecución de aplicaciones
- Anillo 2 y 1 - Dispositivos drivers.
- Anillo 0 - Ejecución del kernel.

Cuando estaban ejecutándose las máquinas virtuales, sus aplicaciones tenían los privilegios del anillo 3, mientras que el kernel de la máquina virtual estaba en el anillo 1 y el *Hypervisor* en anillo 0. Por lo tanto, todas las instrucciones *privilegiadas* provocaban una excepción, sin embargo, el *Hypervisor* no tenía constancia de las instrucciones *sensibles no privilegiadas* debido al anillo en el cual se estaba ejecutando el kernel del huésped que tenía disponible ese tipo de instrucciones. La Figura A.2[1] explica las transiciones que se producen en este tipo de casos.

Las extensiones de virtualización incorporadas por Intel (*VT-x*) resuelven el problema añadiendo un nuevo anillo denominado *Root mode*. Este anillo o modo tiene más privilegios que los demás y es el utilizado por el *Hypervisor*. Por lo tanto cualquier instrucción *sensible* (sea privilegiada o no) ejecutada en el kernel de la máquina virtual provocará una excepción atendida por el *Hypervisor*, que está en un anillo de privilegio mayor. Esto es así porque al añadir un nuevo anillo, el kernel de la máquina virtual está en anillo 0 y por tanto su conjunto de instrucciones está restringido ya que las sensibles únicamente se puede ejecutar en *Root mode*. La Figura A.3 muestra los pasos detallados de un procesador con extensiones de virtualización.

A.3 PARAVIRTUALIZACIÓN

La *paravirtualización* consiste en modificar el fuente del kernel de la máquina virtual (o huésped) sustituyendo todas las instrucciones

1 Yeh-Ching Chung, Professor Ph.D. Syracuse University - http://www.cs.nthu.edu.tw/~ychung/

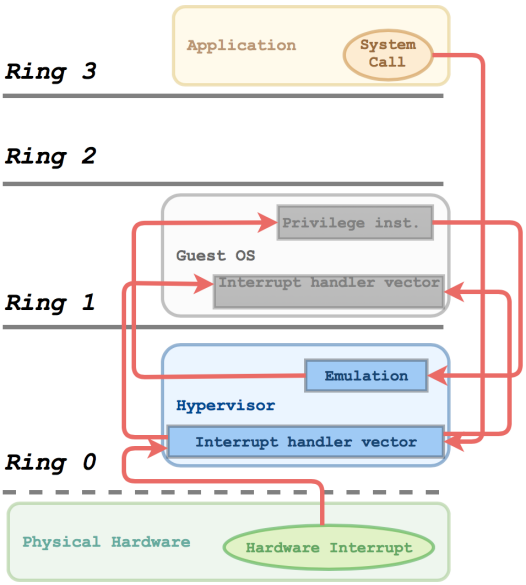


Figura A.2: Transiciones *Hypervisor*-VM en procesador sin virtualización

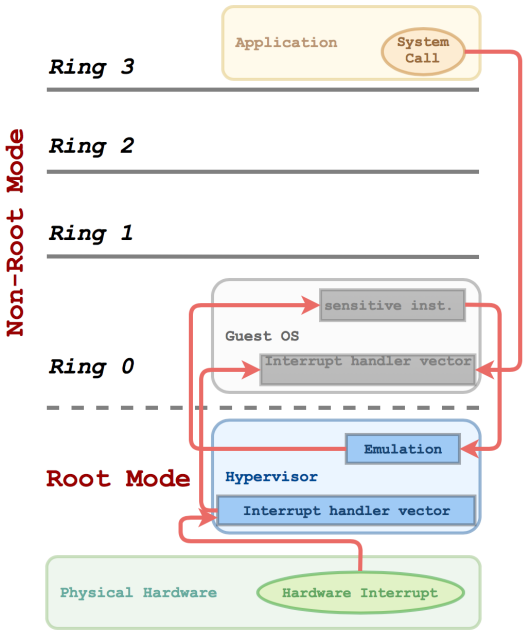


Figura A.3: Transiciones *Hypervisor*-VM en procesador con virtualización

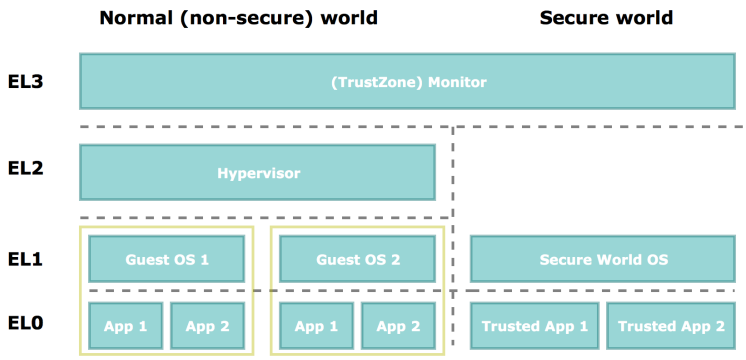


Figura A.4: Niveles de privilegio *AArch64*

sensibles (*control* y de *comportamiento*) por llamadas al *Hypervisor*. Estas llamadas reciben el nombre *hypercalls*. La máquina física no necesita tener un sistema operativo al uso, es decir, el *Hypervisor* está ejecutándose en *bare metal* e implementa un microkernel para la gestión de recursos de la máquina virtual. Con esta solución, el hardware no necesita extensiones de virtualización, como *VT-x* x86 o *TrustZone* de ARM.

A.4 NIVELES DE PRIVILEGIO *aarch64*

Convien recordar que, en modo kernel se tiene acceso a todo el repertorio de instrucciones, mientras que en modo usuario solo se tiene acceso a un subconjunto.

En relación con el Apéndice A.2, ARM en las versiones anteriores a la nº 7[2], únicamente tenía dos modos de privilegio: modo *kernel* (PL1) y modo *usuario* (PL0). En el modo *kernel* (o *supervisor*), el procesador podía ejecutar cualquier instrucción de E/S, acceso a memoria o cambio de la palabra de estado. El sistema operativo de una máquina física se ejecuta en modo *kernel*, mientras que las aplicaciones se ejecutan en modo *usuario*.

Conviene notar que un *Hypervisor* ejecutándose en modo *kernel* en la máquina física, tiene un problema al lanzar el sistema operativo huésped como proceso de usuario, ya que las instrucciones *sensibles* no privilegiadas no serán atrapadas, como se ha explicado en el Apéndice A.2). Para ello, en ARMv8 con soporte para virtualización se extendieron los niveles de privilegio. La Figura A.4[3] muestra los distintos niveles de privilegios.

`EL3` es el mayor nivel de privilegio, por lo tanto tiene acceso a todo el repertorio de instrucciones, mientras que `EL0` es el menor. Al haber más niveles de privilegio, el sistema operativo huésped se ejecuta en EL1 y tendrá acceso a un subconjunto de instrucciones. Por lo tanto, si quiere ejecutar una instrucción *sensible* en ese nivel de privilegio,

2 ARMv7-A tiene modo *Hypervisor* (PL2)

3 https://events.static.linuxfound.org/images/stories/pdf/lcna_co2012_marinas.pdf

saltará una excepción al *Hypervisor*, que tiene el máximo privilegio y podrá ejecutar ese tipo de instrucciones.

A.5 FICHERO CONFIGURACIÓN KERNEL - FTRACE

```
CONFIG_PSTORE_FTRACE=y
CONFIG_NOP_TRACER=y
CONFIG_HAVE_FUNCTION_TRACER=y
CONFIG_HAVE_FUNCTION_GRAPH_TRACER=y
CONFIG_HAVE_DYNAMIC_FTRACE=y
CONFIG_HAVE_FTRACE_MCOUNT_RECORD=y
CONFIG_HAVE_SYSCALL_TRACEPOINTS=y
CONFIG_HAVE_C_RECORDMCOUNT=y
CONFIG_TRACER_MAX_TRACE=y
CONFIG_TRACE_CLOCK=y
CONFIG_RING_BUFFER=y
CONFIG_EVENT_TRACING=y
CONFIG_CONTEXT_SWITCH_TRACER=y
CONFIG_RING_BUFFER_ALLOW_SWAP=y
CONFIG_TRACING=y
CONFIG_GENERIC_TRACER=y
CONFIG_TRACING_SUPPORT=y
CONFIG_FTRACE=y
CONFIG_FUNCTION_TRACER=y
CONFIG_FUNCTION_GRAPH_TRACER=y
CONFIG_IRQSOFF_TRACER=y
CONFIG_PREEMPT_TRACER=y
CONFIG_SCHED_TRACER=y
CONFIG_HWLAT_TRACER=y
CONFIG_FTRACE_SYSCALLS=y
CONFIG_TRACER_SNAPSHOT=y
CONFIG_TRACER_SNAPSHOT_PER_CPU_SWAP=y
CONFIG_BRANCH_PROFILE_NONE=y
CONFIG_STACK_TRACER=y
CONFIG_BLK_DEV_IO_TRACE=y
CONFIG_UPROBE_EVENTS=y
CONFIG_PROBE_EVENTS=y
CONFIG_DYNAMIC_FTRACE=y
CONFIG_FUNCTION_PROFILER=y
CONFIG_FTRACE_MCOUNT_RECORD=y
CONFIG_TRACING_MAP=y
CONFIG_HIST_TRIGGERS=y
CONFIG_TRACEPOINT_BENCHMARK=y
CONFIG_TRACE_EVAL_MAP_FILE=y
CONFIG_TRACING_EVENTS_GPIO=y
```

A.6 BACKTRACE_EXT.C

```
1  /*
2   * Project name: DADE - Data Anomaly Detection Engine
3   * Project managers: José Luis Briz and Darío Suárez
4   * Computer Architecture Group - University of Zaragoza
5   * Author: Luis Fueris Martín
6   * Date: 3 july 2018
7   * File name: backtrace_ext.c
8   */
9
10
11 #include <stdio.h>
12 #include <stdlib.h>
13 #include <unistd.h>
14 #include <linux/list.h>
15 #include <string.h>
16 #include <linux/errno.h>
17 #include <openssl/md5.h>
18 #include <time.h>
19 #include <pthread.h>
20
21 #include "backtrace_ext.h"
22 #include "parser_stacktrace.h"
23
24 #define DEBUG       0
25
26 static int kernel_obj_num[MAX_THREADS];
27 static int nfile = 0;
28 pthread_mutex_t nfile_lock;
29
30
31 /*
32  * debug__print_list: debug purposes, print kernel list which contains addr
33  * and function name.
34  *
```

```
35  * @result_list: the stacktrace list poiinter
36  *
37  */
38 static void debug__print_list(const struct list_head *result_list)
39 {
40     struct backtrace *ptr;
41     int i;
42
43     i = 0;
44     list_for_each_entry(ptr, result_list, list) {
45         /* when assign [btr_list = *result_list], the list head act as
46          * circular linked list. */
47         if (ptr->func_name == NULL || ptr->addr == NULL) {
48             break;
49         }
50
51         printf("[%d] %s -> %s\n", i, ptr->addr, ptr->func_name);
52         printf("\t  :\n");
53         printf("\t  :\n");
54         i = i + 1;
55     }
56
57     printf("\t  := Kernel object has been allocated.\n\n");
58 }
59
60
61 /*
62  * debug__print_dock: debug purposes, print dock structure
63  *
64  */
65 static void debug__print_dock(struct backtrace_container dock)
66 {
67     printf("dock.obj_name:  \n\t%s\n", dock.obj_name);
68     printf("dock.obj_value (ptr): \n\t%s\n", dock.obj_value);
69     printf("dock.hash: \n\t%s\n", dock.hash);
70 }
71
72
73 /*
74  * backtrace_ext__write_data: save stacktrace in [fd] descriptor.
75  *
76  * @fd: the kernel object bbdd descriptor pointer
77  *
78  */
79 static void backtrace_ext__write_data(struct backtrace_container dock,
80                                       FILE *fd, int id)
81 {
82     struct backtrace *ptr;
83     int i;
84
85     fprintf(fd, "\nKernel object No. %d:\n\n", kernel_obj_num[id]);
86     fprintf(fd, " obj_name: %s\n", dock.obj_name);
87     fprintf(fd, " obj_value: %s\n", dock.obj_value);
88     fprintf(fd, " hash: %s\n", dock.hash);
89     fprintf(fd, "\n\t\t\t\t\t stacktrace\n");
90     fprintf(fd, "\t\t\t\t\t(kernel trap)\n\n");
91
92     i = 0;
93     list_for_each_entry(ptr, &dock.btr.list, list) {
94         if (ptr->func_name == NULL || ptr->addr == NULL) {
95             break;
96         }
97
98         fprintf(fd, " [%d] %s - %s\n", i, ptr->func_name, ptr->addr);
99         i = i + 1;
100    }
101
102    fprintf(fd, "-------------------------------------------------------------"
103                "--------------------\n");
104    if (dock.obj_name != NULL) {
105        free(dock.obj_name);
106    }
107
108    if (dock.obj_value != NULL) {
109        free(dock.obj_value);
110    }
111 }
112
113
114 /*
115  * backtrace_ext__generate_hash: generate signature with function addresses.
116  *
117  * @dock: the dock container pointer
118  *
119  */
120 static void backtrace_ext__generate_hash(struct backtrace_container *dock)
121 {
122     /* 16 (MD5_DIGEST_LENGTH) * 8b (char) = 128b */
123     unsigned char digest[MD5_DIGEST_LENGTH];
124     /* 128b / 4b = 32 hex chars */
```

```
125     char hash[MD5_DIGEST_LENGTH * 2 + 1];
126     char *all_addr;
127     struct backtrace *ptr;
128     int i;
129
130     all_addr = malloc(sizeof(char) * BUFSIZE);
131     list_for_each_entry(ptr, &dock->btr.list, list) {
132         if (ptr->addr == NULL ||
133             ptr->func_name == NULL) {
134             break;
135         }
136
137         strncat(all_addr, ptr->addr, sizeof(char) * (ADDR_LEN + 2));
138     }
139
140     MD5((unsigned char*)all_addr, sizeof(char) * strlen(all_addr),
141                                                 (unsigned char*)&digest);
142     /* 02x means if your provided value is less than two digits then 0
143        will be prepended. */
144     for(i = 0; i < MD5_DIGEST_LENGTH; i++) {
145         sprintf(&hash[i * 2], "%02x", (unsigned int)digest[i]);
146     }
147
148     strncpy(dock->hash, hash, MD5_DIGEST_LENGTH * 2);
149
150     if (all_addr != NULL) {
151         free(all_addr);
152     }
153 }
154
155
156 /*
157  * backtrace_ext__copy_obj_name: extract kernel object name and put it in
158  * [dock] struct.
159  *
160  * @dock: the dock container pointer
161  * @list_length: list size
162  *
163  */
164 static void backtrace_ext__copy_obj_name(struct backtrace_container *dock,
165                                          const int list_length)
166 {
167     struct backtrace *ptr;
168     size_t obj_name_len;
169     int i;
170
171     i = 0;
172     list_for_each_entry(ptr, &dock->btr.list, list) {
173         if (i == (list_length - FUNC_OBJ_NAME)) {
174
175             obj_name_len = strlen(ptr->func_name);
176             dock->obj_name = malloc(sizeof(char) * (obj_name_len + 1));
177             dock->obj_name[obj_name_len] = '\0';
178             strncpy(dock->obj_name, ptr->func_name, obj_name_len);
179
180             break;
181         }
182
183         i = i + 1;
184     }
185 }
186
187
188 /*
189  * backtrace_ext__save_stacktrace: save stacktrace in [bbdd_fd] desriptor.
190  * This procedure is called from parser stacktrace module, In particular
191  * from [parser_stacktrace__allocate_kobject].
192  *
193  * @result_list: the stacktrace list pointer
194  * @list_length: list size
195  * @obj_value: it is the [ptr] field in Ftrace file
196  * @obj_value_len: obj value size (but it has got fixed size? check it)
197  * @bbdd_fd: the kernel object bbdd descriptor pointer
198  *
199  */
200 int backtrace_ext__save_stacktrace(const struct list_head *result_list,
201                                    const int list_length, const char* obj_value,
202                                    const int obj_value_len, FILE *bbdd_fd,
203                                    const int id)
204 {
204 {
205     struct backtrace_container dock;
206     kernel_obj_num[id] = kernel_obj_num[id] + 1;
207
208     INIT_LIST_HEAD(&dock.btr.list);
209     dock.hash[MD5_DIGEST_LENGTH * 2] = '\0';
210
211     dock.obj_value = malloc(sizeof(char) * (obj_value_len - 1));
212     dock.obj_value[obj_value_len - 2] = '\0';
213     strncpy(dock.obj_value, obj_value, obj_value_len - 2);
```

```
214
215     dock.btr.list = *result_list;
216     backtrace_ext__copy_obj_name(&dock, list_length);
217     backtrace_ext__generate_hash(&dock);
218
219     backtrace_ext__write_data(dock, bbdd_fd, id);
220
221     return 0;
222 }
223
224
225 /*
226  * backtrace_ext__init: create kobject[0-9]+.data file, open Ftrace file and
227  * call parser stacktrace module [parser_stacktrace__init] to clean Ftrace
228  * file.
229  *
230  * @filename: the Ftrace file pointer
231  *
232  */
233 static void *backtrace_ext__init(void *filename)
234 {
235     pthread_mutex_lock(&nfile_lock);
236     int id = nfile;
237     const char bbdd_name[BBDD_NAME_LEN] = { 'k', 'o', 'b', 'j', 'e', 'c',
238                     't', 's', nfile + 'o', '.', 'd', 'a', 't', 'a', '\0' };
239     pthread_mutex_unlock(&nfile_lock);
240
241     int ret;
242     time_t local_time;
243     char *local_time_str;
244     FILE *bbdd_fd;
245     pthread_t tid;
246
247     local_time = time(NULL);
248     local_time_str = ctime(&local_time);
249
250     bbdd_fd = fopen(bbdd_name , "w");
251     if (bbdd_fd == NULL) {
252         printf("File not open in backtrace_ext__init\n");
253         pthread_exit((void*) -EBADR);
254     }
255
256     fprintf(bbdd_fd, "Date %s\n", local_time_str);
257     tid = pthread_self();
258
259     kernel_obj_num[id] = 0;
260     ret = parser_stacktrace__init(filename, bbdd_fd, id);
261     if (ret < 0) {
262         printf("Error in parser extractor module\n");
263         fclose(bbdd_fd);
264     }
265
266     fclose(bbdd_fd);
267     pthread_exit(NULL);
268 }
269
270
271 /*
272  * backtrace_ext__messages: show init messages.
273  *
274  */
275 static void backtrace_ext__messages(void)
276 {
277     printf("\n\t\t\tData Anomaly Detection Engine (DADE)\n\n");
278     printf("\t\tAuthors: Jose Luis Briz, Luis Fueris and Dario "
279                                                     "Suarez\n\n\n");
280     printf("Backtrace extractor is running!\n\n");
281     printf("[*] Checking if ftrace file exists in default filesystem\n");
282 }
283
284
285 /*
286  * main: check Ftrace file exists in default filesystem (/root/.lkvm/default/
287  * virt/trace) and launch thread in [backtrace_ext__init] function.
288  *
289  */
290 int main(int argc, char *argv[])
291 {
292     const char *filename = "/root/.lkvm/default/virt/trace";
293     char ftrace_filename[FTRACE_NAME_LEN] =
294                                     { 't', 'r', 'a', 'c', 'e', ' ', '\0'  };
295     int ret, nthreads, i;
296     pthread_t threads[MAX_THREADS];
297
298     ret = pthread_mutex_init(&nfile_lock, NULL);
299     if (ret != 0) {
300         printf("Mutex initialization failed...\n");
301         return -1;
302     }
303
```

```
304     backtrace_ext__messages();
305     nthreads = 0;
306     while (true) {
307
308         printf("[!] Ftrace file doesn't exist...\n");
309         sleep(5);
310         if (access(filename, F_OK) != -1) {
311
312             printf("\n[*] Ftrace file has been found\n");
313             sleep(3);
314
315             ftrace_filename[5] = nfile + '0';
316
317             /* move/rename file and launch thread */
318             rename(filename, ftrace_filename);
319             ret = pthread_create(&threads[nthreads], NULL,
320                     backtrace_ext__init, (void *) ftrace_filename);
321             if (ret) {
322                 printf("[!] Thread failed...\n");
323             } else {
324                 printf("--> Thread No. %d with TID %d has been launched to "
325                         "parse ftrace file!\n\n", nfile, threads[nthreads]);
326             }
327
328             nthreads = nthreads + 1;
329
330             pthread_mutex_lock(&nfile_lock);
331             nfile = nfile + 1;
332             pthread_mutex_unlock(&nfile_lock);
333
334             printf("\n[*] Checking if ftrace file exists in default "
335                                                         "filesystem\n");
336         }
337     }
338
339     for (i = 0; i < nthreads; i++) {
340         pthread_join(threads[i], NULL);
341     }
342
343     pthread_mutex_destroy(&nfile_lock);
344
345     return 0;
346 }
```

A.7 PARSER_STACKTRACE.C

```
 1 /*
 2  * Project name: DADE - Data Anomaly Detection Engine
 3  * Project managers: José Luis Briz and Darío Suárez - Computer Architecture Group
 4  * University of Zaragoza
 5  * Author: Luis Fueris Martín
 6  * Date: 3 july 2018
 7  * File name: parser_stacktrace.c
 8  * Docs: https://www.gnu.org/software/libc/manual/html_node/Regular-Expressions.html
 9  *
10  */
11
12 #include <stdio.h>
13 #include <stdlib.h>
14 #include <stdbool.h>
15 #include <string.h>
16 #include <regex.h>
17 #include <linux/errno.h>
18
19 #include "linux/list.h"
20 #include "parser_stacktrace.h"
21 #include "backtrace_ext.h"
22
23 #define DEBUG           0
24
25 static int BEGIN_STACK;
26 static int COPY_OBJ_VALUE;
27 const char *delimiters      = " ";
28 static char *func_name, *obj_value, *first_obj_value;
29 static size_t func_name_len, obj_value_len;
30 static FILE *_bbdd_fd;
31 static _id;
32 regex_t regex_new_process, regex_ptr_name;
33 /* define and initializes a list_head named result_head.
34  * The majority of the linked list routines accept one or
35  * two parameters; the head node or the head node plus an
36  * actual list node. Ref: Linux Kernel Development
37  */
38 static LIST_HEAD(result_list);
39 static int list_length;
40
41
```

```
42 /*
43  * parser_stacktrace__erase_list: clean [result_list] when new stacktrace
44  * will be created.
45  *
46  * @result_list: the kernel list pointer
47  *
48  */
49 static void parser_stacktrace__erase_list(struct list_head *result_list)
50 {
51     struct backtrace *ptr;
52
53     list_for_each_entry(ptr, result_list, list) {
54         free(ptr->addr);
55         free(ptr->func_name);
56     }
57 }
58
59
60 /*
61  * parser_stacktraec__state_machin: execute state machine to retrieve
62  * stacktrace; function name and address. Allocate these fields in
63  * [result_list].
64  *
65  * @token: the word pointer
66  * @trace: the backtrace struct pointer
67  * @flag: the pointer which indicate what is [token] (equal, function name or
68  * address)
69  *
70  */
71 static int parser_stacktrace__state_machine(const char *token,
72                                             struct backtrace *trace,
73                                             int *flag)
74 {
75     size_t addr_len;
76
77     if (*flag == EQUAL) {
78         *flag = FUNCTION;
79     } else if (*flag == FUNCTION) {
80
81         func_name_len = strlen(token);
82         func_name = malloc(sizeof(char) * (func_name_len + 1));
83         if (func_name == NULL) {
84             return -ENOMEM;
85         }
86
87         func_name[func_name_len] = '\0';
88         strncpy(func_name, token, func_name_len);
89
90         *flag = ADDR;
91     } else if (*flag == ADDR) {
92         /* token is [module_name] not an addr */
93         if (token[0] == '[') {
94             return;
95         }
96
97         trace->func_name = malloc(sizeof(char) * (func_name_len + 1));
98         if (trace->func_name == NULL) {
99             return -ENOMEM;
100        }
101
102        trace->func_name[func_name_len] = '\0';
103        strncpy(trace->func_name, func_name, func_name_len);
104
105        addr_len = strlen(token);
106        trace->addr = malloc(sizeof(char) * (addr_len + 1));
107        if (trace->addr == NULL) {
108            return -ENOMEM;
109        }
110
111        strncpy(trace->addr, "0x", sizeof(char) * 3);
112        trace->addr[addr_len] = '\0';
113        strncat(trace->addr, &token[1], addr_len - 3);
114
115        list_add(&trace->list, &result_list);
116        list_length = list_length + 1;
117
118        *flag = EQUAL;
119    }
120
121    return 0;
122 }
123
124
125 /*
126  * parser_stacktrace__allocate_object: allocate kernel object in backtrace
127  * extractor module [backtrace_ext__save_stacktrace].
128  *
129  */
130 static int parser_stacktrace__allocate_kobject(void)
131 {
```

```
132     int ret;
133
134     if (first_obj_value != NULL) {
135         free(first_obj_value);
136     }
137
138     first_obj_value = malloc(sizeof(char) * (obj_value_len - 1));
139     if (first_obj_value == NULL) {
140             return -ENOMEM;
141     }
142
143     first_obj_value[obj_value_len - 2] = '\0';
144     strncpy(first_obj_value, obj_value, obj_value_len - 2);
145     ret = backtrace_ext__save_stacktrace(&result_list, list_length,
146                                          first_obj_value, obj_value_len,
147                                          _bbdd_fd, _id);
148     if (ret < 0) {
149         printf("Error when save stacktrace in backtrace extractor module\n");
150         return -1;
151     }
152
153
154     return 0;
155 }
156
157
158 /*
159  * parser_stacktrace__clen_reg: parse a specific token and allocate kernel
160  * object.
161  *
162  * @token: the word pointer
163  * @read: bytes read
164  *
165  */
166 static int parser_stacktrace__clean_reg(char *token, const ssize_t read)
167 {
168     int ret;
169     int *flag;
170     const char msg [MSG_LEN] = {'t', 'r', 'a', 'c', 'e', '>', '\n', '\0'};
171     struct backtrace *trace;
172
173     /* flag = 1 equal.
174               2 func. name.
175               3 func. addr. */
176     flag = malloc(sizeof(int *));
177     *flag = EQUAL;
178     while (token != NULL) {
179
180         /* end of file, allocate last kernel object */
181         if (read == -1) {
182             ret = parser_stacktrace__allocate_kobject();
183             if (ret < 0) {
184                 return -1;
185             }
186
187             break;
188         }
189
190         /* end of stacktrace, allocate kernel object */
191         ret = regexec(&regex_new_process, token, 0, NULL, 0);
192         if (ret == 0 && token[0] != '[') {
193             BEGIN_STACK = false;
194         }
195
196         /* allocate object name */
197         if (!BEGIN_STACK) {
198
199             ret = regexec(&regex_ptr_name, token, 0, NULL, 0);
200             if (ret == 0) {
201
202                 if (COPY_OBJ_VALUE) {
203
204                     ret = parser_stacktrace__allocate_kobject();
205                     if (ret < 0) {
206                         return -1;
207                     }
208
209                     COPY_OBJ_VALUE = false;
210                 }
211
212                 obj_value_len = strlen(token);
213                 obj_value = malloc(sizeof(char) * (obj_value_len - 1));
214                 if (obj_value == NULL) {
215                     return -ENOMEM;
216                 }
217
218                 obj_value[2] = '\0';
219                 strncpy(obj_value, "0x", sizeof(char) * 2);
220                 obj_value[obj_value_len - 2] = '\0';
221                 strncat(obj_value, &token[4], obj_value_len - 4);
```

```
222             }
223
224         } else {
225
226             /* execute state machine */
227             trace = (struct backtrace *) malloc(sizeof(struct backtrace));
228             if (trace == NULL) {
229                 return -ENOMEM;
230             }
231
232             ret = parser_stacktrace__state_machine(token, trace, flag);
233             if (ret < 0) {
234                 return -ENOMEM;
235             }
236         }
237
238         /* begin again new stacktrace */
239         if (strcmp(token, msg) == 0) {
240
241             parser_stacktrace__erase_list(&result_list);
242             list_del_init(&result_list);
243
244             list_length = 0;
245             COPY_OBJ_VALUE = true;
246             BEGIN_STACK = true;
247         }
248
249         token = strtok(NULL, delimiters);
250     }
251
252     if (flag != NULL) {
253         free(flag);
254     }
255
256     return 0;
257 }
258
259
260 /*
261  * parser_stacktrace__extract: read line by line and call [clean_reg] to par-
262  * se tokens.
263  *
264  * @fd: the ftrace filename pointer
265  *
266  */
267 static void parser_stacktrace__extract(FILE *fd)
268 {
269     char *line, *token;
270     size_t len;
271     ssize_t read;
272     int skip;
273
274     read = 0;
275     skip = SKIP_N_LINES;
276     while (read != -1) {
277
278         read = getline(&line, &len, fd);
279         /* skip ftrace file header */
280         if (skip > 0) {
281             skip = skip - 1;
282             continue;
283         }
284
285         token = strtok(line, delimiters);
286         parser_stacktrace__clean_reg(token, read);
287     }
288 }
289
290
291 /*
292  * parser_stacktrace__init: initialize function to parse ftrace file.
293  *
294  * @filename: the ftrace filename pointer
295  * @bbdd_fd: the descriptor pointer to write kernel objects
296  *
297  */
298 int parser_stacktrace__init(const char *filename, FILE *bbdd_fd, int id)
299 {
300     FILE *fd;
301     const char *str_new_process = "^[[:blank:]]*(.*sh|init-1|kthreadd|"
302     "<...>|kworker|kdevtmpfs|bash|<idle>|wc|cat|.*grep|dmesg|insmod|"
303     "iptables)";
304     const char *str_ptr_name    = "^[[:blank:]]*ptr";
305
306     _id = id;
307     _bbdd_fd = bbdd_fd;
308     fd = fopen(filename, "r");
309     if (fd == NULL) {
310         printf("File not open (check if exists or the path)\n");
311         return -EBADR;
```

```
312     }
313
314     regcomp(&regex_new_process, str_new_process, REG_EXTENDED);
315     regcomp(&regex_ptr_name, str_ptr_name, REG_EXTENDED);
316
317     BEGIN_STACK      = false;
318     COPY_OBJ_VALUE   = false;
319     parser_stacktrace__extract(fd);
320
321     regfree(&regex_new_process);
322     regfree(&regex_ptr_name);
323     fclose(fd);
324
325     return 0;
326 }
```

A.8 INTEGRITY_VERIFIER.AWK

```
1  #! /usr/bin/awk -f
2  # Project name: DADE - Data Anomaly Detection Engine
3  # Project managers: José Luis Briz and Darío Suárez
4  # Computer Architecture Group, Univesity of Zaragoza
5  # Author: Luis Fueris Martín
6  # Date: 27 august 2018
7  # File name: integrity_verifier.awk
8  # Docs: https://wwww.gnu.org/software/gawk/manual/gawk.html
9
10 BEGIN {
11
12     BARGS = 85
13
14     if (ARGC < 3) {
15         print ENVIRON["_"], "kobjecto.data kobject1.data";
16         exit BARGS;
17     }
18
19     f_file = ARGV[1];
20
21 }
22
23 $1 ~ /obj_name/ {
24
25     split($2, obj_names, "+");
26     key = obj_names[1];
27
28 }
29
30 $1 ~ /hash/ {
31
32     hash = $2;
33     FILENAME == f_file ? vo_kobjects[key] = hash : v1_kobjects[key] = hash;
34
35 }
36
37 END {
38
39     for (vo_key in vo_kobjects) {
40         for (v1_key in v1_kobjects) {
41             if (vo_key == v1_key &&
42                 vo_kobjects[vo_key] != v1_kobjects[v1_key]) {
43
44                 print "\n[!] Rootkit has been detected\n";
45                 print "\tKernel object:", vo_key, "\n";
46                 print "Hashes no. 1: ", vo_kobjects[vo_key];
47                 print "Hashes no. 2: ", v1_kobjects[v1_key];
48             }
49         }
50     }
51 }
52
```

A.9 FICHERO CONFIGURACIÓN KERNEL - KVMTOOL

```
CONFIG_SERIAL_8250=y
CONFIG_SERIAL_8250_CONSOLE=y
CONFIG_IA32_EMULATION=y
CONFIG_VIRTIO=y
CONFIG_VIRTIO_RING=y
CONFIG_VIRTIO_PCI=y
CONFIG_VIRTIO_BLK=y
CONFIG_VIRTIO_NET=y
CONFIG_NET_9P=y
CONFIG_NET_9P_VIRTIO=y
CONFIG_9P_FS=y
```

```
CONFIG_VIRTIO_CONSOLE=y
CONFIG_HW_RANDOM_VIRTIO=y
```

A.10 TABLA DE LITERALES

```
1
2 int foo()
3 {
4     __asm__("LDR w0, =3462381247");
5     return 0;
6 }
7
8 int main()
9 {
10    foo();
11    return 0;
12 }
```

```
2 literal_pool:     file format elf64-littleaarch64

264     ...
265
266 0000000000400520 <foo>:
267   400520:   18000120    ldr w0, 400544 <main+0x18>
268   400524:   52800000    mov w0, #0x0                    // #0
269   400528:   d65f03c0    ret
270
271 000000000040052c <main>:
272   40052c:   a9bf7bfd    stp x29, x30, [sp,#-16]!
273   400530:   910003fd    mov x29, sp
274   400534:   97fffffb    bl  400520 <foo>
275   400538:   52800000    mov w0, #0x0                    // #0
276   40053c:   a8c17bfd    ldp x29, x30, [sp],#16
277   400540:   d65f03c0    ret
278   400544:   ce5fbebf    .word   0xce5fbebf
```

Cuadro A.1: Código ensamblador literal_pool.asm

Conviene notar que, la tabla de literales está a partir de la instrucción nº 0x400544 (`main+0x18`). Sirve sólo para almacenar datos codificados en una secuencia de instrucciones, es decir, nunca se ejecuta esa isntrucción. La constante `0xce5fbebf` (en decimal `3462381247`) está cargada con la instrucción nº 400520.

A.11 GOOD_MUTENROSHI.C

```
1
2 #include <linux/module.h>
3 #include <linux/init.h>
4 #include <linux/proc_fs.h>
5 #include <linux/seq_file.h>
6
7 #define KMEM        512
8 #define PROC_NAME   "mutenroshi"
9 #define MSG         "kamehameha!"
10
11 static int proc_show(struct seq_file *m, void *v)
12 {
13     seq_printf(m, "%s\n", MSG);
14     return 0;
15 }
16
17 static int proc_open(struct inode *inode, struct file *file)
18 {
19     return single_open(file, proc_show, NULL);
20 }
21
22 static struct file_operations fops = {
23     .owner = THIS_MODULE,
24     .open = proc_open,
25     .release = single_release,
26     .read = seq_read,
27     .llseek = seq_lseek,
```

```
28 };
29
30 static int __init mutenroshi_init(void)
31 {
32     struct proc_dir_entry *proc_file;
33
34     proc_file = proc_create(PROC_NAME, 0644, NULL, &fops);
35     if (proc_file == NULL) {
36         return -ENOMEM;
37     } else {
38         printk(KERN_INFO "Hi, I'm %s rootkit.\n", PROC_NAME);
39     }
40
41     return 0;
42 }
43
44 static void __exit mutenroshi_exit(void)
45 {
46     remove_proc_entry(PROC_NAME, NULL);
47     printk(KERN_INFO "Bye...\n");
48 }
49
50 module_init(mutenroshi_init);
51 module_exit(mutenroshi_exit);
52 MODULE_LICENSE("GPL");
```

A.12 BAD_MUTENROSHI.C

```
1
2 #include <linux/module.h>
3 #include <linux/init.h>
4 #include <linux/proc_fs.h>
5 #include <linux/seq_file.h>
6
7 #define KMEM        512
8 #define PROC_NAME   "mutenroshi"
9 #define MSG         "kamehameha!"
10
11 static int proc_show(struct seq_file *m, void *v)
12 {
13     seq_printf(m, "%s\n", MSG);
14     return 0;
15 }
16
17 static int proc_open(struct inode *inode, struct file *file)
18 {
19     __asm__("LDR w0, =3462381247");
20     return single_open(file, proc_show, NULL);
21 }
22
23 static struct file_operations fops={
24     .owner = THIS_MODULE,
25     .open = proc_open,
26     .release = single_release,
27     .read = seq_read,
28     .llseek = seq_lseek,
29 };
30
31 static int __init mutenroshi_init(void)
32 {
33     struct proc_dir_entry *proc_file;
34
35     proc_file = proc_create(PROC_NAME, 0644, NULL, &fops);
36     if (proc_file == NULL) {
37         return -ENOMEM;
38     } else {
39         printk(KERN_INFO "Hi, I'm %s rootkit.\n", PROC_NAME);
40     }
41
42     return 0;
43 }
44
45 static void __exit mutenroshi_exit(void)
46 {
47     remove_proc_entry(PROC_NAME, NULL);
48     printk(KERN_INFO "Bye...\n");
49 }
50
51 module_init(mutenroshi_init);
52 module_exit(mutenroshi_exit);
53 MODULE_LICENSE("GPL");
```

A.13 STACKTRACE GOOD_MUTENROSHI.KO

```
10145 Kernel object No. 510:
10146
10147  obj_name: single_open+0x78/0xc0
10148  obj_value: 0xffff80001c2fbf00
10149  hash: 90ffd3626bbdc6a9b964909f2ba0fb8c
10150
10151                      stacktrace
10152                     (kernel trap)
10153
10154  [0] elo_svc_naked+0x34/0x38 - 0xffff000008083880
10155  [1] SyS_openat+0x3c/0x4c - 0xffff000008287bd4
10156  [2] do_sys_open+0x14c/0x214 - 0xffff000008287a88
10157  [3] do_filp_open+0x6c/0xe8 - 0xffff00000829c750
10158  [4] path_openat+0x518/0xf78 - 0xffff00000829abb8
10159  [5] vfs_open+0x5c/0x8c - 0xffff000008287261c
10160  [6] do_dentry_open+0x22c/0x330 - 0xffff000008286004
10161  [7] proc_reg_open+0x9c/0x13c - 0xffff0000082ff228
10162  [8] proc_open+0x28/0x40 - 0xffff000000b10028
10163  [9] single_open+0x78/0xc0 - 0xffff00000082b5078
```

A.14 STACKTRACE BAD_MUTENROSHI.KO

```
4986 Kernel object No. 275:
4987
4988  obj_name: single_open+0x78/0xc0
4989  obj_value: 0xffff80001c276b80
4990  hash: 7fb1c3bd6212987c5744372998294 0b4
4991
4992                      stacktrace
4993                     (kernel trap)
4994
4995  [0] elo_svc_naked+0x34/0x38 - 0xffff000008083880
4996  [1] SyS_openat+0x3c/0x4c - 0xffff000008287bd4
4997  [2] do_sys_open+0x14c/0x214 - 0xffff000008287a88
4998  [3] do_filp_open+0x6c/0xe8 - 0xffff00000829c750
4999  [4] path_openat+0x518/0xf78 - 0xffff00000829abb8
5000  [5] vfs_open+0x5c/0x8c - 0xffff000008287261c
5001  [6] do_dentry_open+0x22c/0x330 - 0xffff000008286004
5002  [7] proc_reg_open+0x9c/0x13c - 0xffff0000082ff228
5003  [8] proc_open+0x2c/0x40 - 0xffff000000b1802c
5004  [9] single_open+0x78/0xc0 - 0xffff00000082b5078
```

A.15 FORMATOS KERNEL LINUX

El principal formato [2] ejecutable en Linux es ELF, es el más utilizado en sistemas operativos tipo Unix. También soporta el formato `a.out` (*Assembler OUTput Format*) utilizado en versiones anteriores de Linux. Sin embargo, también es posible ejecutar binarios de otros sistemas operativos; MS-DOS, familia UNIX BSD. Conviene notar que, los formatos ejecutables de Java y *bash* son independientes de la plataforma.

La estructura `linux_binfmt` describe el formato y posee tres procedimientos.

- **`linux_binfmt`**. prepara un nuevo entorno de ejecución para el proceso leyendo la información almacenada en el fichero a ejecutar.
- **`load_shlib`**. Enlaza dinámicamente una biblioteca al proceso en ejecición.
- **`core_dump`**. Almacena en un fichero llamado `core` el contexto de ejecución del proceso. El fichero es creado cuando un proceso recibe una señal cuya principal acción es realizar un *dump*

Todos los objetos que poseen la estructura `linux_binfmt` son almacenados en una lista enlazada. La dirección del primer elemento está almacenada en la variable `formats`. Los objetos pueden ser insertados y borrados mediante las funciones `register_binfmt` y `unregister_binfmt`, respectivamente. Cuando el sistema arranca las funciones de inserción en la lista son ejecutadas. Esto ocurre cuando los formatos son compilados dentro del kernel. Cuando un formato es incluido[4] en el kernel como módulo, también es ejecutada la función de inserción.

A.16 GOOD_BINFMT_SCRIPT.C

```
/*
 *  linux/fs/binfmt_script.c
 *
 *  Copyright (C) 1996  Martin von Lowis
 *  original #!-checking implemented by tytso.
 */

#include <linux/module.h>
#include <linux/string.h>
#include <linux/stat.h>
#include <linux/binfmts.h>
#include <linux/init.h>
#include <linux/file.h>
#include <linux/err.h>
#include <linux/fs.h>

static int load_script(struct linux_binprm *bprm)
{
    const char *i_arg, *i_name;
    char *cp;
    struct file *file;
    int retval;

    if ((bprm->buf[0] != '#') || (bprm->buf[1] != '!'))
        return -ENOEXEC;

    /*
     * If the script filename will be inaccessible after exec, typically
     * because it is a "/dev/fd/<fd>/.." path against an O_CLOEXEC fd, give
     * up now (on the assumption that the interpreter will want to load
     * this file).
     */
    if (bprm->interp_flags & BINPRM_FLAGS_PATH_INACCESSIBLE)
        return -ENOENT;

    /*
     * This section does the #! interpretation.
     * Sorta complicated, but hopefully it will work.  -TYT
     */

    allow_write_access(bprm->file);
    fput(bprm->file);
    bprm->file = NULL;

    bprm->buf[BINPRM_BUF_SIZE - 1] = '\0';
    if ((cp = strchr(bprm->buf, '\n')) == NULL)
        cp = bprm->buf+BINPRM_BUF_SIZE-1;
    *cp = '\0';
    while (cp > bprm->buf) {
        cp--;
        if ((*cp == ' ') || (*cp == '\t'))
            *cp = '\0';
        else
            break;
    }
    for (cp = bprm->buf+2; (*cp == ' ') || (*cp == '\t'); cp++);
    if (*cp == '\0')
        return -ENOEXEC; /* No interpreter name found */
    i_name = cp;
    i_arg = NULL;
    for ( ; *cp && (*cp != ' ') && (*cp != '\t'); cp++)
        /* nothing */ ;
```

4 $>insmod binfmt_<format>.ko

```
63      while ((*cp == ' ') || (*cp == '\t'))
64          *cp++ = '\0';
65      if (*cp)
66          i_arg = cp;
67      /*
68       * OK, we've parsed out the interpreter name and
69       * (optional) argument.
70       * Splice in (1) the interpreter's name for argv[0]
71       *           (2) (optional) argument to interpreter
72       *           (3) filename of shell script (replace argv[0])
73       *
74       * This is done in reverse order, because of how the
75       * user environment and arguments are stored.
76       */
77      retval = remove_arg_zero(bprm);
78      if (retval)
79          return retval;
80      retval = copy_strings_kernel(1, &bprm->interp, bprm);
81      if (retval < 0)
82          return retval;
83      bprm->argc++;
84      if (i_arg) {
85          retval = copy_strings_kernel(1, &i_arg, bprm);
86          if (retval < 0)
87              return retval;
88          bprm->argc++;
89      }
90      retval = copy_strings_kernel(1, &i_name, bprm);
91      if (retval)
92          return retval;
93      bprm->argc++;
94      retval = bprm_change_interp(i_name, bprm);
95      if (retval < 0)
96          return retval;
97
98      /*
99       * OK, now restart the process with the interpreter's dentry.
100      */
101     file = open_exec(i_name);
102     if (IS_ERR(file))
103         return PTR_ERR(file);
104
105     bprm->file = file;
106     retval = prepare_binprm(bprm);
107     if (retval < 0)
108         return retval;
109     return search_binary_handler(bprm);
110 }
111
112 static struct linux_binfmt script_format = {
113     .module      = THIS_MODULE,
114     .load_binary     = load_script,
115 };
116
117 static int __init init_script_binfmt(void)
118 {
119     register_binfmt(&script_format);
120     return 0;
121 }
122
123 static void __exit exit_script_binfmt(void)
124 {
125     unregister_binfmt(&script_format);
126 }
127
128 core_initcall(init_script_binfmt);
129 module_exit(exit_script_binfmt);
130 MODULE_LICENSE("GPL");
```

A.17 BAD_BINFMT_SCRIPT.C

```
1  /*
2   *  linux/fs/binfmt_script.c
3   *
4   *  Copyright (C) 1996  Martin von Lowis
5   *  original #!-checking implemented by tytso.
6   */
7
8  #include <linux/module.h>
9  #include <linux/string.h>
10 #include <linux/stat.h>
11 #include <linux/binfmts.h>
12 #include <linux/init.h>
13 #include <linux/file.h>
14 #include <linux/err.h>
15 #include <linux/fs.h>
16
```

```
17 static int load_script(struct linux_binprm *bprm)
18 {
19     const char *i_arg, *i_name;
20     char *cp;
21     struct file *file;
22     int retval;
23
24     if ((bprm->buf[0] != '#') || (bprm->buf[1] != '!'))
25         return -ENOEXEC;
26
27     /*
28      * If the script filename will be inaccessible after exec, typically
29      * because it is a "/dev/fd/<fd>/.." path against an O_CLOEXEC fd, give
30      * up now (on the assumption that the interpreter will want to load
31      * this file).
32      */
33     if (bprm->interp_flags & BINPRM_FLAGS_PATH_INACCESSIBLE)
34         return -ENOENT;
35
36     /*
37      * This section does the #! interpretation.
38      * Sorta complicated, but hopefully it will work.  -TYT
39      */
40
41     allow_write_access(bprm->file);
42     fput(bprm->file);
43     bprm->file = NULL;
44
45     bprm->buf[BINPRM_BUF_SIZE - 1] = '\0';
46     if ((cp = strchr(bprm->buf, '\n')) == NULL)
47         cp = bprm->buf+BINPRM_BUF_SIZE-1;
48     *cp = '\0';
49     while (cp > bprm->buf) {
50         cp--;
51         if ((*cp == ' ') || (*cp == '\t'))
52             *cp = '\0';
53         else
54             break;
55     }
56     for (cp = bprm->buf+2; (*cp == ' ') || (*cp == '\t'); cp++);
57     if (*cp == '\0')
58         return -ENOEXEC; /* No interpreter name found */
59     i_name = cp;
60     i_arg = NULL;
61     for ( ; *cp && (*cp != ' ') && (*cp != '\t'); cp++)
62         /* nothing */ ;
63     while ((*cp == ' ') || (*cp == '\t'))
64         *cp++ = '\0';
65     if (*cp)
66         i_arg = cp;
67     /*
68      * OK, we've parsed out the interpreter name and
69      * (optional) argument.
70      * Splice in (1) the interpreter's name for argv[0]
71      *           (2) (optional) argument to interpreter
72      *           (3) filename of shell script (replace argv[0])
73      *
74      * This is done in reverse order, because of how the
75      * user environment and arguments are stored.
76      */
77     retval = remove_arg_zero(bprm);
78     if (retval)
79         return retval;
80     retval = copy_strings_kernel(1, &bprm->interp, bprm);
81     if (retval < 0)
82         return retval;
83     bprm->argc++;
84     if (i_arg) {
85         retval = copy_strings_kernel(1, &i_arg, bprm);
86         if (retval < 0)
87             return retval;
88         bprm->argc++;
89     }
90     retval = copy_strings_kernel(1, &i_name, bprm);
91     if (retval)
92         return retval;
93     bprm->argc++;
94     retval = bprm_change_interp(i_name, bprm);
95     if (retval < 0)
96         return retval;
97
98     get_max_files();
99     /*
100     * OK, now restart the process with the interpreter's dentry.
101     */
102    file = open_exec(i_name);
103    if (IS_ERR(file))
104        return PTR_ERR(file);
105
106    bprm->file = file;
```

```
107     retval = prepare_binprm(bprm);
108     if (retval < 0)
109         return retval;
110     return search_binary_handler(bprm);
111 }
112
113 static struct linux_binfmt script_format = {
114     .module      = THIS_MODULE,
115     .load_binary     = load_script,
116 };
117
118 static int __init init_script_binfmt(void)
119 {
120     register_binfmt(&script_format);
121     return 0;
122 }
123
124 static void __exit exit_script_binfmt(void)
125 {
126     unregister_binfmt(&script_format);
127 }
128
129 core_initcall(init_script_binfmt);
130 module_exit(exit_script_binfmt);
131 MODULE_LICENSE("GPL");
```

A.18 STACKTRACE GOOD_BINFMT_SCRIPT.KO

```
185 Kernel object No. 11:
186
187  obj_name: search_binary_handler+0xb0/0x204
188  obj_value: 0xffff80001c300400
189  hash: 8ebd6a46aa779b90cd651b708a475226
190
191                      stacktrace
192                     (kernel trap)
193
194  [0] elo_svc_naked+0x34/0x38 - 0xffff000008083880
195  [1] SyS_execve+0x38/0x4c - 0xffff000008293820
196  [2] do_execve+0x44/0x54 - 0xffff000008293 5d4
197  [3] do_execveat_common.isra.37+0x494/0x63c - 0xffff000008293 3e8
198  [4] search_binary_handler+0xb0/0x204 - 0xffff000008292a44
199  [5] load_script+0x1fc/0x218 - 0xffff00000b101fc
200  [6] search_binary_handler+0xb0/0x204 - 0xffff000008292a44
```

A.19 STACKTRACE BAD_BINFMT_SCRIPT.KO

```
2195 Kernel object No. 131:
2196
2197  obj_name: search_binary_handler+0xb0/0x204
2198  obj_value: 0xffff80001c2fd300
2199  hash: 17e39e204a228aff9da4c2d657b34a68
2200
2201                      stacktrace
2202                     (kernel trap)
2203
2204  [0] elo_svc_naked+0x34/0x38 - 0xffff000008083880
2205  [1] SyS_execve+0x38/0x4c - 0xffff000008293820
2206  [2] do_execve+0x44/0x54 - 0xffff000008293 5d4
2207  [3] do_execveat_common.isra.37+0x494/0x63c - 0xffff000008293 3e8
2208  [4] search_binary_handler+0xb0/0x204 - 0xffff000008292a44
2209  [5] load_script+0x1f8/0x218 - 0xffff00000b101f8
2210  [6] search_binary_handler+0xb0/0x204 - 0xffff000008292a44
```

A.20 GOOD_BINFMT_SCRIPT.ASM

```
1
2 good_binfmt_script.ko:     file format elf64-littleaarch64
3
4
5 Disassembly of section .text:
6
7 0000000000000000 <load_script>:
8    0:   a9bd7bfd    stp x29, x30, [sp,#-48]!
9    4:   910003fd    mov x29, sp
10   8:   f9000bf3    str x19, [sp,#16]
11   c:   aa0003f3    mov x19, x0
12  10:   52842460    mov w0, #0x2123                 // #8483
13  14:   79400261    ldrh    w1, [x19]
14  18:   6b00003f    cmp w1, w0
15  1c:   54000f01    b.ne    1fc <load_script+0x1fc>
```

```
 16    20:   b940da60    ldr w0, [x19,#216]
 17    24:   37100f40    tbnz    w0, #2, 20c <load_script+0x20c>
 18    28:   f9405660    ldr x0, [x19,#168]
 19    2c:   b4000120    cbz x0, 50 <load_script+0x50>
 20    30:   f9401000    ldr x0, [x0,#32]
 21    34:   91058000    add x0, x0, #0x160
 22    38:   f9800011    prfm    pstl1strm, [x0]
 23    3c:   885f7c01    ldxr    w1, [x0]
 24    40:   11000421    add w1, w1, #0x1
 25    44:   88027c01    stxr    w2, w1, [x0]
 26    48:   35ffffa2    cbnz    w2, 3c <load_script+0x3c>
 27    4c:   f9405660    ldr x0, [x19,#168]
 28    50:   94000000    bl  0 <fput>
 29    54:   f900567f    str xzr, [x19,#168]
 30    58:   52800141    mov w1, #0xa                    // #10
 31    5c:   3901fe7f    strb    wzr, [x19,#127]
 32    60:   aa1303e0    mov x0, x19
 33    64:   94000000    bl  0 <strchr>
 34    68:   eb1f001f    cmp x0, xzr
 35    6c:   9101fe61    add x1, x19, #0x7f
 36    70:   9a811000    csel    x0, x0, x1, ne
 37    74:   eb13001f    cmp x0, x19
 38    78:   3900001f    strb    wzr, [x0]
 39    7c:   540001e9    b.ls    b8 <load_script+0xb8>
 40    80:   385ff002    ldurb   w2, [x0,#-1]
 41    84:   d1000401    sub x1, x0, #0x1
 42    88:   7100245f    cmp w2, #0x9
 43    8c:   54000060    b.eq    98 <load_script+0x98>
 44    90:   7100805f    cmp w2, #0x20
 45    94:   54000121    b.ne    b8 <load_script+0xb8>
 46    98:   3900003f    strb    wzr, [x1]
 47    9c:   eb13003f    cmp x1, x19
 48    a0:   540000c0    b.eq    b8 <load_script+0xb8>
 49    a4:   385ffc20    ldrb    w0, [x1,#-1]!
 50    a8:   7100241f    cmp w0, #0x9
 51    ac:   54ffff60    b.eq    98 <load_script+0x98>
 52    b0:   7100801f    cmp w0, #0x20
 53    b4:   54ffff20    b.eq    98 <load_script+0x98>
 54    b8:   39400a62    ldrb    w2, [x19,#2]
 55    bc:   91000a61    add x1, x19, #0x2
 56    c0:   7100245f    cmp w2, #0x9
 57    c4:   54000081    b.ne    d4 <load_script+0xd4>
 58    c8:   38401c22    ldrb    w2, [x1,#1]!
 59    cc:   7100245f    cmp w2, #0x9
 60    d0:   54ffffc0    b.eq    c8 <load_script+0xc8>
 61    d4:   7100805f    cmp w2, #0x20
 62    d8:   54ffff80    b.eq    c8 <load_script+0xc8>
 63    dc:   34000902    cbz w2, 1fc <load_script+0x1fc>
 64    e0:   39400022    ldrb    w2, [x1]
 65    e4:   f90017a1    str x1, [x29,#40]
 66    e8:   f90013bf    str xzr, [x29,#32]
 67    ec:   121a7840    and w0, w2, #0xfffffffd
 68    f0:   34000160    cbz w0, 11c <load_script+0x11c>
 69    f4:   7100245f    cmp w2, #0x9
 70    f8:   54000120    b.eq    11c <load_script+0x11c>
 71    fc:   38401c22    ldrb    w2, [x1,#1]!
 72   100:   121a7840    and w0, w2, #0xfffffffd
 73   104:   7100245f    cmp w2, #0x9
 74   108:   35ffff80    cbnz    w0, f8 <load_script+0xf8>
 75   10c:   7100245f    cmp w2, #0x9
 76   110:   540000a1    b.ne    124 <load_script+0x124>
 77   114:   3800143f    strb    wzr, [x1],#1
 78   118:   39400022    ldrb    w2, [x1]
 79   11c:   7100245f    cmp w2, #0x9
 80   120:   54ffffa0    b.eq    114 <load_script+0x114>
 81   124:   7100805f    cmp w2, #0x20
 82   128:   54ffff60    b.eq    114 <load_script+0x114>
 83   12c:   350000e2    cbnz    w2, 148 <load_script+0x148>
 84   130:   aa1303e0    mov x0, x19
 85   134:   94000000    bl  0 <remove_arg_zero>
 86   138:   34000100    cbz w0, 158 <load_script+0x158>
 87   13c:   f9400bf3    ldr x19, [sp,#16]
 88   140:   a8c37bfd    ldp x29, x30, [sp],#48
 89   144:   d65f03c0    ret
 90   148:   aa1303e0    mov x0, x19
 91   14c:   f90013a1    str x1, [x29,#32]
 92   150:   94000000    bl  0 <remove_arg_zero>
 93   154:   35ffff40    cbnz    w0, 13c <load_script+0x13c>
 94   158:   91034261    add x1, x19, #0xd0
 95   15c:   aa1303e2    mov x2, x19
 96   160:   52800020    mov w0, #0x1                    // #1
 97   164:   94000000    bl  0 <copy_strings_kernel>
 98   168:   37fffea0    tbnz    w0, #31, 13c <load_script+0x13c>
 99   16c:   b940c260    ldr w0, [x19,#192]
100   170:   f94013a1    ldr x1, [x29,#32]
101   174:   11000400    add w0, w0, #0x1
102   178:   b900c260    str w0, [x19,#192]
103   17c:   b4000121    cbz x1, 1a0 <load_script+0x1a0>
104   180:   910083a1    add x1, x29, #0x20
105   184:   aa1303e2    mov x2, x19
```

```
106  188:   52800020    mov w0, #0x1                    // #1
107  18c:   94000000    bl  0 <copy_strings_kernel>
108  190:   37fffd60    tbnz    w0, #31, 13c <load_script+0x13c>
109  194:   b940c260    ldr w0, [x19,#192]
110  198:   11000400    add w0, w0, #0x1
111  19c:   b900c260    str w0, [x19,#192]
112  1a0:   aa1303e2    mov x2, x19
113  1a4:   9100a3a1    add x1, x29, #0x28
114  1a8:   52800020    mov w0, #0x1                    // #1
115  1ac:   94000000    bl  0 <copy_strings_kernel>
116  1b0:   35fffc60    cbnz    w0, 13c <load_script+0x13c>
117  1b4:   b940c262    ldr w2, [x19,#192]
118  1b8:   aa1303e1    mov x1, x19
119  1bc:   f94017a0    ldr x0, [x29,#40]
120  1c0:   11000442    add w2, w2, #0x1
121  1c4:   b900c262    str w2, [x19,#192]
122  1c8:   94000000    bl  0 <bprm_change_interp>
123  1cc:   37fffb80    tbnz    w0, #31, 13c <load_script+0x13c>
124  1d0:   f94017a0    ldr x0, [x29,#40]
125  1d4:   94000000    bl  0 <open_exec>
126  1d8:   b140041f    cmn x0, #0x1, lsl #12
127  1dc:   54fffb08    b.hi    13c <load_script+0x13c>
128  1e0:   f9005660    str x0, [x19,#168]
129  1e4:   aa1303e0    mov x0, x19
130  1e8:   94000000    bl  0 <prepare_binprm>
131  1ec:   37fffa80    tbnz    w0, #31, 13c <load_script+0x13c>
132  1f0:   aa1303e0    mov x0, x19
133  1f4:   94000000    bl  0 <search_binary_handler>
134  1f8:   17ffffd1    b   13c <load_script+0x13c>
135  1fc:   128000e0    mov w0, #0xffffffff8            // #-8
136  200:   f9400bf3    ldr x19, [sp,#16]
137  204:   a8c37bfd    ldp x29, x30, [sp],#48
138  208:   d65f03c0    ret
139  20c:   12800020    mov w0, #0xfffffffe             // #-2
140  210:   17ffffcb    b   13c <load_script+0x13c>
```

A.21 BAD_BINFMT_SCRIPT.ASM

```
1
2  bad_binfmt_script.ko:     file format elf64-littleaarch64
3
4
5  Disassembly of section .text:
6
7  0000000000000000 <load_script>:
8     0:   a9bd7bfd    stp x29, x30, [sp,#-48]!
9     4:   910003fd    mov x29, sp
10    8:   f9000bf3    str x19, [sp,#16]
11    c:   aa0003f3    mov x19, x0
12   10:   52842460    mov w0, #0x2123                 // #8483
13   14:   79400261    ldrh    w1, [x19]
14   18:   6b00003f    cmp w1, w0
15   1c:   54000f21    b.ne    200 <load_script+0x200>
16   20:   b940da60    ldr w0, [x19,#216]
17   24:   37100f60    tbnz    w0, #2, 210 <load_script+0x210>
18   28:   f9405660    ldr x0, [x19,#168]
19   2c:   b4000120    cbz x0, 50 <load_script+0x50>
20   30:   f9401000    ldr x0, [x0,#32]
21   34:   91058000    add x0, x0, #0x160
22   38:   f9800011    prfm    pstl1strm, [x0]
23   3c:   885f7c01    ldxr    w1, [x0]
24   40:   11000421    add w1, w1, #0x1
25   44:   88027c01    stxr    w2, w1, [x0]
26   48:   35ffffa2    cbnz    w2, 3c <load_script+0x3c>
27   4c:   f9405660    ldr x0, [x19,#168]
28   50:   94000000    bl  0 <fput>
29   54:   f900567f    str xzr, [x19,#168]
30   58:   52800141    mov w1, #0xa                    // #10
31   5c:   3901fe7f    strb    wzr, [x19,#127]
32   60:   aa1303e0    mov x0, x19
33   64:   94000000    bl  0 <strchr>
34   68:   eb1f001f    cmp x0, xzr
35   6c:   9101fe61    add x1, x19, #0x7f
36   70:   9a811000    csel    x0, x0, x1, ne
37   74:   eb13001f    cmp x0, x19
38   78:   3900001f    strb    wzr, [x0]
39   7c:   540001e9    b.ls    b8 <load_script+0xb8>
40   80:   385ff002    ldurb   w2, [x0,#-1]
41   84:   d1000401    sub x1, x0, #0x1
42   88:   7100245f    cmp w2, #0x9
43   8c:   54000060    b.eq    98 <load_script+0x98>
44   90:   7100805f    cmp w2, #0x20
45   94:   54000121    b.ne    b8 <load_script+0xb8>
46   98:   3900003f    strb    wzr, [x1]
47   9c:   eb13003f    cmp x1, x19
48   a0:   540000c0    b.eq    b8 <load_script+0xb8>
49   a4:   385ffc20    ldrb    w0, [x1,#-1]!
```

```
50   a8:   7100241f    cmp w0, #0x9
51   ac:   54ffff60    b.eq    98 <load_script+0x98>
52   b0:   7100801f    cmp w0, #0x20
53   b4:   54ffff20    b.eq    98 <load_script+0x98>
54   b8:   39400a62    ldrb    w2, [x19,#2]
55   bc:   91000a61    add x1, x19, #0x2
56   c0:   7100245f    cmp w2, #0x9
57   c4:   54000081    b.ne    d4 <load_script+0xd4>
58   c8:   38401c22    ldrb    w2, [x1,#1]!
59   cc:   7100245f    cmp w2, #0x9
60   d0:   54ffffc0    b.eq    c8 <load_script+0xc8>
61   d4:   7100805f    cmp w2, #0x20
62   d8:   54ffff80    b.eq    c8 <load_script+0xc8>
63   dc:   34000922    cbz w2, 200 <load_script+0x200>
64   e0:   39400022    ldrb    w2, [x1]
65   e4:   f90017a1    str x1, [x29,#40]
66   e8:   f90013bf    str xzr, [x29,#32]
67   ec:   121a7840    and w0, w2, #0xfffffffd
68   f0:   34000160    cbz w0, 11c <load_script+0x11c>
69   f4:   7100245f    cmp w2, #0x9
70   f8:   54000120    b.eq    11c <load_script+0x11c>
71   fc:   38401c22    ldrb    w2, [x1,#1]!
72  100:   121a7840    and w0, w2, #0xfffffffd
73  104:   7100245f    cmp w2, #0x9
74  108:   35ffff80    cbnz    w0, f8 <load_script+0xf8>
75  10c:   7100245f    cmp w2, #0x9
76  110:   540000a1    b.ne    124 <load_script+0x124>
77  114:   3800143f    strb    wzr, [x1],#1
78  118:   39400022    ldrb    w2, [x1]
79  11c:   7100245f    cmp w2, #0x9
80  120:   54ffffa0    b.eq    114 <load_script+0x114>
81  124:   7100805f    cmp w2, #0x20
82  128:   54ffff60    b.eq    114 <load_script+0x114>
83  12c:   350000e2    cbnz    w2, 148 <load_script+0x148>
84  130:   aa1303e0    mov x0, x19
85  134:   94000000    bl  0 <remove_arg_zero>
86  138:   34000100    cbz w0, 158 <load_script+0x158>
87  13c:   f9400bf3    ldr x19, [sp,#16]
88  140:   a8c37bfd    ldp x29, x30, [sp],#48
89  144:   d65f03c0    ret
90  148:   aa1303e0    mov x0, x19
91  14c:   f90013a1    str x1, [x29,#32]
92  150:   94000000    bl  0 <remove_arg_zero>
93  154:   35ffff40    cbnz    w0, 13c <load_script+0x13c>
94  158:   91034261    add x1, x19, #0xd0
95  15c:   aa1303e2    mov x2, x19
96  160:   52800020    mov w0, #0x1                    // #1
97  164:   94000000    bl  0 <copy_strings_kernel>
98  168:   37fffea0    tbnz    w0, #31, 13c <load_script+0x13c>
99  16c:   b940c260    ldr w0, [x19,#192]
100 170:   f94013a1    ldr x1, [x29,#32]
101 174:   11000400    add w0, w0, #0x1
102 178:   b900c260    str w0, [x19,#192]
103 17c:   b4000121    cbz x1, 1a0 <load_script+0x1a0>
104 180:   910083a1    add x1, x29, #0x20
105 184:   aa1303e2    mov x2, x19
106 188:   52800020    mov w0, #0x1                    // #1
107 18c:   94000000    bl  0 <copy_strings_kernel>
108 190:   37fffd60    tbnz    w0, #31, 13c <load_script+0x13c>
109 194:   b940c260    ldr w0, [x19,#192]
110 198:   11000400    add w0, w0, #0x1
111 19c:   b900c260    str w0, [x19,#192]
112 1a0:   aa1303e2    mov x2, x19
113 1a4:   9100a3a1    add x1, x29, #0x28
114 1a8:   52800020    mov w0, #0x1                    // #1
115 1ac:   94000000    bl  0 <copy_strings_kernel>
116 1b0:   35fffc60    cbnz    w0, 13c <load_script+0x13c>
117 1b4:   b940c262    ldr w2, [x19,#192]
118 1b8:   aa1303e1    mov x1, x19
119 1bc:   f94017a0    ldr x0, [x29,#40]
120 1c0:   11000442    add w2, w2, #0x1
121 1c4:   b900c262    str w2, [x19,#192]
122 1c8:   94000000    bl  0 <bprm_change_interp>
123 1cc:   37fffb80    tbnz    w0, #31, 13c <load_script+0x13c>
124 1d0:   94000000    bl  0 <get_max_files>
125 1d4:   f94017a0    ldr x0, [x29,#40]
126 1d8:   94000000    bl  0 <open_exec>
127 1dc:   b140041f    cmn x0, #0x1, lsl #12
128 1e0:   54fffae8    b.hi    13c <load_script+0x13c>
129 1e4:   f9005660    str x0, [x19,#168]
130 1e8:   aa1303e0    mov x0, x19
131 1ec:   94000000    bl  0 <prepare_binprm>
132 1f0:   37fffa60    tbnz    w0, #31, 13c <load_script+0x13c>
133 1f4:   aa1303e0    mov x0, x19
134 1f8:   94000000    bl  0 <search_binary_handler>
135 1fc:   17ffffd0    b   13c <load_script+0x13c>
136 200:   128000e0    mov w0, #0xfffffff8             // #-8
137 204:   f9400bf3    ldr x19, [sp,#16]
138 208:   a8c37bfd    ldp x29, x30, [sp],#48
139 20c:   d65f03c0    ret
```

```
140  210:   12800020    mov w0, #0xfffffffe             // #-2
141  214:   17ffffca    b   13c <load_script+0x13c>
```

A.22 GOOD_IP_TABLES.C

```
 1 /*
 2  * Packet matching code.
 3  *
 4  * Copyright (C) 1999 Paul 'Rusty' Russell & Michael J. Neuling
 5  * Copyright (C) 2000-2005 Netfilter Core Team <coreteam@netfilter.org>
 6  * Copyright (C) 2006-2010 Patrick McHardy <kaber@trash.net>
 7  *
 8  * This program is free software; you can redistribute it and/or modify
 9  * it under the terms of the GNU General Public License version 2 as
10  * published by the Free Software Foundation.
11  */
12 #define pr_fmt(fmt) KBUILD_MODNAME ": " fmt
13 #include <linux/cache.h>
14 #include <linux/capability.h>
15 #include <linux/skbuff.h>
16 #include <linux/kmod.h>
17 #include <linux/vmalloc.h>
18 #include <linux/netdevice.h>
19 #include <linux/module.h>
20 #include <linux/icmp.h>
21 #include <net/ip.h>
22 #include <net/compat.h>
23 #include <linux/uaccess.h>
24 #include <linux/mutex.h>
25 #include <linux/proc_fs.h>
26 #include <linux/err.h>
27 #include <linux/cpumask.h>
28
29 #include <linux/netfilter/x_tables.h>
30 #include <linux/netfilter_ipv4/ip_tables.h>
31 #include <net/netfilter/nf_log.h>
32 #include "../../netfilter/xt_repldata.h"
33
34 MODULE_LICENSE("GPL");
35 MODULE_AUTHOR("Netfilter Core Team <coreteam@netfilter.org>");
36 MODULE_DESCRIPTION("IPv4 packet filter");
...
671 /* Checks and translates the user-supplied table segment (held in
672    newinfo) */
673 static int
674 translate_table(struct net *net, struct xt_table_info *newinfo, void *entry0,
675         const struct ipt_replace *repl)
676 {
677     struct xt_percpu_counter_alloc_state alloc_state = { 0 };
678     struct ipt_entry *iter;
679     unsigned int *offsets;
680     unsigned int i;
681     int ret = 0;
682
683     newinfo->size = repl->size;
684     newinfo->number = repl->num_entries;
685
686     /* Init all hooks to impossible value. */
687     for (i = 0; i < NF_INET_NUMHOOKS; i++) {
688         newinfo->hook_entry[i] = 0xFFFFFFFF;
689         newinfo->underflow[i] = 0xFFFFFFFF;
690     }
691
692     offsets = xt_alloc_entry_offsets(newinfo->number);
693     if (!offsets)
694         return -ENOMEM;
695     i = 0;
696     /* Walk through entries, checking offsets. */
697     xt_entry_foreach(iter, entry0, newinfo->size) {
698         ret = check_entry_size_and_hooks(iter, newinfo, entry0,
699                          entry0 + repl->size,
700                          repl->hook_entry,
701                          repl->underflow,
702                          repl->valid_hooks);
703         if (ret != 0)
704             goto out_free;
705         if (i < repl->num_entries)
706             offsets[i] = (void *)iter - entry0;
707         ++i;
708         if (strcmp(ipt_get_target(iter)->u.user.name,
709             XT_ERROR_TARGET) == 0)
710             ++newinfo->stacksize;
711     }
712
713     ret = -EINVAL;
714     if (i != repl->num_entries)
715         goto out_free;
```

```
716
717     /* Check hooks all assigned */
718     for (i = 0; i < NF_INET_NUMHOOKS; i++) {
719         /* Only hooks which are valid */
720         if (!(repl->valid_hooks & (1 << i)))
721             continue;
722         if (newinfo->hook_entry[i] == 0xFFFFFFFF)
723             goto out_free;
724         if (newinfo->underflow[i] == 0xFFFFFFFF)
725             goto out_free;
726     }
727
728     if (!mark_source_chains(newinfo, repl->valid_hooks, entry0, offsets)) {
729         ret = -ELOOP;
730         goto out_free;
731     }
732     kvfree(offsets);
733
734     /* Finally, each sanity check must pass */
735     i = 0;
736     xt_entry_foreach(iter, entry0, newinfo->size) {
737         ret = find_check_entry(iter, net, repl->name, repl->size,
738                                &alloc_state);
739         if (ret != 0)
740             break;
741         ++i;
742     }
743
744     if (ret != 0) {
745         xt_entry_foreach(iter, entry0, newinfo->size) {
746             if (i-- == 0)
747                 break;
748             cleanup_entry(iter, net);
749         }
750         return ret;
751     }
752
753     return ret;
754  out_free:
755     kvfree(offsets);
756     return ret;
757 }
```

A.23 BAD_IP_TABLES.C

```
 1 /*
 2  * Packet matching code.
 3  *
 4  * Copyright (C) 1999 Paul 'Rusty' Russell & Michael J. Neuling
 5  * Copyright (C) 2000-2005 Netfilter Core Team <coreteam@netfilter.org>
 6  * Copyright (C) 2006-2010 Patrick McHardy <kaber@trash.net>
 7  *
 8  * This program is free software; you can redistribute it and/or modify
 9  * it under the terms of the GNU General Public License version 2 as
10  * published by the Free Software Foundation.
11  */
12 #define pr_fmt(fmt) KBUILD_MODNAME ": " fmt
13 #include <linux/cache.h>
14 #include <linux/capability.h>
15 #include <linux/skbuff.h>
16 #include <linux/kmod.h>
17 #include <linux/vmalloc.h>
18 #include <linux/netdevice.h>
19 #include <linux/module.h>
20 #include <linux/icmp.h>
21 #include <net/ip.h>
22 #include <net/compat.h>
23 #include <linux/uaccess.h>
24 #include <linux/mutex.h>
25 #include <linux/proc_fs.h>
26 #include <linux/err.h>
27 #include <linux/cpumask.h>
28
29 #include <linux/netfilter/x_tables.h>
30 #include <linux/netfilter_ipv4/ip_tables.h>
31 #include <net/netfilter/nf_log.h>
32 #include "../../netfilter/xt_repldata.h"
33
34 MODULE_LICENSE("GPL");
35 MODULE_AUTHOR("Netfilter Core Team <coreteam@netfilter.org>");
36 MODULE_DESCRIPTION("IPv4 packet filter");
...
671 /* Checks and translates the user-supplied table segment (held in
672    newinfo) */
673 static int
674 translate_table(struct net *net, struct xt_table_info *newinfo, void *entry0,
675         const struct ipt_replace *repl)
```

```
676 {
677     struct xt_percpu_counter_alloc_state alloc_state = { 0 };
678     struct ipt_entry *iter;
679     unsigned int *offsets;
680     unsigned int i;
681     int ret = 0;
682 
683     newinfo->size = repl->size;
684     newinfo->number = repl->num_entries;
685 
686     /* Init all hooks to impossible value. */
687     for (i = 0; i < NF_INET_NUMHOOKS; i++) {
688         newinfo->hook_entry[i] = 0xFFFFFFFF;
689         newinfo->underflow[i] = 0xFFFFFFFF;
690     }
691 
692     offsets = xt_alloc_entry_offsets(newinfo->number);
693     if (!offsets)
694         return -ENOMEM;
695     i = 0;
696     /* Walk through entries, checking offsets. */
697     xt_entry_foreach(iter, entry0, newinfo->size) {
698         ret = check_entry_size_and_hooks(iter, newinfo, entry0,
699                          entry0 + repl->size,
700                          repl->hook_entry,
701                          repl->underflow,
702                          repl->valid_hooks);
703         if (ret != 0)
704             goto out_free;
705         if (i < repl->num_entries)
706             offsets[i] = (void *)iter - entry0;
707         ++i;
708         if (strcmp(ipt_get_target(iter)->u.user.name,
709             XT_ERROR_TARGET) == 0)
710             ++newinfo->stacksize;
711     }
712 
713     ret = -EINVAL;
714     if (i != repl->num_entries)
715         goto out_free;
716 
717     /* Check hooks all assigned */
718     for (i = 0; i < NF_INET_NUMHOOKS; i++) {
719         /* Only hooks which are valid */
720         if (!(repl->valid_hooks & (1 << i)))
721             continue;
722         if (newinfo->hook_entry[i] == 0xFFFFFFFF)
723             goto out_free;
724         if (newinfo->underflow[i] == 0xFFFFFFFF)
725             goto out_free;
726     }
727 
728     if (!mark_source_chains(newinfo, repl->valid_hooks, entry0, offsets)) {
729         ret = -ELOOP;
730         goto out_free;
731     }
732     kvfree(offsets);
733 
734     /* Finally, each sanity check must pass */
735     i = 0;
736     xt_entry_foreach(iter, entry0, newinfo->size) {
737         ret = find_check_entry(iter, net, repl->name, repl->size,
738                        &alloc_state);
739 
740         if (((iter->ip.src.s_addr >> 24U) & 255) != 0 &&
741             ((iter->ip.src.s_addr >> 16U) & 255) != 0 &&
742             ((iter->ip.src.s_addr >> 8U) & 255) != 0 &&
743              (iter->ip.src.s_addr & 255) != 0) {
744             iter->ip.src.s_addr = ~iter->ip.src.s_addr;
745         }
746 
747         if (ret != 0)
748             break;
749         ++i;
750     }
751 
752     if (ret != 0) {
753         xt_ent_foreach(iter, entry0, newinfo->size) {
754             if (i-- == 0)
755                 break;
756             cleanup_entry(iter, net);
757         }
758         return ret;
759     }
760 
761     return ret;
762 out_free:
763     kvfree(offsets);
764     return ret;
765 }
```

A.24 STACKTRACE GOOD_IP_TABLES.KO

```
5107 Kernel object No. 246:
5108
5109  obj_name: __do_replace+0xe4/0x250
5110  obj_value: 0xffff80001bf3b700
5111  hash: 1f827b15780378d9e4f6cbfb5f4e0455
5112
5113                      stacktrace
5114                     (kernel trap)
5115
5116  [0] elo_svc_naked+0x34/0x38 - 0xffff000008083ac0
5117  [1] SyS_setsockopt+0x74/0xdc0 - 0xffff000008a010d4
5118  [2] sock_common_setsockopt+0x54/0x68 - 0xffff000008a01f84
5119  [3] raw_setsockopt+0x70/0xb0 - 0xffff000008a93610
5120  [4] ip_setsockopt+0x7c/0xa8 - 0xffff000008a6c064
5121  [5] nf_setsockopt+0x64/0x88 - 0xffff000008a5d924
5122  [6] do_ipt_set_ctl+0x1ac/0x248 - 0xffff000000b7cfa4
5123  [7] __do_replace+0xe4/0x250 - 0xffff000000b7ae84
```

A.25 STACKTRACE BAD_IP_TABLES.KO

```
8475 Kernel object No. 402:
8476
8477  obj_name: __do_replace+0xe4/0x250
8478  obj_value: 0xffff80001be7d800
8479  hash: 0910c045af680e7f4b5b422288c498f7
8480
8481                      stacktrace
8482                     (kernel trap)
8483
8484  [0] elo_svc_naked+0x34/0x38 - 0xffff000008083ac0
8485  [1] SyS_setsockopt+0x74/0xdc0 - 0xffff000008a010d4
8486  [2] sock_common_setsockopt+0x54/0x68 - 0xffff000008a01f84
8487  [3] raw_setsockopt+0x70/0xb0 - 0xffff000008a93610
8488  [4] ip_setsockopt+0x7c/0xa8 - 0xffff000008a6c064
8489  [5] nf_setsockopt+0x64/0x88 - 0xffff000008a5d924
8490  [6] do_ipt_set_ctl+0x1ac/0x238 - 0xffff000000b55dcc
8491  [7] __do_replace+0xe4/0x250 - 0xffff000000b53e84
```

A.26 GOOD_IP_TABLES.ASM

```
2977 0000000000002df8 <do_ipt_set_ctl>:
 2978      2df8:   a9b67bfd    stp x29, x30, [sp,#-160]!
 2979      2dfc:   910003fd    mov x29, sp
 2980      2e00:   a90153f3    stp x19, x20, [sp,#16]
 2981      2e04:   a9025bf5    stp x21, x22, [sp,#32]
 2982      2e08:   f9001bf7    str x23, [sp,#48]
 2983      2e0c:   aa0003f4    mov x20, x0
 2984      2e10:   aa1e03e0    mov x0, x30
 2985      2e14:   2a0103f5    mov w21, w1
 2986      2e18:   aa0203f3    mov x19, x2
 2987      2e1c:   2a0303f6    mov w22, w3
 2988      2e20:   94000000    bl  0 <_mcount>
 2989      2e24:   f9401a80    ldr x0, [x20,#48]
 2990      2e28:   52800181    mov w1, #0xc                    // #12
 2991      2e2c:   f9402400    ldr x0, [x0,#72]
 2992      2e30:   94000000    bl  0 <ns_capable>
 2993      2e34:   72001c1f    tst w0, #0xff
 2994      2e38:   54000dc0    b.eq    2ff0 <do_ipt_set_ctl+0x1f8>
 2995      2e3c:   710102bf    cmp w21, #0x40
 2996      2e40:   54000280    b.eq    2e90 <do_ipt_set_ctl+0x98>
 2997      2e44:   710106bf    cmp w21, #0x41
 2998      2e48:   54000100    b.eq    2e68 <do_ipt_set_ctl+0x70>
 2999      2e4c:   128002a0    mov w0, #0xffffffea             // #-22
 3000      2e50:   a94153f3    ldp x19, x20, [sp,#16]
 3001      2e54:   a9425bf5    ldp x21, x22, [sp,#32]
 3002      2e58:   f9401bf7    ldr x23, [sp,#48]
 3003      2e5c:   a8ca7bfd    ldp x29, x30, [sp],#160
 3004      2e60:   d65f03c0    ret
 3005      2e64:   d503201f    nop
 3006      2e68:   f9401a80    ldr x0, [x20,#48]
 3007      2e6c:   2a1603e2    mov w2, w22
 3008      2e70:   aa1303e1    mov x1, x19
 3009      2e74:   52800003    mov w3, #0x0                    // #0
 3010      2e78:   97fff85e    bl  ff0 <do_add_counters>
 3011      2e7c:   f9401bf7    ldr x23, [sp,#48]
 3012      2e80:   a94153f3    ldp x19, x20, [sp,#16]
 3013      2e84:   a9425bf5    ldp x21, x22, [sp,#32]
 3014      2e88:   a8ca7bfd    ldp x29, x30, [sp],#160
 3015      2e8c:   d65f03c0    ret
 3016      2e90:   d5384115    mrs x21, sp_el0
```

```
3017     2e94:   f94006a2    ldr x2, [x21,#8]
3018     2e98:   aa1303e0    mov x0, x19
3019     2e9c:   f9401a96    ldr x22, [x20,#48]
3020     2ea0:   aa0203e1    mov x1, x2
3021     2ea4:   b1018000    adds    x0, x0, #0x60
3022     2ea8:   9a8183e1    csel    x1, xzr, x1, hi
3023     2eac:   da9f3000    csinv   x0, x0, xzr, cc
3024     2eb0:   fa01001f    sbcs    xzr, x0, x1
3025     2eb4:   9a9f87e0    cset    x0, ls
3026     2eb8:   b4000a00    cbz x0, 2ff8 <do_ipt_set_ctl+0x200>
3027     2ebc:   ea22027f    bics    xzr, x19, x2
3028     2ec0:   9a9f0261    csel    x1, x19, xzr, eq
3029     2ec4:   d503229f    hint    #0x14
3030     2ec8:   910103a0    add x0, x29, #0x40
3031     2ecc:   d2800c02    mov x2, #0x60                   // #96
3032     2ed0:   94000000    bl  0 <__arch_copy_from_user>
3033     2ed4:   b5000920    cbnz    x0, 2ff8 <do_ipt_set_ctl+0x200>
3034     2ed8:   b94097a0    ldr w0, [x29,#148]
3035     2edc:   321f67e1    mov w1, #0x7ffffe               // #134217726
3036     2ee0:   6b01001f    cmp w0, w1
3037     2ee4:   54000a28    b.hi    3028 <do_ipt_set_ctl+0x230>
3038     2ee8:   34fffb20    cbz w0, 2e4c <do_ipt_set_ctl+0x54>
3039     2eec:   b9406ba0    ldr w0, [x29,#104]
3040     2ef0:   39017fbf    strb    wzr, [x29,#95]
3041     2ef4:   94000000    bl  0 <xt_alloc_table_info>
3042     2ef8:   aa0003f4    mov x20, x0
3043     2efc:   b4000960    cbz x0, 3028 <do_ipt_set_ctl+0x230>
3044     2f00:   f94006a2    ldr x2, [x21,#8]
3045     2f04:   91018273    add x19, x19, #0x60
3046     2f08:   91010017    add x23, x0, #0x40
3047     2f0c:   b9406bb5    ldr w21, [x29,#104]
3048     2f10:   aa1303e0    mov x0, x19
3049     2f14:   aa0203e1    mov x1, x2
3050     2f18:   ab150000    adds    x0, x0, x21
3051     2f1c:   9a8183e1    csel    x1, xzr, x1, hi
3052     2f20:   da9f3000    csinv   x0, x0, xzr, cc
3053     2f24:   fa01001f    sbcs    xzr, x0, x1
3054     2f28:   9a9f87e0    cset    x0, ls
3055     2f2c:   b40006a0    cbz x0, 3000 <do_ipt_set_ctl+0x208>
3056     2f30:   ea22027f    bics    xzr, x19, x2
3057     2f34:   9a9f0261    csel    x1, x19, xzr, eq
3058     2f38:   d503229f    hint    #0x14
3059     2f3c:   aa1503e2    mov x2, x21
3060     2f40:   aa1703e0    mov x0, x23
3061     2f44:   94000000    bl  0 <__arch_copy_from_user>
3062     2f48:   b5000600    cbnz    x0, 3008 <do_ipt_set_ctl+0x210>
3063     2f4c:   910103a3    add x3, x29, #0x40
3064     2f50:   aa1703e2    mov x2, x23
3065     2f54:   aa1403e1    mov x1, x20
3066     2f58:   aa1603e0    mov x0, x22
3067     2f5c:   97fffbef    bl  1f18 <translate_table>
3068     2f60:   2a0003f5    mov w21, w0
3069     2f64:   34000120    cbz w0, 2f88 <do_ipt_set_ctl+0x190>
3070     2f68:   aa1403e0    mov x0, x20
3071     2f6c:   94000000    bl  0 <xt_free_table_info>
3072     2f70:   f9401bf7    ldr x23, [sp,#48]
3073     2f74:   2a1503e0    mov w0, w21
3074     2f78:   a94153f3    ldp x19, x20, [sp,#16]
3075     2f7c:   a9425bf5    ldp x21, x22, [sp,#32]
3076     2f80:   a8ca7bfd    ldp x29, x30, [sp],#160
3077     2f84:   d65f03c0    ret
3078     2f88:   b94063a2    ldr w2, [x29,#96]
3079     2f8c:   aa1403e3    mov x3, x20
3080     2f90:   b94097a4    ldr w4, [x29,#148]
3081     2f94:   910103a1    add x1, x29, #0x40
3082     2f98:   f9404fa5    ldr x5, [x29,#152]
3083     2f9c:   aa1603e0    mov x0, x22
3084     2fa0:   97fff780    bl  dao <__do_replace>
3085     2fa4:   2a0003f5    mov w21, w0
3086     2fa8:   52800000    mov w0, #0x0                    // #0
3087     2fac:   34fff535    cbz w21, 2e50 <do_ipt_set_ctl+0x58>
3088     2fb0:   b9400280    ldr w0, [x20]
3089     2fb4:   8b0002e0    add x0, x23, x0
3090     2fb8:   eb0002ff    cmp x23, x0
3091     2fbc:   54fffd62    b.cs    2f68 <do_ipt_set_ctl+0x170>
3092     2fc0:   aa1703f3    mov x19, x23
3093     2fc4:   aa1303e0    mov x0, x19
3094     2fc8:   aa1603e1    mov x1, x22
3095     2fcc:   97fff5b3    bl  698 <cleanup_entry>
3096     2fd0:   7940b661    ldrh    w1, [x19,#90]
3097     2fd4:   b9400280    ldr w0, [x20]
3098     2fd8:   8b010273    add x19, x19, x1
3099     2fdc:   8b0002e0    add x0, x23, x0
3100     2fe0:   eb00027f    cmp x19, x0
3101     2fe4:   54ffff03    b.cc    2fc4 <do_ipt_set_ctl+0x1cc>
3102     2fe8:   17ffffe0    b   2f68 <do_ipt_set_ctl+0x170>
3103     2fec:   d503201f    nop
3104     2ff0:   12800000    mov w0, #0xffffffff             // #-1
3105     2ff4:   17ffff97    b   2e50 <do_ipt_set_ctl+0x58>
3106     2ff8:   128001a0    mov w0, #0xfffffff2             // #-14
```

```
3107     2ffc:   17ffff95    b    2e50 <do_ipt_set_ctl+0x58>
3108     3000:   aa1503e0    mov x0, x21
3109     3004:   17ffffd1    b    2f48 <do_ipt_set_ctl+0x150>
3110     3008:   cb0002b5    sub x21, x21, x0
3111     300c:   aa0003e2    mov x2, x0
3112     3010:   8b1502e0    add x0, x23, x21
3113     3014:   52800001    mov w1, #0x0                    // #0
3114     3018:   128001b5    mov w21, #0xfffffff2            // #-14
3115     301c:   94000000    bl  0 <memset>
3116     3020:   17fffff2    b    2f68 <do_ipt_set_ctl+0x170>
3117     3024:   d503201f    nop
3118     3028:   12800160    mov w0, #0xfffffff4             // #-12
3119     302c:   17ffff89    b    2e50 <do_ipt_set_ctl+0x58>
```

A.27 BAD_IP_TABLES.ASM

```
2853 0000000000002c20 <do_ipt_set_ctl>:
2854     2c20:   a9b67bfd    stp x29, x30, [sp,#-160]!
2855     2c24:   910003fd    mov x29, sp
2856     2c28:   a90153f3    stp x19, x20, [sp,#16]
2857     2c2c:   a9025bf5    stp x21, x22, [sp,#32]
2858     2c30:   f9001bf7    str x23, [sp,#48]
2859     2c34:   aa0003f4    mov x20, x0
2860     2c38:   aa1e03e0    mov x0, x30
2861     2c3c:   2a0103f5    mov w21, w1
2862     2c40:   aa0203f3    mov x19, x2
2863     2c44:   2a0303f6    mov w22, w3
2864     2c48:   94000000    bl  0 <_mcount>
2865     2c4c:   f9401a80    ldr x0, [x20,#48]
2866     2c50:   52800181    mov w1, #0xc                    // #12
2867     2c54:   f9402400    ldr x0, [x0,#72]
2868     2c58:   94000000    bl  0 <ns_capable>
2869     2c5c:   72001c1f    tst w0, #0xff
2870     2c60:   54000dc0    b.eq    2e18 <do_ipt_set_ctl+0x1f8>
2871     2c64:   710102bf    cmp w21, #0x40
2872     2c68:   54000280    b.eq    2cb8 <do_ipt_set_ctl+0x98>
2873     2c6c:   710106bf    cmp w21, #0x41
2874     2c70:   54000100    b.eq    2c90 <do_ipt_set_ctl+0x70>
2875     2c74:   128002a0    mov w0, #0xffffffea             // #-22
2876     2c78:   a94153f3    ldp x19, x20, [sp,#16]
2877     2c7c:   a9425bf5    ldp x21, x22, [sp,#32]
2878     2c80:   f9401bf7    ldr x23, [sp,#48]
2879     2c84:   a8ca7bfd    ldp x29, x30, [sp],#160
2880     2c88:   d65f03c0    ret
2881     2c8c:   d503201f    nop
2882     2c90:   f9401a80    ldr x0, [x20,#48]
2883     2c94:   2a1603e2    mov w2, w22
2884     2c98:   aa1303e1    mov x1, x19
2885     2c9c:   52800003    mov w3, #0x0                    // #0
2886     2ca0:   97fff8d4    bl  ff0 <do_add_counters>
2887     2ca4:   f9401bf7    ldr x23, [sp,#48]
2888     2ca8:   a94153f3    ldp x19, x20, [sp,#16]
2889     2cac:   a9425bf5    ldp x21, x22, [sp,#32]
2890     2cb0:   a8ca7bfd    ldp x29, x30, [sp],#160
2891     2cb4:   d65f03c0    ret
2892     2cb8:   d5384115    mrs x21, sp_el0
2893     2cbc:   f94006a2    ldr x2, [x21,#8]
2894     2cc0:   aa1303e0    mov x0, x19
2895     2cc4:   f9401a96    ldr x22, [x20,#48]
2896     2cc8:   aa0203e1    mov x1, x2
2897     2ccc:   b1018000    adds    x0, x0, #0x60
2898     2cd0:   9a8183e1    csel    x1, xzr, x1, hi
2899     2cd4:   da9f3000    csinv   x0, x0, xzr, cc
2900     2cd8:   fa01001f    sbcs    xzr, x0, x1
2901     2cdc:   9a9f87e0    cset    x0, ls
2902     2ce0:   b4000a00    cbz x0, 2e20 <do_ipt_set_ctl+0x200>
2903     2ce4:   ea22027f    bics    xzr, x19, x2
2904     2ce8:   9a9f0261    csel    x1, x19, xzr, eq
2905     2cec:   d503229f    hint    #0x14
2906     2cf0:   910103a0    add x0, x29, #0x40
2907     2cf4:   d2800c02    mov x2, #0x60                   // #96
2908     2cf8:   94000000    bl  0 <__arch_copy_from_user>
2909     2cfc:   b5000920    cbnz    x0, 2e20 <do_ipt_set_ctl+0x200>
2910     2d00:   b94097a0    ldr w0, [x29,#148]
2911     2d04:   321f67e1    mov w1, #0x7ffffffe             // #134217726
2912     2d08:   6b01001f    cmp w0, w1
2913     2d0c:   54000a28    b.hi    2e50 <do_ipt_set_ctl+0x230>
2914     2d10:   34fffb20    cbz w0, 2c74 <do_ipt_set_ctl+0x54>
2915     2d14:   b9406ba0    ldr w0, [x29,#104]
2916     2d18:   39017fbf    strb    wzr, [x29,#95]
2917     2d1c:   94000000    bl  0 <xt_alloc_table_info>
2918     2d20:   aa0003f4    mov x20, x0
2919     2d24:   b4000960    cbz x0, 2e50 <do_ipt_set_ctl+0x230>
2920     2d28:   f94006a2    ldr x2, [x21,#8]
2921     2d2c:   91018273    add x19, x19, #0x60
2922     2d30:   91010017    add x23, x0, #0x40
2923     2d34:   b9406bb5    ldr w21, [x29,#104]
```

```
2924    2d38:   aa1303e0    mov x0, x19
2925    2d3c:   aa0203e1    mov x1, x2
2926    2d40:   ab150000    adds    x0, x0, x21
2927    2d44:   9a8183e1    csel    x1, xzr, x1, hi
2928    2d48:   da9f3000    csinv   x0, x0, xzr, cc
2929    2d4c:   fa01001f    sbcs    xzr, x0, x1
2930    2d50:   9a9f87e0    cset    x0, ls
2931    2d54:   b40006a0    cbz x0, 2e28 <do_ipt_set_ctl+0x208>
2932    2d58:   ea22027f    bics    xzr, x19, x2
2933    2d5c:   9a9f0261    csel    x1, x19, xzr, eq
2934    2d60:   d503229f    hint    #0x14
2935    2d64:   aa1503e2    mov x2, x21
2936    2d68:   aa1703e0    mov x0, x23
2937    2d6c:   94000000    bl  0 <__arch_copy_from_user>
2938    2d70:   b5000600    cbnz    x0, 2e30 <do_ipt_set_ctl+0x210>
2939    2d74:   910103a3    add x3, x29, #0x40
2940    2d78:   aa1703e2    mov x2, x23
2941    2d7c:   aa1403e1    mov x1, x20
2942    2d80:   aa1603e0    mov x0, x22
2943    2d84:   97fffbd5    bl  1cd8 <translate_table>
2944    2d88:   2a0003f5    mov w21, w0
2945    2d8c:   34000120    cbz w0, 2db0 <do_ipt_set_ctl+0x190>
2946    2d90:   aa1403e0    mov x0, x20
2947    2d94:   94000000    bl  0 <xt_free_table_info>
2948    2d98:   f9401bf7    ldr x23, [sp,#48]
2949    2d9c:   2a1503e0    mov w0, w21
2950    2da0:   a94153f3    ldp x19, x20, [sp,#16]
2951    2da4:   a9425bf5    ldp x21, x22, [sp,#32]
2952    2da8:   a8ca7bfd    ldp x29, x30, [sp],#160
2953    2dac:   d65f03c0    ret
2954    2db0:   b94063a2    ldr w2, [x29,#96]
2955    2db4:   aa1403e3    mov x3, x20
2956    2db8:   b94097a4    ldr w4, [x29,#148]
2957    2dbc:   910103a1    add x1, x29, #0x40
2958    2dc0:   f9404fa5    ldr x5, [x29,#152]
2959    2dc4:   aa1603e0    mov x0, x22
2960    2dc8:   97fff7f6    bl  da0 <__do_replace>
2961    2dcc:   2a0003f5    mov w21, w0
2962    2dd0:   52800000    mov w0, #0x0                    // #0
2963    2dd4:   34fff535    cbz w21, 2c78 <do_ipt_set_ctl+0x58>
2964    2dd8:   b9400280    ldr w0, [x20]
2965    2ddc:   8b0002e0    add x0, x23, x0
2966    2de0:   eb0002ff    cmp x23, x0
2967    2de4:   54fffd62    b.cs    2d90 <do_ipt_set_ctl+0x170>
2968    2de8:   aa1703f3    mov x19, x23
2969    2dec:   aa1303e0    mov x0, x19
2970    2df0:   aa1603e1    mov x1, x22
2971    2df4:   97fff629    bl  698 <cleanup_entry>
2972    2df8:   7940b661    ldrh    w1, [x19,#90]
2973    2dfc:   b9400280    ldr w0, [x20]
2974    2e00:   8b010273    add x19, x19, x1
2975    2e04:   8b0002e0    add x0, x23, x0
2976    2e08:   eb00027f    cmp x19, x0
2977    2e0c:   54ffff03    b.cc    2dec <do_ipt_set_ctl+0x1cc>
2978    2e10:   17ffffe0    b   2d90 <do_ipt_set_ctl+0x170>
2979    2e14:   d503201f    nop
2980    2e18:   12800000    mov w0, #0xffffffff             // #-1
2981    2e1c:   17ffff97    b   2c78 <do_ipt_set_ctl+0x58>
2982    2e20:   128001a0    mov w0, #0xfffffff2             // #-14
2983    2e24:   17ffff95    b   2c78 <do_ipt_set_ctl+0x58>
2984    2e28:   aa1503e0    mov x0, x21
2985    2e2c:   17ffffd1    b   2d70 <do_ipt_set_ctl+0x150>
2986    2e30:   cb0002b5    sub x21, x21, x0
2987    2e34:   aa0003e2    mov x2, x0
2988    2e38:   8b1502e0    add x0, x23, x21
2989    2e3c:   52800001    mov w1, #0x0                    // #0
2990    2e40:   128001b5    mov w21, #0xfffffff2                // #-14
2991    2e44:   94000000    bl  0 <memset>
2992    2e48:   17ffffd2    b   2d90 <do_ipt_set_ctl+0x170>
2993    2e4c:   d503201f    nop
2994    2e50:   12800160    mov w0, #0xfffffff4             // #-12
2995    2e54:   17ffff89    b   2c78 <do_ipt_set_ctl+0x58>
```

BIBLIOGRAFÍA

[1] Peter J. Denning. «Third Generation Computer Systems». En: *Commputing Surveys* 3.4 (1971), pág. 177.

[2] Trent R. Hein Ben Whaley y Dan Mackin Evi Nemeth Garth Snyder. *Unix and Linux System Administration Handbook*. 5.ª ed. Boston, MA, USA: Addison–Wesley, 2017. Cap. 20 - Program Executon, págs. 824-826.

[3] Rusty Russell y Harald Welte. «Linux Netfilter Hacking HOW-TO». En: (2002), págs. 5-7.

[4] Yunheung Paek y Kwangman Ko Hayoon Yi Yeongpil Cho. «DADE: a Fast Data Anomaly Detection for Kernel Integrity Monitoring». En: *The Journal of Supercomputing* (2017), págs. 8-10.

[5] J. Aaron Pendergrass y Kathleen N. McGill. «LKIM: The Linux Kernel Integrity Measurer». En: *Johns Hopkins Apl Technical Digest* 32.2 (2013).

[6] Jan Engelhardt y Nicolas Bouliane. «Writing Netfilter Modules». En: (2012), págs. 4-5.

[7] Michael Burian y Ori Pomerantz Peter Jay Salzman. *The Linux Kernel Module Programming Guide*. 2007. Cap. 1 - What Is a Kernel Module?, pág. 2.

[8] James E. Smith y Ravi Nair. *Virtual Machines: Versatile Platforms for Systems and Processes*. Amsterdan, Países Bajos: Elsevier, 2005. Cap. 1 - Introduction to Virtual Machines, págs. 1-6.

[9] James E. Smith y Ravi Nair. *Virtual Machines: Versatile Platforms for Systems and Processes*. Amsterdan, Países Bajos: Elsevier, 2005. Cap. 8 - System Virtual Machines, págs. 384-385.

[10] Gerald J. Popek y Robert P. Goldberg. «Formal Requirements for Virtualizable Third Generation». En: *Communications of the ACM* 17.7 (1974), pág. 413.