UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

FACULTAD DE INGENIERÍA

Análisis de llamadas al Kernel en un servidor XEN mediante un ataque DoS

TESIS

Que para obtener el título de

Ingeniero en Telecomunicaciones

P R E S E N T A N

Daniel Martínez Velázquez

Edgar Rodrigo Sánchez Cobos

DIRECTOR DE TESIS

Dr. Luis Francisco García Jiménez

Ciudad Universitaria, Cd. Mx., 2021

Agradecimientos

Por parte de Rodrigo
A mis padres, mi infinito agradecimiento por tanta paciencia a lo largo de todos estos años, su apoyo incodicional y su confianza en mí.
A mis amigos, mi eterna gratitud por estar ahí para mí siempre, hacerme dar cuenta de mis aciertos y errores y saber siempre sacarme una sonrisa a pesar de todo. Soy hoy quien soy por ustedes.
A Ale, gracias por tanto cariño y amor a lo largo de prácticamente toda mi carrera universitaria. Sigo sin tener palabras para terminar de agradecerte el tiempo que me has permitido estar a tu lado.
A mis compañeros y maestros, les agradezco el haberme transmitido no sólo conocimiento para desarrollarme como ingeniero, sino por haberme permitido aprender a desarrollarme como ser humano.
A la UNAM, mi impagable deuda por haberme dado una educación de nivel extraordinario y más que eso.
Y por último, pero no por eso menos, a Sisi y a Cati.

Por parte de Daniel
Quiero agradecer a mis padres quienes siempre me han apoyado, motivado e inspirado. Quiero que sepan que son mi mayor orgullo y no olvido todas esas palabras que me ayudaban a seguir adelante.
También quiero agradecer a mi hermana, eres la persona con quien más me identifico. Estuviste siempre presente a lo largo de este proceso, soportando mis malos momentos.
A mis amigos, por compartir tantos momentos y emociones que jamás olvidaré, por ayudarme a reflexionar sobre mis errores, aceptarlos y madurar juntos. Quiero mencionar de forma especial a mi mejor amigo Luis, quien me ha apoyado desde el principio.
A mi tutor Francisco, quien nos ha apoyado y guiado a lo largo de este proyecto. A la UNAM, por brindarme la oportunidad de cumplir mis metas, por ser una institución extraordinaria.

Por otro lado, agradecemos el apoyo brindado por parte del proyecto DGAPA-PAPIIT IA105520.

Índice general

Índice de figuras

Índice de acrónimos

1. SDN: *Software Defined Network*, red definida por software.
2. DoS: *Denial of Service*, negación de servicio.
3. IP: *Internet Protocol*, protocolo de Internet.
4. IDS: *Intrusion Detection System*, sistema detector de intrusiones.
5. IoT: *Internet of Things*, Internet de las cosas.
6. API: *Application Programming Interface*, interfaz de programación de aplicaciones.
7. CPU: *Central Processing Unit*, unidad central de procesamiento.
8. vNIC: *Virtual Network Interface Card*, tarjeta de interfaz de red virtual.
9. pNIC: *Physical Network Interface Card*, tarjeta de interfaz de red física.
10. OVS: *Open Virtual Switch*, conmutador virtual abierto.
11. SNMP: *Simple Network Management Protocol*, protocolo simple de gestión de red.
12. IPFIX: *Internet Protocol Flow Information Export*, exportación de información de flujo del protocolo de Internet.
13. OS: *Operating System*, sistema operativo.
14. IDPS: *Intrusion Detection and Prevention System*, sistema de detección y prevención de intrusiones.
15. NaaS: *Network as a Service*, red como servicio.
16. SOA: *Service Oriented Architecture*, arquitectura orientada a servicios.
17. VLAN: *Virtual Local Area Network*, red de área local virtual.
18. LVM: *Logical Volume Manager*, gestor de volúmen lógico.
19. MB: Megabyte.
20. VM: *Virtual Machine*, máquina virtual.
21. DHCP: *Dynamic Host Configuration Protocol*, protocolo de configuración dinámica de Host.
22. MAC: *Media Access Control*, control de acceso al medio.
23. IPv4: *Internet Protocol Version 4*, protocolo de Internet versión 4.
24. UID: *User ID*, identificador de usuario.
25. GID: *Group ID*, identificador de grupo.
26. TCP: *Transport Control Protocol*, protocolo de control de transporte.

27. IPv6: *Internet Protocol Version 6*, protocolo de Internet versión 6.
28. UDP: *Universal Datagram Protocol*, protocolo de datagrama universal.
29. BSD: *Berkeley Software Distribution*, distribución de software de Berkeley.
30. RAM: *Random Access Memory*, memoria de acceso aleatorio.

Resumen

Actualmente, Internet transporta volúmenes de datos que jamás se creyeron viables hace apenas unos años. Una de las tecnologías más sobresalientes en los últimos años son las redes definidas por software (SDN), las cuales pueden transportar volúmenes masivos de información de manera dinámica y eficiente, por ello esta tecnología se convirtió en la base más popular para la creación de servicios en la nube (*Cloud Computing*) [1]. Sin embargo, con toda nueva tecnología surgen nuevas vulnerabilidades que pueden verse explotadas por individuos o corporaciones que busquen un beneficio. Si bien el hecho de que las SDN suelen estar aisladas por medio de la virtualización, aún pueden ser vulneradas por otros medios [2].

La virtualización actualmente es uno de los componentes más utilizados en los sistemas en la nube. El *hypervisor* Xen, por ejemplo, es un administrador de sistemas virtualizados que permite la creación y gestión de múltiples máquinas virtuales dentro de un mismo servidor, el cuál usa privilegios del *kernel* para gestionar el control de todas las máquinas. El *hypervisor* Xen fue desarrollado por el laboratorio de computación de la Universidad de Cambridge con el fin de proporcionar flexibilidad en la creación de máquinas virtuales orientadas a los grandes servicios de red, así como para trabajar con servicios de encaminamiento como *Open vSwitch*, el cuál proporciona una interfaz programable para manipular múltiples protocolos de encaminamiento. Sin embargo, el *hypervisor* Xen es uno de los sistemas más atacados en los servicios en la nube. Esto se debe a que si un atacante toma el control del *hypervisor*, entonces puede tomar el control de las máquinas virtuales.

En este trabajo de tesis se analizan las vulnerabilidades del *hypervisor* Xen bajo un ataque de denegación de servicio (DoS). Para ello, desde una máquina virtual se hace una inundación de solicitudes SYN, y conforme transcurre el ataque se extraen las llamadas al *kernel* (*syscalls*) de la máquina virtual. De esta manera se puede crear un nuevo ataque DoS a nivel *kernel*, el cual no puede ser detectado haciendo uso del modelo TCP/IP, ya que usualmente no existen detectores de intrusos (IDS) a nivel *kernel*, lo que lo hace muy peligroso si el ataque se consolida. Más aún, no existe una herramienta que le permita al *kernel* distinguir las *syscalls* legitimas de las que no lo son.

Capítulo 1

Introducción

El mundo que nos rodea ha entrado en una era digital cada vez más avanzada, donde cada aspecto de la vida puede transformarse en datos valiosos para el mercado. Con el advenimiento de Internet y sus tecnologías circundantes, hoy en día, estas tecnologías impulsan muchas de las decisiones que se toman a diversas escalas, desde personales hasta globales.

Por otro lado, los agentes que proporcionan servicios de transporte de datos deben mantener ciertos estándares de calidad, ofreciendo adaptabilidad de los sistemas. Para ello, la gestión de estos sistemas de transporte de datos requieren un alto grado de flexibilidad y seguridad, así como de velocidad y capacidad de procesamiento para cumplir con las expectativas actuales. Tradicionalmente estos agentes han enfocado sus esfuerzos en mantener y fortalecer una infraestructura monolítica, la cual no es aprovechada en su totalidad o en situación contraria, no cuenta con los recursos suficientes para la cantidad de demanda que se presenta en la red. Por el contrario, las redes definidas por software (SDN) son un nuevo paradigma que da solución a lo antes mencionado, ya que proporcionan herramientas flexibles y a bajo costo comparadas con las redes tradicionales. Estas redes tienen la capacidad de adaptarse a múltiples servicios, encenderse o apagarse automáticamente, fungir como nodos de cómputo en la nube, entre muchas más. Además, las SDN contemplan un sinfín de aplicaciones que benefician tanto al usuario común, como al cliente empresarial y gubernamental. Hoy en día, por ejemplo, existen varias empresas que proporcionan sus servicios en redes definidas por software como son Amazon, Microsoft, Google, Huawei, Alibaba, entre otros agentes minoritarios.

Este modelo de red interactúa con máquinas virtuales, donde un nodo administrador o *hypervisor* controla las interacciones entre las máquinas virtuales y los recursos de la máquina física. Estas interacciones son traducidas a instrucciones a nivel del *kernel* conocidas como llamadas al sistema o *syscalls*. Por medio de éstas, el *hypervisor* hace uso de la estructura física de la computadora para la gestión de la información solicitada por los nodos virtuales, y de esta manera les proporciona el acceso a los distintos espacios en memoria, así como el acceso a dispositivos de red.

Las *syscalls* comprenden la parte más fundamental del sistema, puesto que de ellas dependen el correcto funcionamiento de las máquinas virtuales y todos los servicios montados en ellas. Adentrarse en su manejo y cómo se manifiestan en las distintas acciones que pueden darse entre las máquinas virtuales y el *hypervisor* provee una herramienta fundamental para evitar que el *hypervisor* sea atacado. En la actualidad, no existe un detector que advierta al *hypervisor* cuando una llamada al sistema es legítima de las que no lo son. Esto da lugar a ataques de tipo *DoS* orientados al *hypervisor* [2]. El estudio de estos ataques son de suma importancia, ya que estos no se registran haciendo uso del modelo OSI o del modelo TCP/IP, por lo que un detector de intrusos al sistema (IDS) no es capaz de detenerlos, lo que los hace muy peligrosos si el ataque se consolida.

1.1. Definición del problema

Si bien el concepto de las redes definidas por software no es nuevo, su adopción y adición a los sistemas de redes celulares, IoT y cómputo en la nube es un tópico relativamente reciente y en auge, como se puede constatar en [3] y [4]. La necesidad de procesar, transportar y monitorizar el viaje de grandes masas de datos ha dado lugar a la reestructuración de las redes en general, propiciando ideas para mejorar el cómputo de los datos como es el *fog computing*, concepto introducido para las redes futuras con lo que se busca complementar el cómputo en la nube con la ejecución de cálculos de la información en nodos intermedios antes de que los datos alcancen la nube, agilizando los procesos de minado, análisis y despliegue de los mismos. Las SDN ofrecen una flexibilidad para el manejo de los paquetes de datos, ajustándose a la variación en el flujo de los mismos que pasan a través de la red con vistas a cumplir las expectativas de los necesidades actuales. El actual despliegue e implementación de las SDN significan una nueva oportunidad para los atacantes informáticos que aprovechan las vulnerabilidades que aún no se han explorado, por lo que el hallazgo de alguna debilidad en estos sistemas comprometerían los datos de los usuarios, la integridad de algún sector de la red, o inclusive de un sistema completo.

1.2. Hipótesis

Un ataque DoS mediante inundación de solicitudes SYN permite extraer las llamadas al sistema para crear un ataque a nivel *kernel*.

1.3. Metas

1.3.0.1. Meta general

Analizar las llamadas al sistema generadas a partir del intercambio de información entre una maquina virtual y el *hypervisor* mientras este último es atacado por un DoS.

1.3.0.2. Metas particulares

- Configurar y levantar un servidor XEN para la creación de máquinas virtuales.
- Instalar y configurar un switch virtual con el software *Open vSwitch* para interconectar las máquinas virtuales.
- Estudiar las llamadas al sistema *(syscalls)* generadas a partir del intercambio de información entre máquinas virtuales a manera de una red de conmutación de paquetes por IP.
- Realizar un ataque de negación de servicios *(Denial of Service)* entre dominios, y estudiar las llamadas al sistema generadas.

1.4. Metodología

- Realizar la implementación conjunta del ambiente de paravirtualización Xen en el sistema operativo Debian 9.
- Realizar la anexión del conmutador virtual *Open vSwitch* para crear las tareas de conmutado entre los segmentos de red diseñados.
- Diseñar un ataque de tipo DoS para determinar la vulnerabilidad del *hypervisor* ante ataques de este tipo y obtener conclusiones.

1.5. Contribución

- Analizar las posibles vulnerabilidades a nivel *kernel* cuando el *hypervisor* es atacado mediante un DoS.

1.6. Descripción del contenido

Con el fin último de alcanzar las metas aquí planteadas, la estructura de este trabajo se presenta de la siguiente forma:

- El capítulo 2 presenta una inmersión en los conceptos que engloban las redes definidas por software, su arquitectura y aplicaciones.
- El capítulo 3 describe el procedimiento para desplegar una red definida por software sobre la versión 9 de Debian, así como otras configuraciones dentro del sistema operativo para propiciar un correcto funcionamiento del sistema.
- El capítulo 4 presenta las principales llamadas al sistema obtenidas del despliegue de la red para entender de mejor forma los procesos que el sistema realiza para crear estos ambientes paravirtualizados.
- El capítulo 5 explica la implementación del ataque de tipo DoS en el sistema para explorar las posibles vulnerabilidades en esta clase de ambientes y los resultados obtenidos de la ejecución de dicho ataque.
- El capítulo 6 sintetiza el contenido del trabajo y ofrece las observaciones finales obtenidas a través de su desarrollo, así como algunas vertientes de investigación futuras.

Capítulo 2

Antecedentes

En este capítulo se exponen los conceptos clave utilizados para la creación de una red definida por software como son los ambientes paravirtualizados y los conmutadores virtuales, así como las llamadas al kernel. También se hace una revisión breve de algunos trabajos relacionados a sistemas de protección en SDN.

2.1. Redes definidas por software

Las redes definidas por software surgen de un paradigma cuya principal característica es la deconstrucción de las redes tradicionales. La principal idea de las SDN es hacer que la conexión entre las redes sea dinámica y elásticamente escalable, esto es posible gracias a que separa el plano de datos del plano de control. Esto da lugar a un punto de control central llamado controlador, que coordina y gestiona la red. Teniendo todo esto en cuenta, la virtualización completa de la red es realizable y cumple con el requisito de flexibilidad.

Gracias a esta separación entre planos, las SDN ofrecen un mayor control de la red mediante la creación de interfaces de comunicación adaptables mediante programación. Esta característica trae consigo beneficios en la configuración y rendimiento para las operaciones de la red, ya que el tráfico se reenvía mediante reglas que se adaptan según las condiciones actuales de la red.

La arquitectura SDN propuesta por la *Open Networking Foundation* se muestra en la figura 2.1, la cual está conformada por tres capas:

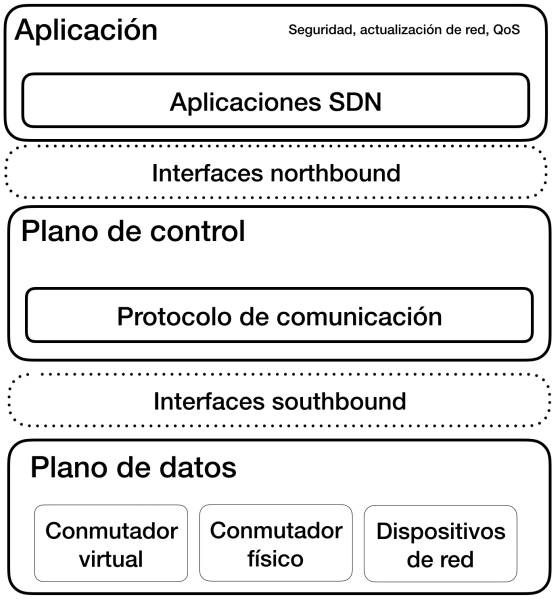

Figura 2.1: Arquitectura de una SDN.

- Capa de aplicación: esta capa contiene principalmente aplicaciones orientadas a servicios de red que se comunican con la capa de control. Ejemplos de aplicaciones de red, pueden ser sistemas de detección de intrusos, balanceadores de carga, cortafuegos, entre otros. Entre la capa de aplicación y la capa de control se utiliza la API *northbound* para establecer una interfaz de comunicación con el controlador de la SDN.
- Capa de plano de control: esta capa consiste en un controlador central que actúa como el cerebro de la SDN, sus funciones principales son mantener una vista global y dinámica de la red, tomar las peticiones de la capa de aplicación y administrar los dispositivos de red (estos dispositivos son, para una SDN, virtuales o reales).
- Capa de plano de datos: este plano consiste en los dispositivos de red (estos son los switches y routers virtuales o físicos) de una SDN. Entre la capa de control y la capa de plano de datos se usa la API *southbound* que establece la interfaz de comunicación entre ambas capas [5]. Es en esta interfaz donde el controlador mediante programación establece las reglas de reenvío [6].

2.2. Ambientes paravirtualizados

La paravirtualización es un método de virtualización cuya principal característica es mejorar el rendimiento. Esto se logra al eliminar elementos emulados que requieren una gestión especial como instrucciones con privilegios al kernel [7]. En un ambiente paravirtualizado existe una entidad llamada *hypervisor*, también conocida como *virtual machine monitor* (VMM), la cual es la entidad esencial de software que crea la separación entre máquinas virtuales [8].

2.2.1. Xen project hypervisor

Xen project hypervisor es un monitor de máquinas virtuales que soporta múltiples *guest-OS* con un alto nivel de aislamiento y rendimiento. Una de las principales características de Xen es la paravirtualización de las máquinas virtuales, lo que permite su optimización. Xen es un software *open-source* que es usado como la base de muchas aplicaciones en la nube. Xen puede ser instalado con Linux como el stack de control principal (*host*) o también llamado dominio 0 (*Dom0*), mientras que los dominios secundarios (máquinas virtuales) pueden ejecutar distintos sistemas operativos. El *Dom0* ejecuta la pila de herramientas de administración

de Xen y tiene privilegios especiales, como poder acceder al hardware directamente. El *Dom0* contiene los controladores para el hardware y proporciona acceso a la red y a los discos de los sistemas invitados (máquinas virtuales). Xen denomina a estas máquinas virtuales como dominios no privilegiados (*DomU*). Los *DomU* mantienen un driver paravirtualizado *(netfront)*, con el cual pueden interactuar con los componentes del dominio 0 [8].

2.2.2. Arquitectura de Xen

Xen hace uso de los siguientes elementos para lograr su operación:

- Xen *hypervisor*: es la capa básica, la cual es responsable de gestionar el CPU, particionar la memoria de múltiples máquinas virtuales y de controlar la ejecución de estas.

- Dominio 0: es una modificación del kernel de linux que es usado como el sistema operativo base (*host*). Xen necesita que el dominio 0 se ejecute en el arranque de la máquina física para poder encender las demás máquinas virtuales y es la única máquina virtual con acceso a los recursos físicos I/O *(Input-Output)*. El dominio 0 cuenta con dos drivers: *network backend*, cuya función es la de establecer una comunicación con el hardware local de red para el procesamiento de las peticiones hechas en las máquinas virtuales y *block backend*, el cual se comunica con el almacenamiento local del disco para leer y escribir datos basado en las peticiones del dominio U.

- Dominio de gestión y control *(Xen DM&C)*: estos elementos se ejecutan en el dominio 0 y consisten en la gestión y control de un ambiente virtualizado. Estos son los elementos que conforman el dominio de gestión y control:

  - Xend: es un demonio *(se le conoce como demonio a un programa que se ejecuta como proceso en segundo plano esperando un evento)* programado en Python que se encarga de gestionar el ambiente de Xen. *Xend* hace uso de *libxenctrl* para hacer peticiones a Xen *hypervisor*.

  - Xm: es una herramienta que toma las entradas de usuario mediante línea de comandos y las envía a Xen vía XML.

  - Xendstored: es un demonio que contiene un registro de información de memoria y enlaces de canales de eventos entre el dominio 0 y el dominio U.

  - Libxenctrl: es una biblioteca de C que permite la comunicación entre *Xend* y Xen *hypervisor* mediante el dominio 0.

  - Qemu-dm: esta herramienta maneja las peticiones del dominio U de disco y red para permitir un ambiente virtualizado.

- Dominio U
  Se le llama dominio U a todas las máquinas virtuales y las máquinas paravirtualizadas que son ejecutadas por Xen *hypervisor*. También se les conoce como dominios U PV y cada una puede ejecutar su propio sistema operativo. Los dominios U contienen dos drivers para los accesos a la red y al disco; *PV network* y *PV block* [9], en la figura 2.2 se ilustra la composición de Xen.

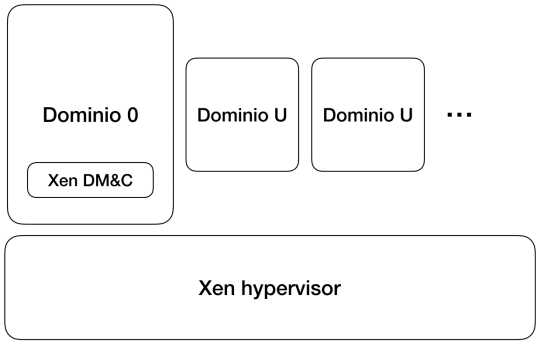

Figura 2.2: Elementos que componen a Xen.

2.3. Conmutadores virtuales

Los conmutadores virtuales *(Virtual Switch)* forman parte fundamental de un ambiente virtualizado ya que, de forma análoga a los *switches* físicos, son los responsables de que las máquinas virtuales puedan ser interconectadas entre ellas y a la red física. Algunas de las ventajas que traen consigo los conmutadores virtuales son su flexibilidad, el bajo costo y la eliminación de costosas licencias de marcas propietarias.

De forma general, los conmutadores virtuales conectan las tarjetas de red virtuales (*vNIC*) con las interfaces de red física (*pNIC*). La arquitectura de un conmutador virtual permite el tráfico entre *vNICs* a *vNIC* y de *vNIC* a *pNIC*. En la figura 2.3 se ilustra la arquitectura de un conmutador virtual.

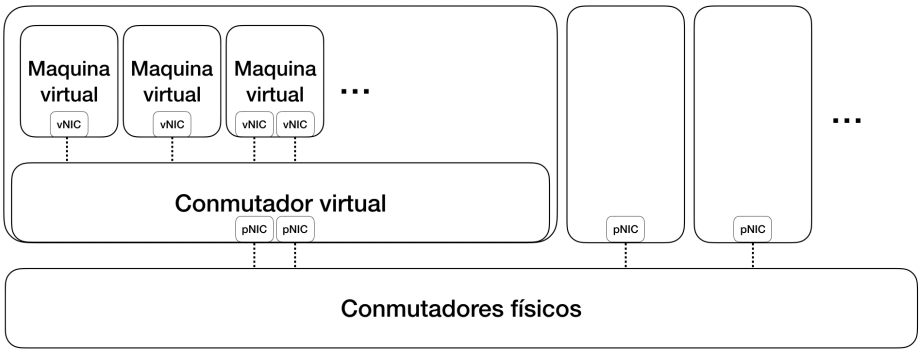

Figura 2.3: Arquitectura general de un conmutador virtual.

2.3.1. Open vSwitch

Open vSwitch (OVS) es un conmutador virtual multicapa *open-source* diseñado para permitir una automatización de la red mediante su programación, soporta distintos estándares como OpenFlow, SNMP e IPFIX [10].

OVS cuenta con dos rutas de procesamiento, en ambas rutas los componentes principales son *ovs-vswitchd* que controla el conmutador e implementa el protocolo OpenFlow y *datapath* que es un módulo del kernel que implementa el reenvío de paquetes [11]. Los paquetes que son procesados por *datapath* pueden tomar rutas rápidas (*fast path*), de modo que son procesados directamente por el módulo del kernel. Esto es gracias a que el procesamiento en *datapath* se hace mediante reglas del sistema que mantienen una tabla de flujo en memoria, la cual asocia el flujo con acciones. Por otro lado, los paquetes que no son procesados por *datapath* deben tomar rutas lentas (*slow path*), el cual es implementado por *vswitchd* mediante las reglas de OpenFlow, que copia el paquete en el espacio de usuario como se ilustra en la figura 2.4 [10].

A continuación se mencionan los principales componentes de *Open vSwitch*:

- *OVS-vswitchd*: es el principal programa de espacio de usuario en *Open vSwitch*. Permite leer la configuración desde la base de datos *ovsdb-server* y transfiere la información a la biblioteca *ofproto*.
- Ofproto: es la biblioteca de *Open vSwitch* que implementa OpenFlow. Permite la comunicación de los controladores de OpenFlow por medio de la red.
- Netdev: es una biblioteca de *Open vSwitch* que interactúa con los dispositivos de red como las interfaces ethernet. Es capaz de abrir puertos mediante el uso del demonio *netved provider*.
- Netdev-provider: trabaja con el hardware y software de los conmutadores e implementa un sistema operativo y una interfaz específica de hardware a los componentes de red como eth0.
- Dpif: explota las reglas *wildcard* en las entradas de emparejamiento, esto permite búsquedas de reglas más rápidas mediante funciones *hash* [12].

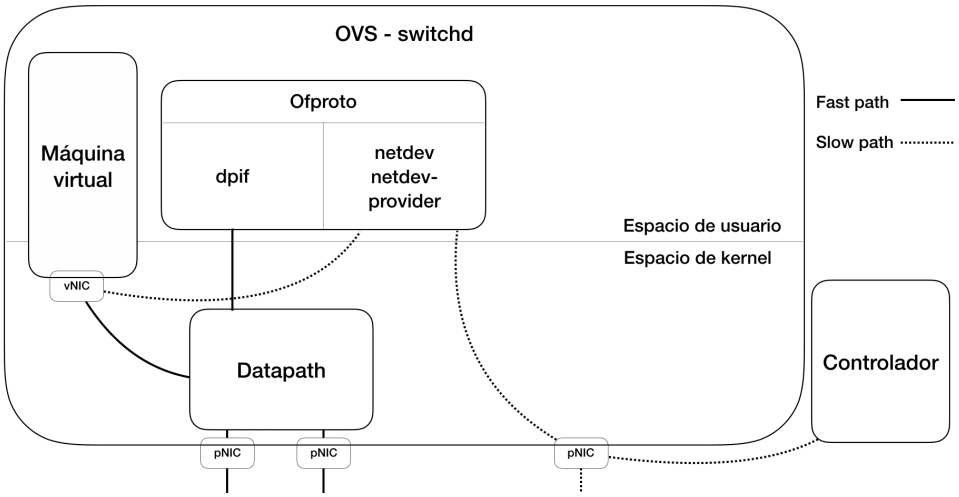

Figura 2.4: Arquitectura de OvS.

2.4. System calls

El *kernel* de Linux tiene como tarea principal proveer una serie de servicios a los programas y al mismo tiempo mantener una integridad del sistema. La forma en la que los programas solicitan recursos al kernel es por medio de *syscalls*. Es decir, *system calls (syscalls)* son el mecanismo que permite el acceso a los recursos físicos y es el medio que permite la comunicación entre capas desde el modo de usuario al espacio del *kernel*, es por esta razón que las *syscalls* son un mecanismo conveniente para un atacante, ya que puede invocar llamadas maliciosas al sistema operativo (OS) del *host* [13].

Las *syscalls* son instrucciones a bajo nivel optimizadas que ofrecen integridad al *kernel*, y deben cumplir con las siguientes restricciones:

- Retornar un número entero.
- El argumento del usuario al *kernel* es de la misma longitud.
- Los datos enviados al *kernel* se hacen por referencia, esto se refiere a que se deberán usar apuntadores a las estructuras, de este modo se logra optimizarlas.

Para utilizar las *syscalls* se debe importar la biblioteca de C <*unistd.h*>. Es importante mencionar de manera general algunos de los servicios que ofrecen las *syscalls*:

- Gestión y programación de procesos.
- Gestión de memoria.
- Servicios de red.
- Completar la interfaz de manejo de las señales.
- Servicios del sistema de archivos [14].

2.5. Hypercalls

Las *hypercalls* tienen la capacidad de ejecutar instrucciones de bajo nivel, es por ello que son la forma en la que los dominios (*DomU*) solicitan recursos. Algunas de las instrucciones que pueden realizar las *hypercalls* son:

- Manejo de excepciones.
- Programación (*scheduling*).
- Gestión de memoria.
- Control de acceso de discos y dispositivos de red.
- Operaciones de CPU.
- Comunicación entre dominios.

Al igual que las *syscalls*, las *hypercalls* son el mecanismo con el que se establece una interacción de capas entre Xen y las máquinas virtuales. Los dominios usan las *hypercalls* para solicitar operaciones de alto privilegio como peticiones de memoria de bajo nivel.

Ya que las *hypercalls* son usadas para acceder a los recursos del hardware y ejecutar instrucciones, los atacantes buscan alterar las *hypercalls* para acceder a espacios con privilegios [15].

2.6. Ataques basados en *hypercalls*

Las *hypercalls* comprenden las llamadas al sistema hechas por los *guests* (máquinas virtuales) en sesión activa, las cuales pasan hacia el *hypervisor* del sistema, el cual las autoriza o niega para que las primeras puedan tener acceso y hacer uso de los *drivers* y la memoria de la máquina física. Dentro de este mismo ámbito, estas llamadas no son filtradas, por lo que es muy difícil reconocer su legitimidad de una manera simple para el sistema. Por ello, un ataque de negación de servicio (DoS) basado en el uso de estas llamadas se complica detectarlo, ya que existe poca viabilidad de reconocer una secuencia de llamadas legítimas de una secuencia maliciosa o malintencionada.

Como se detalla en [2], los medios por los cuales estos ataques buscan cumplir su objetivo son los siguientes:

- Corrupción de imagen del *host*: a través de la sobreescritura de espacios de memoria del *host* o el desbordamiento del *buffer* de memoria disponible, se busca inutilizar el sistema operativo para evitar que éste pueda seguir proporcionando servicio al resto de los *guests* no infectados.
- Errores de tiempo de ejecución: a través de la búsqueda de excepciones no pre-programadas en el *kernel*, se busca caer en una condición que haga imposible que éste ultimo pueda seguir trabajando.

- Agotamiento de memoria: el propósito de este ataque consiste en disminuir la memoria disponible en el *host*, de tal suerte que no tenga suficiente memoria para atender sus propios procesos y como consecuencia disminuir la estabilidad de los sistemas invitados (máquinas virtuales).

Aunado a todo lo anterior, como puede constatarse en [16], el intervalo entre el descubrimiento de las vulnerabilidades en el *hypervisor* y la creación de una solución puede ser amplia, lo que permite que en ese espacio temporal surja una mayor cantidad de ataques. Por ello, se han propuestos mecanismos de seguridad para evitar este tipo de ataques, conocidos como Sistemas de Prevención y Detección de Intrusos (IDPS). Algunos ejemplos de estos abarcan métodos como *Collabra*, en el cual se etiquetan las invocaciones de las *hypercalls* como maliciosas o benignas partiendo del uso de políticas para determinar su legitimidad, muy similar a la forma en la que lo hace el sistema *XSM-FLASK*; o el sistema *RandHyp*, los cuales verifican aleatoriamente los valores de los parámetros de ciertas *hypecalls* para bloquear la ejecución desde locaciones que pudieran resultar de poca fiabilidad. Sin embargo, los autores en [16] afirman que estos sistemas no logran ser efectivos contra ataques que utilicen secuencias muy comunes de *hypecalls* que no sean sujetas a control por tratarse de sitios comunes de invocación o de rutina. Ante dichas circunstancias, existen trabajos que proponen IDPS más avanzados, con sistemas inteligentes de detección y un criterio más amplio de reconocimiento de comportamientos anómalos en el *kernel*. Por ejemplo, el sistema propuesto en [16] presenta una herramienta que los autores nombran como *hInjector*; la cual, a grandes rasgos, contiene una serie de ataques realistas, para evaluar al IDPS y es entrenado para que reconozca ataques con ciertos patrones de comportamiento, así como para evaluar el *overhead* incurrido por la utilización de estos sistemas en un ambiente real. Con un enfoque distinto, en [17] se usa un algoritmo diseñado con base en las reglas de frecuencia de la invocación de las llamadas al sistema cuya operación estuvo en observación experimental a priori durante un intervalo de tiempo prolongado. A partir de esto, puede hacerse un análisis de tráfico con reglas que sean capaces de reconocer los patrones anómalos de un *guest* posiblemente infectado.

2.7. Uso de las SDN

Hoy en día, es importante recalcar las grandes aportaciones que los sistemas virtualizados han proporcionado a múltiples servicios de red. Dos grandes ramas que se han visto beneficiadas por este tipo de arquitectura de red son las redes de *Data Centers* y servicios de tipo *Network as a Service (NaaS)*; este último, le permite a múltiples usuarios acceder a servicios de red y procesamiento desde un punto centralizado en la nube, lo que lo hace menos costoso que una infraestructura tradicional [18]. Esta arquitectura de red está orientada a servicios, conocidos como *Service Oriented Architecture (SOA)*, y basadas en un sistema de software integrado en múltiples unidades lógicas. El canon al que responde esta arquitectura se centra en el abastecimiento de recursos de procesamiento *on-demand* en un modelo de negocio llamado *pay-as-you-go*. La utilización conjunta de estos modelos de servicios permiten la obtención de interfaces de usuario de alto potencial que controlan las distintas capas de red y observan su proceder.

En cuanto a los *Data Centers*, las SDN les han provisto de una mayor escalabilidad comparadas con las redes tradicionales, debido a que estos sistemas virtualizados aprovechan todos los recursos de hardware, además que el uso de protocolos como OpenFlow proporcionan mejoras notables en el desempeño de procesamiento y de ancho de banda, reduciendo de forma importante el costo y tiempo de operaciones de las configuraciones de los conmutadores. Por otra parte, el creciente tráfico de datos da lugar a una mayor probabilidad de congestionamiento, algo afrontado frecuentemente a través de costosas operaciones de *multipathing*, los cuales muchas veces resultan en pérdidas dentro de los flujos; ante esta situación, la provisión de una solución que reduce considerablemente estas pérdidas representa una ventaja inconmensurable a favor de las SDN por sobre las redes tradicionales. Esto se logra a través de la implementación por software de métodos de reducción del sobrecoste en la actualización de los *path-loads*.

Otra ventaja que las SDN proporcionan a los *Data Centers* comprende la adaptabilidad a las diferentes aplicaciones en la nube, removiendo las limitaciones por el número de VLAN disponibles y actualizaciones *on-demand* de la red.

Por otro lado, OpenFlow ayuda a los *Data Centers* a aumentar el rendimiento de la red, ya que modifica los encabezados, reemplazando la información redundante con identificadores de flujo, con lo que mejora la latencia. Esta solución de la arquitectura OpenFlow se conoce como *Scissor*.

Capítulo 3

Implementación de Xen-OVS sobre Debian

3.1. Xen sobre Debian

Para el desarrollo experimental de este trabajo de tesis, se siguieron las recomendaciones mostradas en el sitio web del proyecto Xen [19], donde se detalla la guía de instalación del sistema operativo Debian en conjunto con Xen *hypervisor*. A continuación, se describen los pasos más importantes:

- Se instala el sistema operativo Debian 9 en una computadora portátil que sirva como máquina *host*.
- Se instala la última versión de Xen estable (4.9.0.12) a través del sistema de gestor de paquetes *apt* mediante la instrucción `apt-get install xen-linux-system-amd64, xen-tools, bridge-utils`.
- Se instala el gestor de volúmenes lógicos de Linux *(LVM)* para generar la unidad de memoria dentro de la cual serán alojadas las máquinas virtuales.
- Se configuran las herramientas de encaminamiento para que las máquinas virtuales puedan comunicarse unas con otras. También se crea un puente entre las máquinas virtuales y las interfaces de red de la máquina *host* para que tengan salida a Internet.
- Finalmente, se pueden crear las máquinas virtuales mediante los comandos propios del *hypervisor*. Aquí se puede indicar la cantidad de memoria del volumen lógico que alojará una máquina virtual, así como el sistema operativo que usará y cómo le será asignada la dirección IP. A continuación se presenta un ejemplo:

```
xen-create-image --hostname=tutorial-pv-guest \
--memory=512mb \
--vcpus=2 \
--lvm=vg0 \
--dhcp \
--pygrub \
--dist=buster
```

donde:

- `hostname`: es el nombre que llevará la máquina, queda a discreción del administrador de red, aunque siempre se recomienda asignar un nombre que la haga fácil de identificar, como su función o un número. En el caso de este trabajo de tesis, se optó por asignarles un número.

- `memory`: corresponde a la cantidad de memoria que se reserva a la imagen creada, para este ejemplo es de 512 MB. Dicha medida puede cambiar si el administrador así lo programa, para que dicho espacio sea asignado a otra VM o se le asigne un mayor espacio.
- `vcpus`: designa un número determinado de CPU a los que la máquina tendrá acceso para llevar a cabo sus tareas.
- `lvm`: determina el volumen lógico en el cual dicha máquina se aloja. `vg0` es el nombre del volumen lógico creado para almacenar particiones lógicas.
- `dhcp`: dictamina de qué manera recibirá la IP para ser reconocida como parte de una red, siendo en este caso por DHCP, aunque de igual forma puede asignarse de forma estática o manual.
- `pygrub`: pygrub es una herramienta proveniente desde las versiones más tempranas de Xen, con la cual el *bootloader* de cada máquina puede ser administrado desde su propio *kernel* en vez de tener que recurrir a uno localizado en el `Dom0`.
- `dist`: Xen permite que la distribución instalada en el *host* no necesariamente sea la que se instale a las máquinas huéspedes, por lo que cuenta con un catálogo de distintas distribuciones de Linux que se pueden instalar al crear la imagen de cada una de las máquinas. Para esta tesis, se optó por utilizar la distribución 9 y 10 de Debian, '`stretch`' y '`buster`', respectivamente. Como dato curioso, las distribuciones de Debian reciben sus nombres de los personajes de la afamada saga de películas de Disney-Pixar, Toy Story.

La figura 3.1 esquematizada el *hypervisor* y las dos máquinas virtuales creadas para esta tesis.

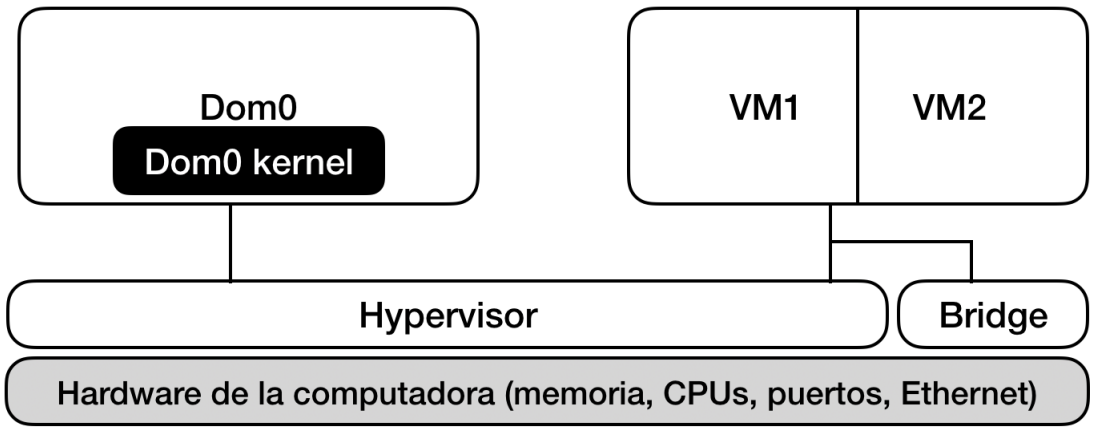

Figura 3.1: Arquitectura de la implementación de Xen sobre Debian 9.

Es importante resaltar que hasta el momento no se han definido flujos de tráfico determinados, por lo que las máquinas paravirtualizadas se pueden comunicar unas con otras a través del bridge.

3.2. Implementación de Open vSwitch

La compilación e instalación del proyecto Open vSwitch se puede seguir de [20]. A continuación se describen los pasos para la construcción de la arquitectura utilizada en esta tesis:

1. Se activa el sistema de gestión de OVS, en el cual se exportan los *scripts* descriptores del sistema para su inicialización desde la localidad en `/usr/local/share/openvswitch/`.

Para ello se exporta la siguiente variable de entorno y se guarda en el *bash_profile* del usuario:

```
export PATH=$PATH:/usr/local/share/openvswitch/scripts
```

2. El servicio *Open vSwitch* trabaja con dos comandos esenciales: `ovs-vsctl`, el cual se refiere al sistema de conmutadores en general; y `ovs-ofctl`, el cual se refiere a cada uno de los conmutadores generados y su manejo de flujos de información. Para esta tesis se propone la arquitectura mostrada en las figuras 3.2 y 3.3:

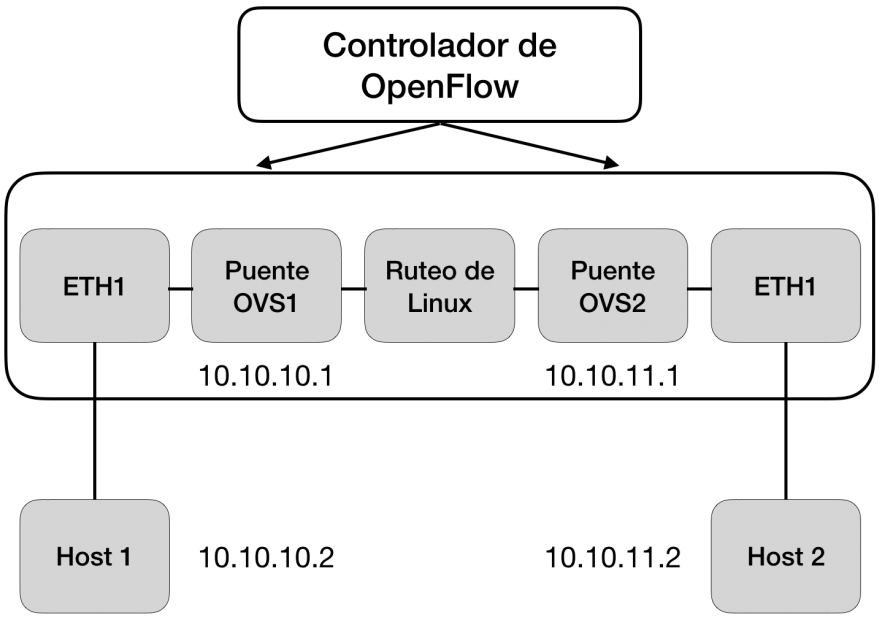

Figura 3.2: Arquitectura implementada de OVS-Xen sobre Debian.

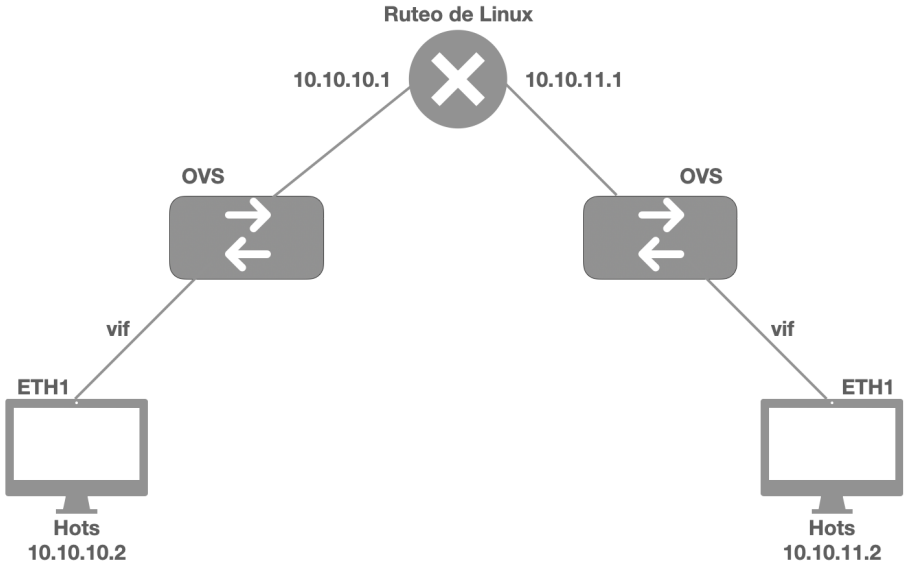

Figura 3.3: Diagrama de la red SDN con equipos reales.

Para la creación de los dos conmutadores mostrados en la figura 3.2, se hace uso de `ovs-vsctl`, de la siguiente forma:

```
ovs-vsctl add-br OVSbrx
```

donde `x` corresponde al número de puente que se desea crear.

3. Una vez creados los conmutadores se les puede asignar sus direcciones IP, para lo cual se hace uso de la herramienta de configuración de red en Linux `ifconfig`. De este modo, para ambos conmutadores se ejecuta:

```
ifconfig OVSbr1 10.10.10.1/24 up
ifconfig OVSbr2 10.10.11.1/24 up
```

Con el fin de asegurarse que los cambios fueron aplicados en las interfaces recién descritas, se ejecuta el comando `ip add` en la línea de comandos:

```
8: OVSbr1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue
    state UNKNOWN group default qlen 1000
link/ether ba:8d:41:35:53:4f brd ff:ff:ff:ff:ff:ff
inet 10.10.10.1/24 brd 10.10.10.255 scope global OVSbr1
   valid_lft forever preferred_lft forever
inet6 fe80::b88d:41ff:fe35:534f/64 scope link
   valid_lft forever preferred_lft forever
9: OVSbr2: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue
state UNKNOWN group default qlen 1000
link/ether be:a9:db:ce:45:45 brd ff:ff:ff:ff:ff:ff
inet 10.10.11.1/24 brd 10.10.11.255 scope global OVSbr2
   valid_lft forever preferred_lft forever
inet6 fe80::bca9:dbff:fece:4545/64 scope link
   valid_lft forever preferred_lft forever
```

4. Una vez asignadas las IP a cada conmutador, es necesario indicarle a las máquinas virtuales que su conexión sea a través de estos conmutadores como salida hacia el exterior. Para esto es necesario modificar los archivos de configuración de cada una de las máquinas virtuales (`/etc/xen/mach1.cfg` y `/etc/xen/mach2.cfg`) que se localizan en cada extremo de la red (ver figura 3.2), así como el archivo general de configuración de inicialización `/etc/xen/xl.conf`.
Relativo a los archivos de configuración de cada máquina, es necesario indicar en la sección de *Networking* la conexión a su puente. Por default, dicha sección está descrita de la siguiente manera:

```
#Networking
dhcp        ='dhcp'
vif         =['mac=xx:xx:xx:xx:xx:xx']
```

donde las `xx` representan cada par de dígitos correspondientes a la clave hexadecimal de los componentes de la dirección MAC de la máquina virtual. A lo que se debe añadir:

```
#Networking
dhcp        ='dhcp'
vif         =['mac=xx:xx:xx:xx:xx:xx,bridge=OVSbry']
```

donde `y` representa el número de conmutador al cual la máquina está conectada.
Por otro lado, la modificación del archivo de configuración general para todas las máquinas `xl.conf` se lleva a cabo en la sección de script de interfaz virtual para los *guests*, de la siguiente manera:

```
#default vif script to use if none is specified in the guest config
vif.default.script='vif-openvswitch'
```

Con estos cambios en los archivos de configuración, las máquinas podrán inicializarse y conectarse a sus respectivos conmutadores.

5. Descrita la inicialización de las máquinas virtuales en conjunto con OVS, el siguiente paso es iniciar las máquinas virtuales con el comando `xl create -c` para cada máquina. Si se revisa una vez más la configuración de red, se notará que se han añadido el número de máquinas virtuales inicializadas al conjunto de dispositivos de red, siendo en este caso interfaces virtuales o `vif`. Mediante el comando `ip add`, se obtiene una lectura como la siguiente:

```
10: vif5.0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq master
ovs-system state UP group default qlen 32
link/ether fe:ff:ff:ff:ff:ff brd ff:ff:ff:ff:ff:ff
inet6 fe80::334a:6b59:16a1:fb39/64 scope link
valid_lft forever preferred_lft forever
```

Tanto en los sistemas de conmutado virtual, como en los físicos, es necesario implementar reglas de flujo; dicha configuración se lleva a cabo dentro de cada conmutador, haciendo uso de la herramienta `ovs-ofctl`, por ejemplo:

- Se puede verificar el número de interfaces virtuales activas de acorde con el número de máquinas huésped inicializadas haciendo uso de `ovs-ofctl show OVSbrx`, donde `x` representa el identificador correspondiente a cada conmutador, para el caso de esta tesis 1 y 2. La salida de dicho comando del conmutador OVSbr1, se muestra a continuación:

```
OFPT_FEATURES_REPLY (xid=0x2): dpid:0000ba8d4135534f
n_tables:254, n_buffers:0
capabilities: FLOW_STATS TABLE_STATS PORT_STATS QUEUE_STATS
ARP_MATCH_IP
actions: output enqueue set_vlan_vid set_vlan_pcp strip_vlan
mod_dl_src mod_dl_dst mod_nw_src mod_nw_dst mod_nw_tos
mod_tp_src mod_tp_dst
1(vif5.0): addr:fe:ff:ff:ff:ff:ff
    config:     0
    state:      0
    speed: 0 Mbps now, 0 Mbps max
LOCAL(OVSbr1): addr:ba:8d:41:35:53:4f
    config:     0
    state:      0
    speed: 0 Mbps now, 0 Mbps max
OFPT_GET_CONFIG_REPLY (xid=0x4): frags=normal miss_send_len=0
```

  donde `vif5.0` corresponde a la interfaz virtual conectada con la máquina virtual de acuerdo a su archivo de configuración de inicialización.
- Con las máquinas inicializadas y correctamente conectadas a su conmutador, se puede definir los flujos de tráfico que serán permitidos en dicho segmento. Esto es

de vital importancia, ya que por default no viene ninguna regla definida; por lo que sin flujos, la máquina huésped no podrá transferir datos a su propio conmutador, ni al exterior de su segmento.

Mediante el comando `ovs-ofctl dump-flows OVSbrx`, se puede verificar las reglas de flujo definidas, hasta este momento, se obtendrá una salida nula, ya que aún no se establece ninguna regla que regule el tráfico en el conmutador. Para definir estas reglas, es necesario ejecutar un par de instrucciones por máquina:

```
ovs-ofctl add-flow OVSbrx in_port=y,actions=LOCAL
ovs-ofctl add-flow OVSbrx in_port=LOCAL,actions=output:y
```

donde una vez más, `x` corresponde al identificador del conmutador donde se fijará la regla `y`, en este caso, `y` es el número de puerto en el cual se ubicó la interfaz virtual de la(s) máquina(s) inicializada(s) dentro del subconjunto de red propio del conmutador. Estas reglas permiten el flujo bidireccional entre el puerto definido y el conmutador virtual. Hecho esto, si se vuelve ejecutar el comando `ovs-ofctl dump-flows OVSbrx`, la salida debería ser diferente de nula y en este caso deberá mostrar en dónde se insertó el número de puerto de la interfaz virtual, y el nombre de la interfaz. A continuación se muestra la respuesta para el conmutador 1, cuya interfaz virtual se encuentra en el puerto 1:

```
root@tesis:~# ovs-ofctl show OVSbr1
OFPT_FEATURES_REPLY (xid=0x2): dpid:0000ba8d4135534f
n_tables:254, n_buffers:0
capabilities: FLOW_STATS TABLE_STATS PORT_STATS QUEUE_STATS
ARP_MATCH_IP
actions: output enqueue set_vlan_vid set_vlan_pcp strip_vlan
mod_dl_src mod_dl_dst mod_nw_src mod_nw_dst mod_nw_tos
mod_tp_src mod_tp_dst
 1(vif6.0): addr:fe:ff:ff:ff:ff:ff
     config:     0
     state:      0
     speed: 0 Mbps now, 0 Mbps max
 LOCAL(OVSbr1): addr:ba:8d:41:35:53:4f
     config:     0
     state:      0
     speed: 0 Mbps now, 0 Mbps max
OFPT_GET_CONFIG_REPLY (xid=0x4): frags=normal miss_send_len=0
root@tesis:~# ovs-ofctl dump-flows OVSbr1
 cookie=0x0, duration=100.699s, table=0, n_packets=68,
 n_bytes=7124, priority=0 actions=NORMAL
root@tesis:~# ovs-ofctl add-flow OVSbr1 in_port=1,actions=LOCAL
root@tesis:~# ovs-ofctl add-flow OVSbr1 in_port=LOCAL,actions=output:1
root@tesis:~# ovs-ofctl dump-flows OVSbr1
 cookie=0x0, duration=23.020s, table=0, n_packets=10,
 n_bytes=810, in_port="vif6.0" actions=LOCAL
 cookie=0x0, duration=6.770s, table=0, n_packets=0,
 n_bytes=0, in_port=LOCAL actions=output:"vif6.0"
 cookie=0x0, duration=160.672s, table=0, n_packets=86,
 n_bytes=8482, priority=0 actions=NORMAL
```

- Como requerimiento adicional, es necesario habilitar la función de *IP forwarding*, la cual permitirá el encaminamiento de paquetes al pasar de las máquinas al conmutador y del conmutador al controlador de Linux. Para ello, se ejecuta:
  `echo 1 > /proc/sys/net/ipv4/ip_forward`

6. Como último paso, en cada una de las máquinas se debe editar el archivo de configuración de red, a modo de que se fije la dirección IPV4:

```
auto eth0
iface eth0 inet static
    address 10.10.x.2/24
    gateway 10.10.x.1
```

   donde `x` representa el segmento de red del conmutador al que pertenece la máquina. Sólo es necesario reiniciar los servicios de red en la máquina con el comando `service networking restart`, el cual permite que los cambios hechos tomen efecto.

Estos pasos se deben realizar para todas las máquinas virtuales. De esta manera la arquitectura planteada para esta tesis está completa. En los capítulos siguientes se presenta el manejo de llamadas al kernel, así como el diseñó del ataque DoS.

Capítulo 4

Syscalls

En este capítulo se analizan las *syscalls* obtenidas mediante distintas tareas y protocolos ejecutados entre máquinas virtuales. La estructura, así como la descripción de las *system calls* que a continuación se presentan, se obtuvieron del *Linux manual page* [21]. Durante la ejecución de las tareas entre las máquinas virtuales, múltiples *system calls* pueden ocurrir, y cada una requiere de la capacidad de intercambiar de un modo no privilegiado *(modo de usuario)* a un modo privilegiado *(modo kernel)* [22].

En este punto es necesario diferenciar entre una API *(application programmer interface)* y una *system call*, ya que una API es una definición de función que especifica cómo obtener un servicio determinado, mientras que una *system call* es una petición explícita al *kernel* mediante interrupciones de software. Algunas API son definidas por bibliotecas que forman parte de los sistemas Unix como lo es *libc*, las cuales se refieren a rutinas *wrapper* que tienen como único propósito generar una *system call* [23]. Una API no corresponde a una llamada al sistema necesariamente ya que estas pueden ofrecer los servicios directamente desde el modo de usuario. Un ejemplo de esto es la implementación de las API `malloc()`, `calloc()` y `free()` que se encuentra en la biblioteca *libc*, es en esta misma biblioteca donde se usa la *system call* `brk()` para ampliar o reducir la pila de procesos.

Antes de analizar las distintas *system calls* se debe considerar que muchas de ellas tienen un valor de retorno. Algunas rutinas *wrapper* tienen como valor de retorno un número entero, el cual depende de la respuesta de la *system call*. Cuando el valor de retorno es igual a -1 comúnmente implica que el *kernel* se encuentra inhabilitado para cumplir con la petición, esta situación puede ser ocasionada por parámetros inválidos o falta de recursos disponibles. Por otro lado, un valor de retorno positivo o 0 implica una ejecución exitosa de la *system call*.

De forma general, la primera *system call* que se obtiene al realizar una tarea es `execve`. Esta *system call* ejecuta el programa referido al parámetro `pathname` como se muestra en la figura 4.1, este parámetro deberá ser un binario ejecutable o un script que comience con la línea `#!interpreter [ ]`. El programa que se está ejecutando por el proceso de llamada es reemplazado por un nuevo programa con nuevos segmentos de datos inicializados.

execve

#include <unistd.h>

int execve(const char *_pathname_, char *const _argv[ ]_,
char *const _envp[ ]_);

Figura 4.1: Parámetros de excve.

El parámetro `argv` es un arreglo de apuntadores a strings, donde el primer elemento de este arreglo (`argv[0]`) contiene el nombre asociado con el archivo que es ejecutado. El siguiente

parámetro, `envp`, es un arreglo de apuntadores a strings que funciona como el entorno del nuevo programa.

Así como `execve`, `break` es otra *system call* que tiene como tarea cambiar la ubicación de la ruptura del programa, la cual define el final del segmento de datos del proceso. Esta llamada establece el final del segmento de datos al valor especificado por `addr` siempre que el valor sea razonable o el sistema tenga la memoria suficiente. El valor de retorno de *brk* es `0` cuando la operación es exitosa y `-1` cuando ocurre un error (ver figura 4.2).

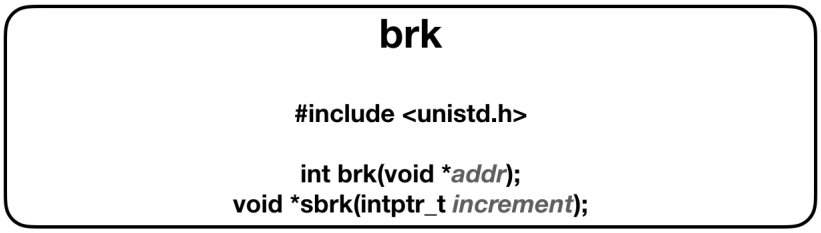

Figura 4.2: Parámetros de brk.

`Access` tiene como tarea verificar si el proceso puede acceder al archivo referido al parámetro `pathname` (ver figura 4.3). El parámetro `mode` especifica la comprobación de accesibilidad. Los valores que `mode` puede tomar son; `F_OK` que verifica la existencia de un archivo, `R_OK`, `W_OK` y `X_OK`, estos tres últimos valores de `mode` verifican la existencia del archivo y otorgan permiso de lectura, escritura y ejecución respectivamente.

`Access` es de mucha importancia porque el proceso de comprobación se realiza usando *UID* y *GID* del proceso. De forma similar al *root user*, la comprobación usa el conjunto de capacidades permitidas en lugar del conjunto de capacidades efectivas. De esta forma los programas `set-user-ID` y `capability-endowed` pueden determinar la autoridad del usuario. Dicho de otra forma, lo que permite es responder a la pregunta *¿El usuario que me invoco puede leer/escribir/ejecutar este archivo?* Esto da la posibilidad a los programas `verb-user-ID` de prevenir que usuarios maliciosos puedan leer archivos que no deberían.

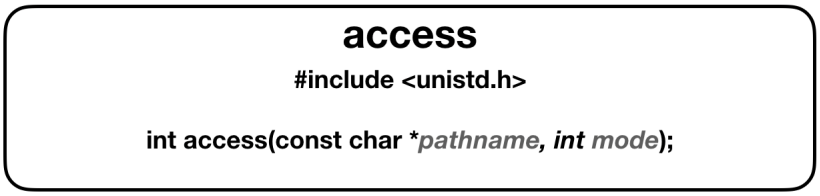

Figura 4.3: Parámetros de access.

`Openat` es una *system call* que permite abrir el archivo especificado por `pathname`, si el archivo no existe se puede crear si se especifica con `O_CREAT` en el parámetro `flags`. El valor de retorno es un descriptor del nuevo archivo que debe tomar un valor entero positivo y -1 si ocurre un error. En caso de que `dirfd` tenga el valor especial `AT_FDCWD`, entonces `pathname` se interpreta como relativo al actual directorio del proceso (ver figura 4.4).

La bandera `O_CLOEXEC` será importante para el análisis del comportamiento de las *system calls*. Esta bandera permite al programa evitar operaciones adicionales. Su uso es importante para programas "multihilo", ya que algunas de las operaciones que evita no pueden soportar algunas condiciones donde un hilo abre un descriptor de archivo y trata de establecer su bandera *close-on-exec* y al mismo tiempo otro hilo corre un `fork` y un `execve`.

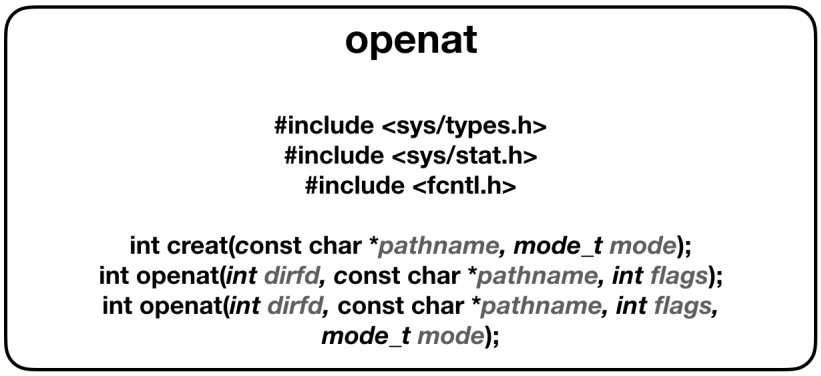

Figura 4.4: Parámetros de openat.

`Fstat` forma parte de un grupo de *system calls* (*stat, lstat y fstatat*) que obtienen información de un archivo (especificado por el parámetro `fd`) en `statbuf`, para el caso de *fstat* es necesario el permiso para ejecutar el archivo (ver figura 4.5).

Como primer elemento de `statbuf` encontramos el tipo de archivo, a continuación se muestra como la estructura switch-case determina el tipo de archivo mediante el argumento `sb.st_mode & S_IFMT`:

```
S_IFBLK  block device
S_IFCHR  character device
S_IFDIR  directory
S_IFIFO  FIFO
S_IFLNK  symlink
S_IFREG  regular file
S_IFSOCK socket
```

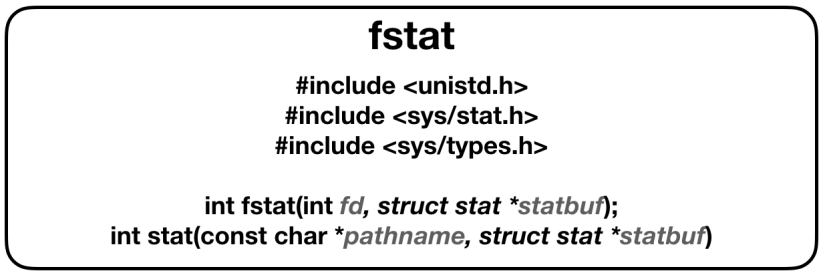

Figura 4.5: Parámetros de fstat.

`Arch_prctl` establece el proceso específico de la arquitectura o el estado del hilo. El parámetro `code` selecciona una subfunción, mientras que `addr` es interpretado como *unsigned long* para las operaciones *"set"*, o como *unsigned long ** para las operaciones *"get"* (ver figura 4.6).

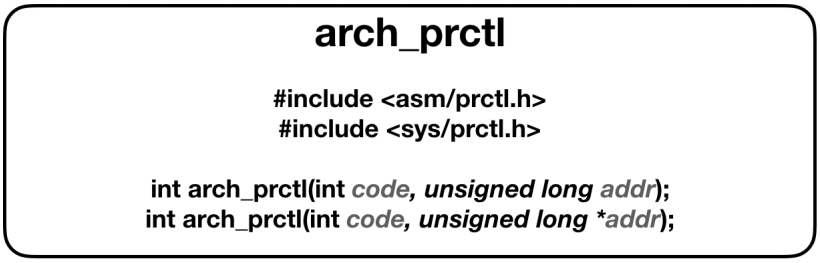

Figura 4.6: Parámetros de arch prctl.

`Mmap` crea un nuevo mapeo en el espacio de las direcciones virtuales del proceso. El parámetro `addr` especifica el principio de la dirección para el nuevo mapeo, mientras que `length` especifica la longitud del mapeo. Existe la posibilidad de que el parámetro `addr` sea *nulo*, en esta situación el *kernel* elige el comienzo de la dirección. Cuando el valor de `addr`, si se especifica, el *kernel* lo toma como indicio sobre donde hacer el mapeo siempre considerando un valor mayor que se especifica en `/proc/sys/vm/mmap_min_addr`. En caso de que otro mapeo ya exista en esa dirección, el *kernel* se encargará de elegir una nueva dirección que no necesariamente dependerá del indicio y esta nueva dirección será el retorno de la *system call*. El mapeo es inicializado usando `length` bytes empezando en `offset` (este parámetro deberá ser múltiplo del tamaño de la página que se obtiene mediante `sysconf(_SC_PAGE_SIZE)`) en el archivo referido mediante el descriptor `fd`. El parámetro `prot` describe la protección de memoria requerida del mapeo, estas son las posibles banderas que `prot` puede adoptar:

```
PROT_EXEC     Páginas pueden ser ejecutadas
PROT_READ     Páginas pueden ser leídas
PROT_WRITE    Páginas pueden ser escritas
PROT_NONE     Páginas no pueden ser accedidas
```

El parámetro `flags` determina si las actualizaciones del mapeo son visibles para otros procesos de mapeo en la misma región, también si pueden ser llevadas a través de un archivo subyacente. Estas son los posibles comportamientos de `flags`:

```
MAP_SHARED:
          El mapeo puede ser compartido, las actualizaciones
          son visibles a los procesos de mapeo en la misma
          región y se pueden llevar a cabo por un
          archivo subyacente.

MAP_SHARED_VALIDATE:
          Esta bandera se encuentra disponible a partir de
          Linux 4.15. Cuando se asigna esta bandera se adopta
          el comportamiento de MAP_SHARED, con la diferencia
          que cuando se usa MAP_SHARED_VALIDATE, el kernel
          verifica que todas las banderas son conocidas y si
          alguna no lo es, el mapeo falla con el error
          EOPNOTSUPP.

MAP_PRIVATE:
          Con esta bandera se crea un mapeo privado
          copy-on-write (esta es una técnica de optimización en
          la que múltiples procesos pueden solicitar recursos
          iguales, y se les devuelven apuntadores al mismo
          recurso), las actualizaciones no son visibles y los
          archivos adyacentes no pueden llevar a cabo las
          actualizaciones.
```

```
MAP_ANONYMOUS:
        Esta bandera indica que el mapeo no es respaldado por ningún
        archivo, por lo que su contenido es inicializado en cero.
        El parámetro fd es ignorado.
```

En caso de éxito, `mmap` devuelve un apuntador al área de mapeo y en caso de error el valor de retorno es -1. Por otro lado, la *system call* `munmap` elimina el mapeo en la sección de memoria especificada por sus parámetros `addr` y `length`.

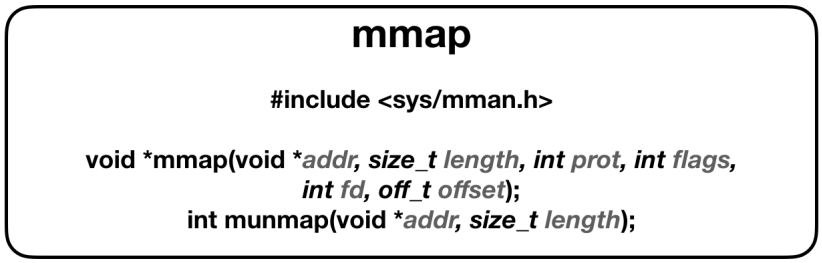

Figura 4.7: Parámetros de mmap.

En cuanto a los hilos, el *kernel* tiene dos atributos; `set_child_tid` y `clear_child_tid`, ambos atributos se encuentran en *NULL* por defecto.

```
set-child-tid: Es usado cuando un hilo es iniciado
               con la bandera CLONE-CHILD-SETTID.

clear-child-tid: Es usado cuando un hilo es iniciado con la
                 bandera CLONE-CHILD-CREARTID.
```

`Set_tid_address` es usado para establecer un apuntador al ID de un hilo. Por otro lado, cuando *clear-child-tid* no es *NULL*, entonces se escribe `0` en la dirección especificada por *clear-child-tid*. Con esto, se despierta un solo hilo que realiza un *futex* en la memoria.

Nota: futex (fast userspace mutex) es una system call que es usada para implementar un bloqueo básico).

La *system call* `set_tid_address` establece el valor `clear_child_tid` al hilo de llamada del parámetro `tidptr`. Como valor de retorno, esta *system call* siempre devuelve el ID del hilo de llamada (ver figura 4.8).

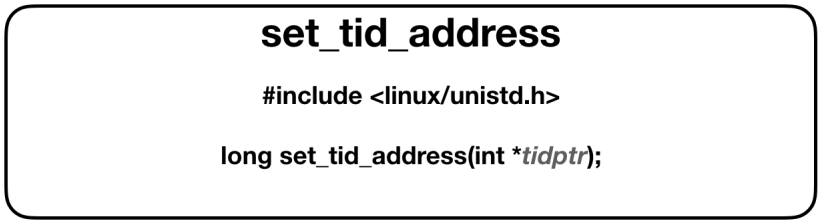

Figura 4.8: Parámetros de set-tid-address.

`Mprotect` es una *system call* que será de vital importancia para el análisis de este capítulo (ver figura 4.9), ya que esta se encarga de cambiar la protección de acceso a las páginas de memoria de un proceso, esta sección de memoria contiene rangos de memoria dentro del rango comprendido entre `(addr, addr+len-1)`. En caso de que el proceso intente acceder a la memoria de un modo "violento", el *kernel* genera una señal para el proceso `SIGSEGV`.

```
  PROT_NONE:
            La memoria no puede ser accedida en absoluto.
  PROT_READ:
            La memoria se puede leer.
  PROT_WRITE:
            La memoria puede ser modificada.
  PROT_EXEC:
            La memoria puede ser ejecutada.
  PROT_SEM:
            Esta bandera está disponible a partir de Linux 2.5.7, la
            memoria puede ser usada para instrucciones atómicas, estas
            instrucciones son implementadas para realizar múltiples
            procesos vistos por el sistema como uno solo, de modo que
            es indivisible.
  PROT_SAO:
            Esta bandera está disponible a partir de Linux 2.6.26,
            e indica que la memoria debe tener una orden de acceso.
```

La ventaja del uso de esta *system call* es que siempre es permitido llamarla en cualquier dirección en el espacio del proceso, por lo que es usada para cambiar el código de mapeo existente y así permitir la escritura.

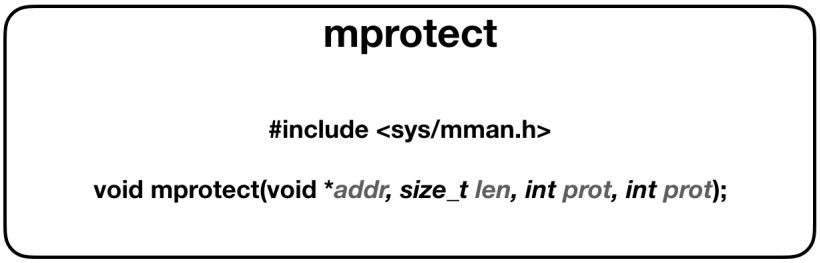

Figura 4.9: Parámetros de mprotect.

Capget y *capset* son la interfaz del *raw kernel*, estas *system calls* obtienen y asignan las capacidades del hilo. Algunos detalles importantes sobre las estructuras son:

```
  #define _LINUX_CAPABILITY_VERSION_1  0x19980330
  #define _LINUX_CAPABILITY_U32S_1     1

/* V2 agregado en la versión Linux 2.6.25*/
  #define _LINUX_CAPABILITY_VERSION_2 0x20071026
  #define _LINUX_CAPABILITY_U32S_2     2

/*V3 agregado en la versión  Linux 2.6.26*/
  #define _LINUX_CAPABILITY_VERSION_3  0x20080522
  #define _LINUX_CAPABILITY_U32S_3     2

        typedef struct __user_cap_data_struct {
         __u32 version;
             int pid;
            } *cap_user_header_t;

            typedef struct __user_cap_data_struct {
             __u32 effective;
             __u32 permitted;
```

```
    __u32 inheritable;
  } *cap_user_data_t;
```

Los campos `effective`, `permitted` e `inheritable` son máscaras de las capacidades definidas en *capabilities*, que es un overview de las capacidades de Linux. *Capabilities* tiene como propósito comprobar las funciones de permiso. En los sistemas Unix se implementan dos categorías de procesos: *privileged* (para usuarios cuyo ID es `0`, el cual representa al superusuario o *root*) y *unprivileged*. Los procesos privilegiados tienen la capacidad de evitar la revisión de permisos del *kernel*, mientras que los procesos no privilegiados deben comprobar los permisos basado en las credenciales del proceso (*effective ID, effective GID* y *supplementary group list*).

Capget (ver figura 4.11) puede obtener las capacidades de cualquier proceso mediante su ID con `hdrp->pid`.

capget/capset

#include <sys/capability.h>

int capget(cap_user_header_t *hdrp*, *cap_user_data_t datap*);
int capset(cap_user_header_t *hdrp*, *const cap_user_data_t datap*);

Figura 4.10: Parámetros de capget/capset.

La *system call* `uname` Devuelve información del sistema usando la estructura apuntada por `buf`. Esta estructura es llamada `utsname` y su forma es:

```
    struct utsname{
char systname[];  /*Nombre del sistema operativo*/
char nodename[];  /*Nombre en "implementation-defined network"*/
char release[];  /*Edición de systema operativo*/
char versión[];  /*Versión del sistema operativo*/
char machine[]; /*Identificador de hardware*/
};
```

uname

#include <sys/utsname.h>

int uname(struct utsname *buf);

Figura 4.11: Parámetros de uname.

La *system call* `socket` crea un endpoint para la comunicación y devuelve un archivo descriptor del endpoint creado. El parámetro `domain` especifica el dominio de comunicación, el cual selecciona el protocolo que será usado (las familias de protocolos de comunicación se definen en `<sys/socket.h>`) , ver figura 4.12.

```
Nombre    Objetivo
AF_UNIX    Comunicación local
```

```
AF_INET    Protocolo IPv4
AF_INET6   Protocolo IPv6
AF_PACKET  Interfaz de paquete de bajo nivel
```

El parámetro `type` especifica la semántica de la comunicación, mientras que el parámetro `protocol` asigna el protocolo que se usará con el socket. A continuación se mencionan las banderas que `type` puede adoptar:

```
SOCK_STREAM:
                Proporciona un flujo de bytes secuenciados
                bidireccionales basados en conexión.

SOCK_DGRAM:
                Soporta datagramas (mensajes sin conexión
                y poco fiables)

SOCK_SEQPACKET:
                Proporciona un camino de transmisión de datos
                para datagramas secuencial y bidireccional
                basado en conexión. Se necesita de un cliente
                para leer un paquete completo con cada entrada
                al sistema.

SOCK_RAW:
                Proporciona un protocolo "raw" de acceso a la
                red.
```

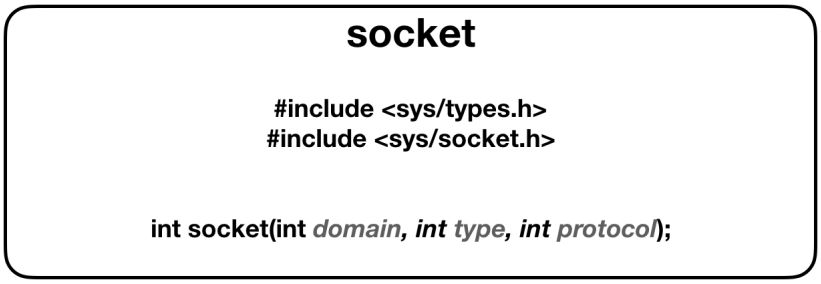

Figura 4.12: Parámetros de socket.

Cuando un socket es creado con la *system call* `socket`, este existe pero no tiene una dirección asignada. `Bind` (ver figura 4.13) asigna la dirección `addr` al socket `sockfd`, y por medio de `addrlen` se especifica el tamaño (en bytes) de la estructura de dirección apuntada por `addr`. Comúnmente se asigna una dirección local a un socket de tipo SOCKSTREAM antes de que este pueda recibir conexiones.

Nota: Esta acción es conocida como "assigning a name to a socket"

La estructura apuntada por `addr` es:

```
 struct sockaddr{
sa_family_t  sa_family;
char  sa_data[14];
 }
```

Nota: Esta estructura solo es usada para evitar advertencias del compilador

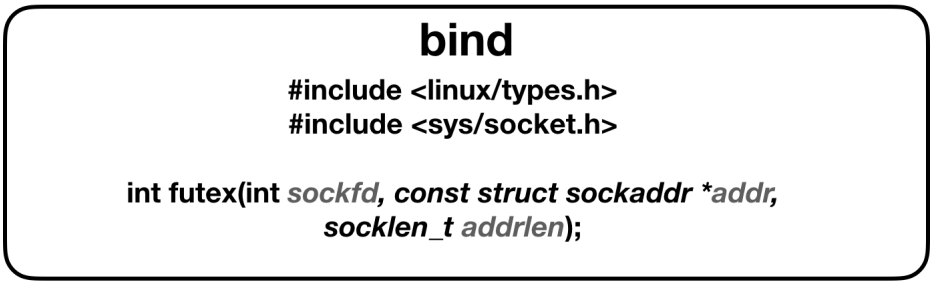

Figura 4.13: Parámetros de bind.

La *system call* `listen` "marca" el socket `sockfd` como pasivo, lo que quiere decir que ese socket aceptará solicitudes de conexión usando *accept*, ver figura 4.14.

El parámetro `backlog` define la longitud máxima del queue de conexiones pendientes del socket. Si una solicitud de conexión llega cuando el queue está lleno, el cliente recibirá un error con la bandera `ACONNREFUSED` o en caso de que el protocolo soporte retransmisión, la petición será ignorada y así el cliente intentará conectarse otra vez con éxito.

Cuando se trata de sockets TCP, el comportamiento del argumento `backlog` cambia desde Linux 2.2, ahora `backlog` especifica la longitud del queue para los sockets esperando a ser aceptados. La longitud máxima del queue se define en `/proc/sys/net/ipv4/tcp_max_syn_backlog`. Cuando `syncookies` es habilitado, no hay una longitud máxima y este ajuste es ignorado.

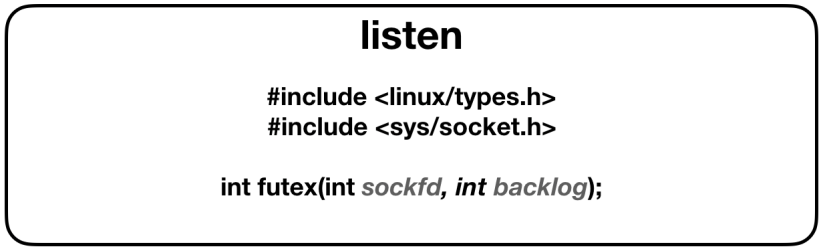

Figura 4.14: Parámetros de listen.

La *system call* `futex` (ver figura 4.15) cuenta con un método para esperar hasta que una condición específica se cumple. Es usada como bloqueo en el contexto de sincronización de memoria compartida, la mayoría de las operaciones de sincronización se hacen en el espacio de usuario. Otra forma de usar `futex` es para despertar un proceso o un hilo que esté esperando una condición particular. En procesos multihilo, `futex` puede ser usada globalmente con el fin de ser compartida por todos los hilos. Si la operación `futex` solicita el bloqueo de un hilo, el *kernel* lo hará solo si *futex word* tiene el valor que el hilo de llamada proporcionó como valor esperado de *futex word*.

El parámetro `uaddr` apunta al valor *futex word*, el parámetro `futex_op` especifica la operación de `futex` donde el significado de `val` depende de la operación `futex_op`. Los demás parámetros (`timeout, uaddr2` y `va13`) solo son requeridos para ciertos valores de `futex_op`. Algunos valores de `futex_op` son:

```
FUTEX_PRIVATE_FLAG:

   Esta opción puede ser anexada a todas las operaciones de futex. Su
   función es informar al kernel que el proceso de futex es privado
   y no será compartido con otro proceso. La forma de utilizar esta
   bandera es colocando PRIVATE después de cada operación.
   Por ejemplo FUTEX_WAIT_PRIVATE y FUTEX_WAKE_PRIVATE.

FUTEX_WAKE:
```

```
Esta operación despierta val de los waiters que estan
esperando en futex word en la dirección uaddr.
```

futex

```
#include <linux/futex.h>
#include <sys/time.h>

int futex(int *uaddr, int futex_op, int val, const struct
timespec *timeout, int *uaddr2, int va13);
```

Figura 4.15: Parámetros de futex.

`Connect` tiene la función de conectar el socket referido por el archivo descrito en el parámetro `sockfd` en la dirección `addr`. Cuando `sockfd` toma el valor de `SOCK_DGRAM` el parámetro `addr` es la dirección a la que se envían los datagramas por defecto. Si `sockfd` toma el valor de `SOCK_STREAM` o `SOCK_SEQPACKET`, entonces la llamada establece una conexión con el socket al que está vinculado la dirección `addr`, ver figura 4.16.

connect

```
#include <sys/types.h>
#include <sys/socket.h>

int connect(int sockfd, const struct sockaddr *addr,
socklen_t addrlen);
```

Figura 4.16: Parámetros de connect.

La *system call* `getsockname` devuelve la dirección a la que el socket `sockfd` está vinculado al buffer apuntado por `addr`, el parámetro `addrlen` debe ser inicializado para indicar el total de espacio en bytes señalado por `addr`, ver figura 4.17.

getsockname

```
#include <sys/socket.h>

int getsockname(int sockfd, struct sockaddr *addr,
socklen_t *addrlen);
```

Figura 4.17: Parámetros de getsockname.

`Getsockopt` y `setsockopt` tienen la capacidad de manipular las opciones del socket referido por el archivo descriptor `sockfd`. Para poder manipular el socket se debe especificar el nivel y nombre de la opción, ya que existen opciones en el nivel API (en este caso el parámetro `level` toma el valor de `SOL_SOCKET`), para manipular las opciones en los demás niveles, el número del protocolo de la opción es suministrado. El parámetro `optname` y las opciones especificadas se pasan al módulo de protocolo apropiado para su interpretación, en `<sys/socket.h>` se incluyen las definiciones para las opciones del nivel del socket. Los parámetros `optval` y `optlen` sirven para acceder a los valores de opción de `setsockopt`. En el caso de `getsockopt`, los parámetros anteriormente mencionados identifican el buffer al

que se devuelven las opciones solicitadas, de forma particular, `optlen` es el argumento "valor resultado" y que inicialmente contiene el tamaño del buffer apuntado por `optval`. Al cerrar un socket, su dirección (compuesta por la IP y el número de puerto) no podrá ser usada por un periodo de tiempo corto, para poder usarla se deberá usar `setsockopt`, ver figura 4.18.

Entre las opciones del socket, las siguientes son algunas que pueden ser asignadas (setsockopt) o leídas (getsockopt) con un nivel de socket en `SOL_SOCKET`.

```
    SO_ERROR:
Obtiene y limpia los errores pendientes del socket.
    PACKET_AUXDATA:
    Esta opción permite que el socket pase una estructura de metadatos en cada
    paquete en el campo de control de recvmsg. La estructura es:
            struct tpacket_auxdata{
            __u32   tp_status;
            __u32   tp_len; /*longitud del paquete*/
            __u32   tp_snaplen; /*longitud capturada*/
            __u16   tp_mac;
            __u16   tp_net;
            __16   tp_vlan_tci;
            __16   tp_vlan_tpid
            }
    PACKET_RX_RING:
        Crea un buffer de anillo de memoria mapeada para los paquetes de recepción
        asíncronos. El paquete del socket reserva una región continua de espacio
        de direcciones de aplicación, lo pone en un array de slots de paquetes y
        copia los paquetes en slots subsecuentes. Cada paquete es seguido de una
        estructura de metadatos.
```

getsockopt/setsockopt

```
#include <sys/types.h>
#include <sys/socket.h>

int getsockopt(int sockfd, int level, int optname, void
*optval, socklen_t *optlen);
int setsockopt(int sockfd, int level, int optname, void
const *optval, socklen_t *optlen);
```

Figura 4.18: Parámetros de getsockopt/setsockopt.

Las *system call* `set_robust_list` y `get_robust_list` trabajan con listas de futex robustas. Las listas son procesadas en el espacio de usuario, mientras que el *kernel* tiene conocimiento solo de la ubicación del *head* de la lista. La forma en la que el hilo informa al *kernel* sobre la localización de su lista de `futex` robusta es con `set_robust_list`, mientras que la forma de obtener la dirección de la lista es usando `get_robust_list`. El uso de las listas robustas de `futex` permite asegurar que en caso de que el hilo fallé accidentalmente al desbloquear un `futex` antes de que termine o se llame a la *system call* `execve`, un hilo que esté esperando el `futex` sea notificado que el propietario del `futex` ya no existe, ver figura 4.19.

La *system call* `set_robust_list` solicita al *kernel* que registre el *head* de la lista robusta de futex cuyo dueño es el hilo de llamada. El parámetro `head` es el *head* de la lista a registrar, y el parámetro `len` es el tamaño de `*head`.

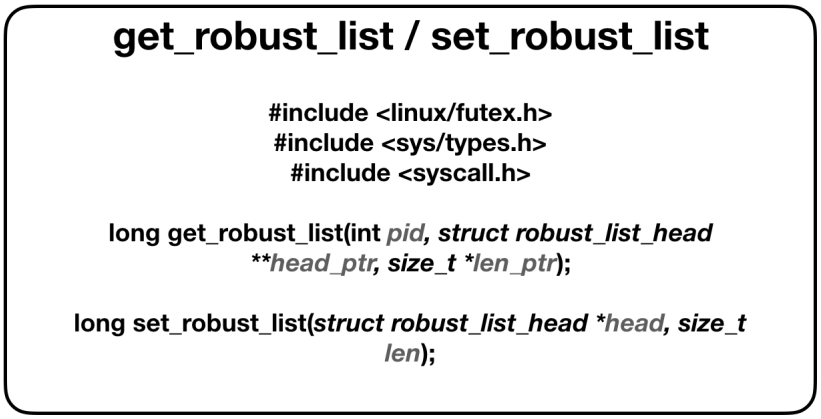

Figura 4.19: Parámetros de get-robust-list/set-robust-list.

La *system call* `sigaction` es usada para cambiar la acción tomada por un proceso al recibir una señal (ver figura 4.20). El parámetro `signum` especifica la señal. Algunas de las señales que puede adoptar `signum` son:

```
SIGALRM:
     Esta señal indica que el "timer" asociado a la función alarm finaliza
     o que el primer intervalo es configurado, el cual es setitimer.
SIGCHLD:
     Indica al "padre" que el proceso terminó. Generalmente el proceso padre
     usa la función wait para obtener información del "hijo" y así evitar
     procesos "zombies".
SIGINT:
     Esta señal se genera al presionar DELETE o Control-C para finalizar
     un proceso.
 SIGKILL:
     Termina un proceso.
 SIGTSTP:
     Se genera al presionar Control-Z.
 SIGSTOP:
     Esta señal cumple la función de SIGTSTP, pero a diferencia de la anterior,
     esta no puede ser ignorada.
 SIGCONT:
     Reanuda un proceso que fue suspendido por SIGSTOP.
 SIGQUIT:
     Sale desde el teclado.
 SIGRTMIN/SIGRTMAX:
     Desde la versión 2.2, linux es capaz de soportar señales en tiempo real,
     el rango de señales en tiempo real soportadas se define por SIGRTMIN y
     SIGRTMAX.
     El kernel de linux soporta un rango de 33 diferentes señales en tiempo
     real (desde los números 32 a 64).
     La implementación de hilos glibc POSIX utilizan por lo general dos (para
     NPTL) o tres (para LinuxThreads) señales en tiempo real y ajustan el valor
     de SIGRTMIN a 34 o 35.
```

Cuando el parámetro `act` es distinto de NULL, la acción previa se salva en el parámetro `oldact`. La estructura de *sigaction* es:

```
struct sigaction{
```

```
    void      (*sa_handler)(int);
    void      (*sa_sigaction)(int, siginfo_t *, void *);
    sigset_t  sa_mask;
    int       sa_flags;
    void      (*sa_restorer)(void);
};
```

donde `sa_handler` especifica la acción a la que estará asociado `signum`, generalmente es un apuntador a una función de manejo de señales. `sa_mask` especifica la máscara de señales, la cual debe ser bloqueada durante la ejecución del "operador" de señales. `sa_flags` especifica el conjunto de banderas que modifican el comportamiento de la señal [24], algunas de ellas son:

```
SA_INTERRUPT:
     Esta bandera indica un "operador" de interrupciones rápido.
     Las interrupciones "rápidas" se ejecutan con todas las demás
     interrupciones deshabilitadas en el procesador actual (los
     demás procesadores pueden seguir manejando interrupciones).
     La intención de usar SA_INTERRUPT solo es en situaciones muy
     específicas, ya que en los sistemas modernos no hay una gran
     diferencia entre las interrupciones "rápidas" y "lentas", la única
     diferencia importante es la deshabilitación de otras interrupciones.
SA_RESTORER:
    Esta bandera es usada por las bibliotecas de C para indicar que el
    campo sa_restorer contiene la dirección de la "signal trampoline".

SA_SIGINFO:
    Esta bandera permite al manipulador de la señal tomar tres argumentos.
    Estos tres argumentos son:
       sig:
         Es el número de la señal que causa la invocación del manipulador.
       info:
         Es un apuntador a siginfo_t, el cual es una estructura que contiene
         información sobre la señal.
       ucontext:
         Este es un apuntador a la estructura ucontext_t, esta estructura
         contiene la información de contexto de la señal que es salvada en
         el espacio de usuario por el kernel.

SA_RESTART:
    Esta bandera proporciona un comportamiento compatible con las semánticas
    de una señal BSD reiniciando ciertas system calls a través de las señales.
```

Por último, `sa_restorer`, indica la dirección de "signal trampoline".

Nota: Si `SA_SIGINFO` se especifica en `sa_flags`, entonces `sa_sigaction` será usada para especificar la función "signal-handling" para `signum`.

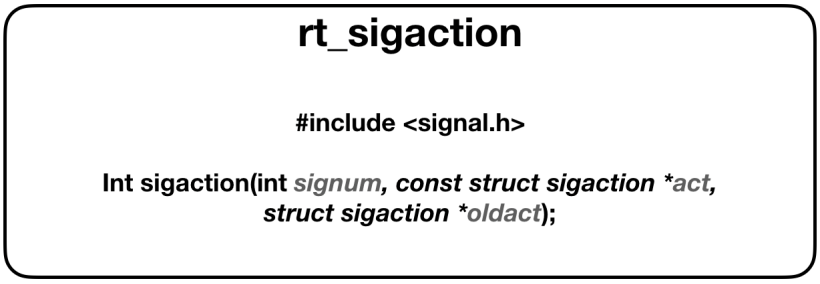

Figura 4.20: Parámetros de rt-sigaction.

La *system call* `rt_sigprocmask` es usada para traer (o cambiar) la máscara de una señal del hilo de llamada. El parámetro `how` describe el comportamiento de `rt_sigprocmask`. A continuación se presentan los tres principales valores que puede adoptar este parámetro:

```
SIG_BLOCK:
     El conjunto de señales bloqueadas es la unión del actual conjunto
     con el argumento set.
SIG_UNBLOCK:
     Las señales del argumento set son eliminadas por el conjunto
     actual de señales bloqueadas.
SIG_SETMASK:
    El conjunto de señales bloqueadas se coloca en el argumento set.
```

Cuando el parámetro `oldset` es diferente a NULL, el valor previo de la mascara de señal se guarda en `oldset`. Si `set` es NULL, la máscara de señal no cambia y su valor actual es guardado en `oldset`. El parámetro `sigsetsize` especifica el tamaño (en bytes) del conjunto de señales en `set` y `oldset`.

Figura 4.21: Parámetros de rt-sigprocmask.

El conjunto de *system calls* `getrlimit`, `setrlimit` y `prlimit` (ver figura 4.22) obtienen y establecen los límites de los recursos, se describen por la estructura `rlimit`:

```
struct rlimit{
   rlim_t rlim_cur;  /*soft limit*/
   rlim_t rlim_max; /*hard limit*/
}
```

El parámetro `rlim_cur` conocido como `soft limit` indica el valor que el *kernel* impone para el recurso correspondiente. Por otro lado, el parámetro `rlim_max` conocido también como `hard limit` funciona como un tope para `soft limit`. Los procesos no privilegiados pueden colocar su `soft limit` desde un valor `0` hasta el valor de `hard limit`. Los procesos privilegiados, por otro lado, pueden cambiar arbitrariamente sus límites.

A continuación se muestran algunos de los valores que el argumento `resource` puede tener:

```
RLIMIT_STACK:
   Este es el máximo tamaño de la pila del proceso, una vez que se llega
   a este límite, una señal SIGSEGV se genera. Este límite también determina
   la cantidad de espacio por los argumentos y variables de entorno de un
   proceso

RLIMIT_CPU:
   Es el límite en segundos de la cantidad de tiempo de CPU  que el proceso
   puede consumir. Cuando el proceso alcanza a soft limit, una señal SIGXCPU
   se envía. Lo que esta señal hace normalmente es terminar el proceso,
   en caso de que no se termine el proceso, una señal SIGXCPU será enviada
   cada segundo hasta que el valor hard limit sea alcanzado y así una señal
   SIGKILL es enviada.
```

La *system call* `prlimit` tiene las propiedades de *setrlimit* y `getrlimit`. Si el parámetro `new_limit` es distinto de NULL, hace uso de la estructura `rlimit` para establecer los límites `soft limit` y `hard limit` para el parámetro `resource`. Si el parámetros `old_limit` es distinto de NULL, entonces los valores previos de `soft limit` y `hard limit` de `resource` son colocados en la estructura `rlimit` llamada `old_limit`.

El parámetro `pid` especifica el ID del proceso, en caso de que `pid = 0`, entonces la llamada se aplica al proceso de llamada.

El valor `RLIM_INFINITY` indica que un recurso no tiene límites.

getrlimit / setrlimit / prlimit

```
#include <linux/time.h>
#include <sys/resource.h>

int getrlimit(int resource, struct rlimit *rlim);
int setrlimit(int resource, const struct rlimit *rlim);

int prlimit(pid_t pid, int resource, const struct rlimit
    *new_limit, struct rlimit *old_limit);
```

Figura 4.22: Parámetros de getrlimit, setrlimit y prlimit.

La *system call* `sysinfo` (ver figura 4.23) devuelve estadísticas del uso de la memoria RAM y swap bajo la siguiente estructura:

```
struct sysinfo{

   long uptime;             /*Segundos transcurridos desde el arranque*/
   unsigned long loads[3];  /*1, 5, y 15 minutos de promedio de carga*/
   unsigned long totalram;  /*Tamaño total de memoria utilizable*/
   unsigned long freeram;   /*Tamaño de memoria disponible*/
   unsigned long shareram;  /*Total de memoria compartida*/
   unsigned long bufferram; /*Memoria usada por buffers*/
   unsigned long totalswap; /*Tamaño total de espacio de swap*/
   unsigned long freeswap;  /*Espacio de swap aún disponible*/
   unsigned short procs;    /*Número de procesos actuales*/
   unsigned long totalhigh; /*Tamaño total de high memory*/
   unsigned long freehigh;  /*Tamaño disponible de high memory*/
   unsigned int mem_unit;   /* Tamaño de unidad de memoria en bytes*/

   char _f[20-2*sizeof(long)-sizeof(int)]; /*padding a 64 bytes*/
```

```
}
```

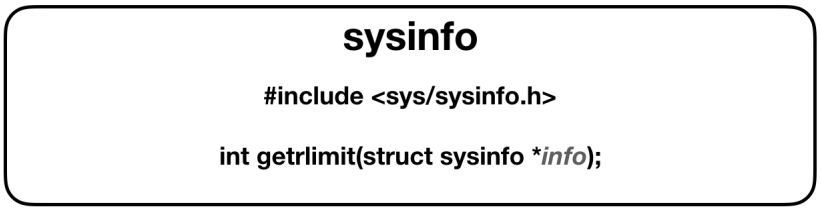

Figura 4.23: Parámetros de sysinfo.

La *system call* `sendto` al igual que `send` y `sendmsg` tiene como objetivo transmitir un mensaje a otro socket, ver figura 4.24.

El parámetro `sockfd` es el archivo descriptor del socket que enviará el mensaje. En `send` y `sendto` el parámetro `buf` contiene el mensaje con una longitud `len`. Normalmente el socket no estará en estado "conectado", por lo que la dirección de la tarjeta se proporciona por los parámetros `dest_addr` y `addrlen`. En caso de que el socket se encuentre en estado "conectado", los parámetros `dest_addr` y `addrlen` serán ignorados.

El parámetro `flags` puede adoptar las siguientes banderas:

```
MSG_CONFIRM     MSG_DONTROUTE     MSG_DONTWAIT     MSG_EOR
MSG_MORE        MSG_NOSIGNAL      MSG_OOB
```

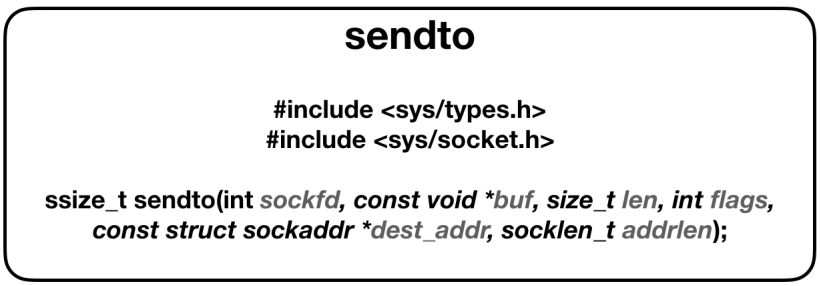

Figura 4.24: Parámetros de sendto.

`Recvmsg` forma parte de un grupo de *system calls* (que incluye `recv` y `recvfrom`) que son usadas para recibir mensajes desde sockets que pueden ser sin conexión u orientados a conexión. Estas *system calls* devuelven la longitud del mensaje en caso de éxito, en caso de que el mensaje sea muy largo para el buffer, los bytes en exceso serán descartados (esto depende del tipo de socket), ver figura 4.25. Estas *system calls* hacen uso del parámetro `flags`, en caso de que este parámetro sea igual a `0` cumpliran la misma función que la *system call* `read`. Como se mencionó anteriormente, el parámetro `flags` es de mucha importancia, ya que sin él este grupo de *system calls* perderían relevancia. `Flags` es formado por valores `0Ring` entre otros como se muestra a continuación:

```
MSG_GMSG_CLOEXEC:
          Esta bandera solo está disponible para la system
          call recvmsg a partir de la versión de
          Linux 2.6.23. Se utiliza esta bandera para
          el descriptor de archivo UNIX usando la
          operación SCM_RIGHTS.
```

```
  MSG_DONTWAIT:
            Esta bandera está disponible a partir de la versión
            de Linux 2.2, tiene la función de permitir la
            operación sin bloqueo, en el caso de que la
            operación sea bloqueada, un error EAGAIN
            o EWOULDBLOCK aparecerá.

  MSG_OOB:
            Esta bandera solicita datos fuera de la banda que no
            debería recibir en el flujo de datos normal. Esta
            bandera no puede ser usada por algunos protocolos.
```

De forma específica, `recvmsg` hace uso de la estructura `msghdr` (dentro de `sys/socket.h`) para minimizar la cantidad de argumentos. A continuación se presenta la estructura:

```
 struct iovec {           /*Dispersar/juntar elementos del arreglo*/
   void   *iov_base;         /*Dirección de inicio*/
   size_t  iov_len;         /*Número de bytes a transferir*/

};

struct msghdr {
   void  *msg_name;              /*Dirección opcional*/
   socklen_t  msg_namelen;       /*Tamaño de la dirección*/
   struct  iovec *msg_iov;       /*Dispersar/juntar arreglo*/
   size_t  msg_iovlen;           /*Número de elementos en msg_iov*/
   void  *msg_control;           /*Datos complementarios*/
   size_t  msg_controllen;       /*longitud del buffer de datos complementarios*/
   int  msg_flags;               /*Banderas del mensaje recibido*/
};
```

Como primer campo de la estructura `msghdr` se tiene `msg_name`, que apunta a un buffer "caller-allocated" el cual regresa la dirección de fuente si el socket está desconectado. Por otro lado, el campo `msg_namelen` deberá ser asignado al tamaño del buffer previamente a la llamada, de este modo este campo contendrá la longitud de la dirección de retorno. Los siguientes dos campos de la estructura son `msg_iov` y `msgiovlen`, los cuales describen la ubicación de dispersión.

Los campos `msg_control` y `msg_controllen` apuntan al buffer para otros mensajes relacionados con el control del protocolo entre otros datos auxiliares y contienen la longitud del buffer en `msg_control`, respectivamente. Cuando la operación es exitosa, la llamada contendrá la longitud del mensaje de control, los cuales tienen la siguiente forma:

```
struct cmsghdr {
   size_t  cmsg_len;  /*contador de bytes de datos (incluyendo el header)*/
   int cmsg_level;    /*protocolo de origen*/
   int cmsg_type;     /*tipo de protocolo*/
};
```

Como último campo de la estructura `msghdr` se encuentran las banderas `msg_flags`, las cuales pueden adoptar los siguientes valores:

```
MSG_EOR:
           Indica el final del registro.
```

```
MSG_TRUNC:
           Indica que la porción de un datagrama fue
           descartada por ser más grande que el buffer.
MSG_CTRUNC:
           Indica que parte de los datos de control fue
           descartada por falta de espacio para datos auxiliares.
MSG_OOB:
           Indica que se recibieron datos fuera de banda.
MSG_ERRQUEUE:
           Indica que los datos no fueron recibidos.
```

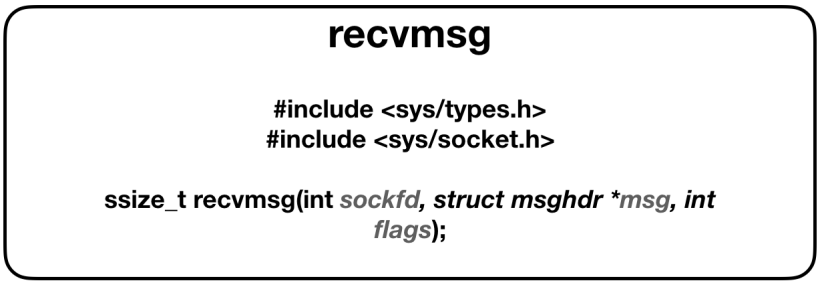

Figura 4.25: Parámetros de recvmsg.

La *system call* `sigreturn` no devuelve ningun valor, pero es importante ya que en caso de que el *kernel* determine que existe una señal desbloqueada pendiente de un proceso, entonces en la siguiente transición el *kernel* crea un nuevo frame en el espacio de usuario en donde guarda el estado del proceso *(estado del procesador, registros, máscara de señal)*. El *kernel* también llama al operador de señales durante la transición al modo de usuario, donde una vez que el operador de señales termina, el control pasa al espacio de usuario llamado *"signal trampoline"*, el cual llama a `sigreturn`. `sigreturn` deshace lo que se hizo, cambia la máscara de señal del proceso, intercambia las pilas de señal para invocar el operador de las señales. Cambiando todos estos parámetros que fueron salvados anteriormente en el espacio de usuario logra hacer que el proceso reanuda la ejecución en el punto en el que fue interrumpido por la señal.

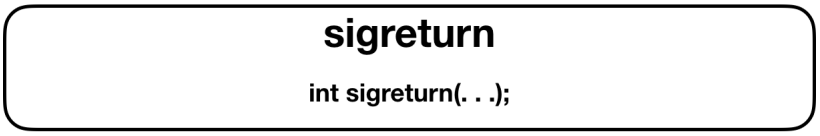

Figura 4.26: Parámetros de sigreturn.

Por último mencionaremos a *exit group*, que es equivalente a *exit* a diferencia que *exit group* se limita a terminar no solo el hilo de llamada, sino también termina los demás hilos del grupo de hilos del proceso de llamada.

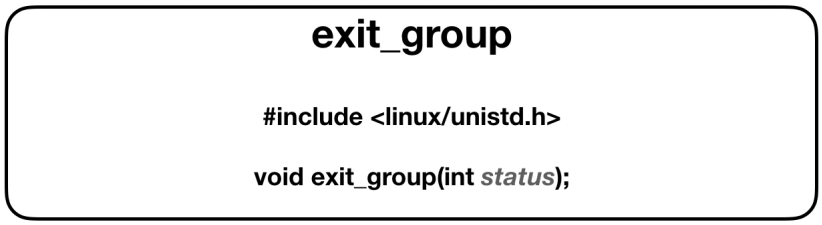

Figura 4.27: Parámetros de exit-group.

4.1. Ejemplos del uso de system calls sobre Xen con OVS

4.1.1. Ping

Como primera prueba se realizó un *ping* desde Dom0 a la máquina virtual 2 (véase figura 3.2), la cual comprende la red `10.10.11.0`. El comando *ping* es usado para corroborar si una red está disponible y si el host es accesible, de este modo es sencillo utilizar este comando para comprobar si un servidor está funcionando, comprobar el funcionamiento de nuestra conexión a internet, si una máquina remota está conectada a la red, entre otras cosas. La sintaxis del comando *ping* es:
`ping [opción] [hostname o dirección IP]`

Para obtener las *system calls* fue a través del comando `strace`, cuya estructura es:
`strace [opción] [comando a ejecutar]` El comando `strace` ejecuta el proceso descrito por el parámetro `comando a ejecutar` mientras permite interceptar y visualizar el nombre, argumentos y valores de retorno de las *system calls* que son llamadas por el proceso.
Mediante `strace` se puede obtener las *system calls* al ejecutar un `ping` desde el host 2 (`10.10.10.2`) hacia la máquina virtual 1:
`strace ping 10.10.11.1`

Como se mencionó con anterioridad, la primera *system call* que es ejecutada al comenzar un proceso es `execve`, la cual ejecuta el programa al que se refiere mediante el primer argumento (`pathname -> /bin/ping`), el siguiente argumento es un arreglo y por último, el argumento `envp` es un apuntador que indica el entorno del nuevo programa:
`execve("/bin/ping", ["ping", "10.10.11.1"], 0x7ffcb36852f8 /* 19 vars */) = 0`

La siguiente *system call* es `brk`, la cual cambia la ubicación de la ruptura del programa, pero cuando se obtiene `brk(NULL) = dirección de memoria` el proceso está pidiendo información de donde termina su memoria acumulada.
`brk(NULL) = 0x55ea20403000`

Posteriormente se llama la *system call openat*:

```
openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libcap.so.2", O_RDONLY|O_CLOEXEC) = 3
```

Como primer argumento aparece `AT_FDCWD`, que indica que `pathname` será relativo al actual directorio del proceso con una bandera `O_RDONLY`, la que indica una petición para solo lectura y también hace uso de `O_CLOEXEC` que evita el funcionamiento de programas "multihilo".

Las siguientes *system calls* son `openat, fstat y mmap`, de las cuales solo se muestran algunas por cuestión de espacio:

```
openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libcap.so.2", O_RDONLY|O_CLOEXEC) = 3
fstat(3, {st_mode=S_IFREG|0644, st_size=54201, ...}) = 0
mmap(NULL, 54201, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7f9a98af5000
fstat(3, {st_mode=S_IFREG|0644, st_size=26864, ...}) = 0
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0)
     = 0x7f9a98af3000
mmap(NULL, 29016, PROT_READ, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x7f9a98aeb000
```

```
mprotect(0x7f9a98aed000, 16384, PROT_NONE) = 0
mmap(0x7f9a98aed000, 8192, PROT_READ|PROT_EXEC,
     MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x2000) = 0x7f9a98aed000
mmap(0x7f9a98aef000, 4096, PROT_READ, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE,
     3, 0x4000) = 0x7f9a98aef000
mmap(0x7f9a98af1000, 8192, PROT_READ|PROT_WRITE,
     MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x5000) = 0x7f9a98af1000
```

Como se puede observar en `openat`, se refiere a otro archivo *libcap.so.2* con las mismas características en los argumentos que anteriormente se analizó. La siguiente *system call* es `fstat`, la cual obtiene información sobre el archivo especificado en su primer argumento *(3)*, que se refiere a libcap.so.2 por ser el valor de retorno de `openat`, después en el segundo argumento se especifica que se trata de un archivo tipo `S_IFREG` que es un archivo regular. Ahora las *system calls* `mmap`, que tienen como primer argumento `*addr`, la cual es una dirección de memoria que comprende el mismo segmento de esta, excepto por la primer *system call* (comienza con NULL), y el valor de retorno es la misma dirección ya que la llamada fue exitosa. El segundo argumento `length` determina hasta donde se hace el mapeo, por lo que la siguiente *system call* `mmap` consecutiva comenzará desde `*addr + length`, donde `*addr` está en hexadecimal y `length` en decimal.

```
Ejemplo para dos mmap consecutivos:

     *addr de primer mmap = 0x7f9a98aed000
     Valor de length en decimal = 8192
     Valor de length en hexadecimal = 2000

     0x7f9a98aed000 + 2000 = 0x7f9a98aef000

     Y el valor *addr del mmap consecutivo es: 0x7f9a98aef000
```

Después, se puede comprobar que la *system call* correspondiente al segundo `mprotect` protege una sección de la memoria desde `addr` a `addr+len-1`, la cual es la sección que ocupan los tres `mmap` consecutivos con la bandera `PROT_NONE`. Esta bandera indica que la memoria no puede ser accedida.

```
0x7f9a98aed000 + 16384 [base decimal] = 0x7f9a98aed000 + 4000 = 0x7f9a98af1000
```

En vista de lo anterior, se puede aseverar que esta primer sección de *system calls* protege con la bandera `PROT_NONE` 4000 bytes de memoria y hace un mapeo total de 111889 bytes.

A partir de este primer análisis se propone una estructura para analizar las siguientes secciones de *system calls*, las cuales seguirán la estructura básica anterior pero compactada. A continuación se muestra dicha estructura:

```
System call               Argumento(s)

openat(*pathname)              .
fstat(*statbuf)[0]             .
mprotect(*addr, len)           .
mmap(*addr, length)            .

Bytes protegidos:
Logitud total del mapeo:
```

*Nota: el argumento *addr del primer mmap será su valor de retorno, ya que el primer mmap de cada sección tiene NULL como argumento*

Segunda sección:

```
System call                    Argumento(s)

openat(*pathname)              (/usr/lib/x86_64-linux-gnu/libidn2.so.0)
fstat(*statbuf)[0]             (st_mode=S_IFREG)
mprotect(*addr, len)           No usa mprotect
mmap(*addr, length)            (0x7f9a98acc000,122904)
mmap(*addr, length)            (0x7f9a98ace000,16384)
mmap(*addr, length)            (0x7f9a98ad2000,94208)
mmap(*addr, length)            (0x7f9a98ae9000,8192)

Bytes protegidos:              0 bytes
Logitud total del mapeo:       241688 bytes
```

Tercera sección:

```
System call                    Argumento(s)

openat(*pathname)              (/usr/lib/x86_64-linux-gnu/libnettle.so.6)
fstat(*statbuf)[0]             (st_mode=S_IFREG)
mprotect(*addr, len)           No usa mprotect
mmap(*addr, length)            (0x7f9a98a94000,226472)
mmap(*addr, length)            (0x7f9a98a9d000,118784)
mmap(*addr, length)            (0x7f9a98aba000,61440)
mmap(*addr, length)            (0x7f9a98ac9000,12288)

Bytes protegidos:              0 bytes
Logitud total del mapeo:       418984 bytes
```

Cuarta sección:

```
System call                    Argumento(s)

openat(*pathname)              (/lib/x86_64-linux-gnu/libresolv.so.2)
fstat(*statbuf)[0]             (st_mode=S_IFREG)
mmap(*addr, length)            (0x7f9a98a7a000,105088)
mprotect(*addr, len)           (0x7f9a98a7e000,73728)
mmap(*addr, length)            (0x7f9a98a7e000,53248)
mmap(*addr, length)            (0x7f9a98a8b000,16384)
mmap(*addr, length)            (0x7f9a98a90000,8192)
mmap(*addr, length)            (0x7f9a98a92000,6784)

Bytes protegidos:              73728 bytes
Logitud total del mapeo:       189696 bytes
```

Quinta sección:

```
System call                    Argumento(s)

openat(*pathname)              (/lib/x86_64-linux-gnu/libm.so.6)
fstat(*statbuf)[0]             (st_mode=S_IFREG)
mprotect(*addr, len)           No usa mprotect
mmap(*addr, length)            (0x7f9a988f7000,1581384)
mmap(*addr, length)            (0x7f9a98904000,651264)
mmap(*addr, length)            (0x7f9a989a3000,872448)
mmap(*addr, length)            (0x7f9a98a78000,8192)

Bytes protegidos:              0 bytes
Logitud total del mapeo:       3113288 bytes
```

Sexta sección:

```
System call                    Argumento(s)

openat(*pathname)              (/lib/x86_64-linux-gnu/libc.so.6)
fstat(*statbuf)[0]             (st_mode=S_IFREG)
mmap(*addr, length)            (0x7f9a98736000,1837056)
mprotect(*addr, len)           (0x7f9a98758000,1658880)
mmap(*addr, length)            (0x7f9a98758000,1343488)
mmap(*addr, length)            (0x7f9a988a0000,311296)
mmap(*addr, length)            (0x7f9a988ed000,24576)
mmap(*addr, length)            (0x7f9a988f3000,14336)

Bytes protegidos:              1658880 bytes
Logitud total del mapeo:       3530752 bytes
```

Septima sección:

```
System call                    Argumento(s)

openat(*pathname)              (/usr/lib/x86_64-linux-gnu/libunistring.so.2)
fstat(*statbuf)[0]             (st_mode=S_IFREG)
mprotect(*addr, len)           No usa mprotect
mmap(*addr, length)            (0x7f9a98734000,8192)
mmap(*addr, length)            (0x7f9a985b0000,1587464)
mmap(*addr, length)            (0x7f9a985c1000,225280)
mmap(*addr, length)            (0x7f9a985f8000,1273856)
mmap(*addr, length)            (0x7f9a98a78000,20480)

Bytes protegidos:              0 bytes
Logitud total del mapeo:       3115272 bytes
```

Al terminar estas secciones se usa la *system call* `munmap` para eliminar el mapeo descrito por sus parámetros `addr` y `length` que es la sección de memoria utilizada por el primer `mmap` de todo el proceso:

```
munmap(0x7f9a98af5000, 54201) = 0
```

En la siguiente sección de *system calls* se hace uso de `captget, capset` y `socket`:

```
capget({version=_LINUX_CAPABILITY_VERSION_3, pid=0}, NULL) = 0
capget({version=_LINUX_CAPABILITY_VERSION_3, pid=0}, {effective=0,
   permitted=1<<CAP_NET_ADMIN|1<<CAP_NET_RAW, inheritable=0}) = 0
capset({version=_LINUX_CAPABILITY_VERSION_3, pid=0}, {effective=1<<CAP_NET_RAW,
   permitted=1<<CAP_NET_ADMIN|1<<CAP_NET_RAW, inheritable=0}) = 0
socket(AF_INET, SOCK_DGRAM, IPPROTO_ICMP) = -1 EACCES (Permission denied)
socket(AF_INET, SOCK_RAW, IPPROTO_ICMP) = 3
socket(AF_INET6, SOCK_DGRAM, IPPROTO_ICMPV6) = -1 EACCES (Permission denied)
socket(AF_INET6, SOCK_RAW, IPPROTO_ICMPV6) = 4
```

Las primeras dos *system calls* que son ejecutadas en esta sección son `capget`. En su primer argumento se puede comprobar que su estructura pertenece a la versión 3 agregada en la versión de Linux 2.6.26, de igual manera esto se comprueba en la *system call* `capset`. Los argumentos de la estructura `__user_cap_data_struct` de capset son:

```
__u32 effective = 1<<CAP_NET_RAW
__u32 permitted = 1<<CAP_NET_RAW|1<<CAP_NET_RAW
__u32 inheritable = 0
```

Donde asigna la capacidad `CAP_NET_RAW` al argumento `effective`. Después se intenta crear el endpoint asignando al argumento `domain` la bandera `AF_INET`, que indica la creación de un socket IPv4. El segundo argumento es `type`, el cual tiene la bandera `SOCK_DGRAM` para soportar mensajes sin conexión y poco fiables. Esta *system call* tiene como valor de retorno -1, lo que indica un falla. Después intenta crear un socket raw IPv4 (contiene las banderas `domain=AF_INET` y `type=SOCK_RAW`) el cual es exitoso. De forma similar se intenta crear un socket IPV6 pero al fallar se crea un socket raw IPv6. Los valores de retorno para la creación de los sockets IPv4 e IPv6 son `3` y `4` respectivamente, los cuales se usaran para referirse a los sockets creados.

En la siguiente sección de *system calls* se repite el mismo procedimiento anterior, pero ahora se hace uso de `capget` el cual comprueba que las capacidades para `effective` es `1<<CAP_NET_RAW`, y con esto se asigna un `0` mediante `capset`. Al asignar `effective = 0`, la creación del socket se considera exitosa:

```
socket(AF_INET, SOCK_DGRAM, IPPROTO_IP) = 5
connect(5, {sa_family=AF_INET, sin_port=htons(1025),
   sin_addr=inet_addr("10.10.11.1")}, 16) = 0
getsockname(5, {sa_family=AF_INET, sin_port=htons(46211),
   sin_addr=inet_addr("10.10.10.2")}, [16]) = 0
```

En este caso la *system call socket* es exitosa usando la bandera `SOCK_DGRAM` sobre un socket IPv4. El siguiente paso es conectar el socket, por lo que se usa la *system call* `connect` que se usará para conectar el socket a la dirección `addr` y al puerto UDP 1025. La siguiente *system call* es `getsockname`, la cual obtiene la dirección a la que el socket esta ligado (`sin_port=htons(46211), sin_addr=inet_addr("10.10.10.2")`) y es usada para inicializar el total de espacio en bytes que indica `socklen_t = [16]`.

La siguiente sección de *system calls* son *setsockopt* y *getsockopt*:

```
setsockopt(3, SOL_RAW, ICMP_FILTER,
   ~(1<<ICMP_ECHOREPLY|1<<ICMP_DEST_UNREACH|1<<ICMP_SOURCE_QUENCH|
   1<<ICMP_REDIRECT|1<<ICMP_TIME_EXCEEDED|1<<ICMP_PARAMETERPROB), 4) = 0
setsockopt(3, SOL_IP, IP_RECVERR, [1], 4) = 0
setsockopt(3, SOL_SOCKET, SO_SNDBUF, [324], 4) = 0
setsockopt(3, SOL_SOCKET, SO_RCVBUF, [65536], 4) = 0
getsockopt(3, SOL_SOCKET, SO_RCVBUF, [131072], [4]) = 0
```

En las anteriores *system calls* el socket de referencia es 3, el cual es el socket IPv4 creado con anterioridad. Se asigna `SOL_SOCKET` al argumento `level` en ambas `setsockopt` y se agrega `IP_SNDBUF` (para enviar el tamaño del bufer) y `IP_RCVBUF` (para recibir el tamaño del buffer) al argumento `optname` de la penultima y ultima *system call* respectivamente. Los parámetros `optval` y `optlen` (`optlen` contiene el tamaño del buffer apuntador por `optval`) servirán para acceder a los valores de opción de *setsockopt*. La *system call* `getsockopt` es usado para obtener las opciones previamente descritas.

```
rt_sigaction(SIGINT, {sa_handler=0x55ea1ec38e30, sa_mask=[],
   sa_flags=SA_RESTORER|SA_INTERRUPT, sa_restorer=0x7f9a9876d840},
   NULL, 8) = 0
rt_sigaction(SIGALRM, {sa_handler=0x55ea1ec38e30, sa_mask=[],
   sa_flags=SA_RESTORER|SA_INTERRUPT, sa_restorer=0x7f9a9876d840},
   NULL, 8) = 0
rt_sigaction(SIGQUIT, {sa_handler=0x55ea1ec38e20, sa_mask=[],
   sa_flags=SA_RESTORER|SA_INTERRUPT, sa_restorer=0x7f9a9876d840},
   NULL, 8) = 0
rt_sigprocmask(SIG_SETMASK, [], NULL, 8) = 0
```

En la sección anterior se hace uso de `rt_sigaction` y `rt_sigprocmask`. Podemos ver que en las tres *system calls* `rt_sigaction` se ingresan los mismos argumentos para los campos

`act` y `oldact`, lo cual tiene sentido ya que se trata de la misma acción. Lo importante es el cambio en el campo `signum` porque este determina la señal. Lo que se puede observar es que hace uso de `SIGINT`, `SIGALRM` y `SIGQUIT`, las cuales indican la finalización de un proceso por medio de `Control-C`, el "timer"finalizó y que existe una salida por teclado respectivamente. La *system call* `rt _sigprocmask` describe, por medio del campo `how = SIG_SETMASK` que el conjunto de señales bloqueadas se colocan en el campo `set([])` cuyo tamaño es `sigsetsize = 8 bytes`, también se aprecia que no hay nada guardado en el campo `oldset`.

En la siguiente sección de *system calls* se hace uso de `sendto` y `recvmsg`. Primero se usa `sendto`, como primer argumento se puede observar `sockfd = 3`, lo que indica que se trabaja sobre el archivo descriptor del socket *3* IPv4 creado anteriormente. El siguiente argumento es `buf`, el cual contiene el mensaje con una longitud descrita por el valor `len`. Ya que el socket inicialmente no esta conectado, se proporciona la dirección de la tarjeta por medio de los parámetros `dest_addr` y `addrle` que tienen como valor `sa_family=AF_INET, bsin_port=htons(0), sin_addr=inet_addr("10.10.11.1")` y `16` respectivamente. Esta dirección corresponde a la dirección asignada al socket *5* IPv4 mediante la *system call connect*, sin embargo el número de puerto no se especifica en el argumento de `sendto`. Después dos *system calls* `recvmsg` son llamadas, las dos tienen como primer argumento `sockfd = 3`, lo cual indica que el socket 3 IPv4 es el que recibirá los mensajes. El siguiente argumento `msg` se refiere a la estructura `msghdr` a continuación se mencionan los campos más relevantes:

```
msg_name:
     Esta es la dirección de la cual recibe la información. Como argumento
     se asigna {sa_family=AF_INET, sin_port=htons(0),
     sin_addr=inet_addr("10.10.11.1")} y por el tipo de socket y la IP se
     puede notar que corresponde a la dirección asignada al socket 5
     IPv4.
msg_namelen:
     Este es el tamaño de la dirección, el cual tiene como valor 128->16.
```

Como último argumento de la primer *system call recvmsg* tenemos `flags = 0`, lo que indica que *recvmsg* desarrolla las mismas funciones que la *system call read*.

La segunda *system call recvmsg* solo asigna al campo `msg_namelen=128` de la estructura `msghdr`, pero ocurre un error ya que el valor de retorno es `-1`.

Después se ejecuta esta sección de *system calls* (en las que solo algunos parámetros de la structura `msghdr` cambian) hasta que detenemos la ejecución de *ping*.

```
sendto(3, "\10\0\10|\1\301\0\1Yj\373^\0\0\0\0\321%\t\0\0\0\0\0\20\21\22
   \23\24\25\26\27"..., 64, 0, {sa_family=AF_INET, sin_port=htons(0),
   sin_addr=inet_addr("10.10.11.1")}, 16) = 64
recvmsg(3, {msg_name={sa_family=AF_INET, sin_port=htons(0),
   sin_addr=inet_addr("10.10.11.1")}, msg_namelen=128->16,
   msg_iov=[{iov_base="E\0\0TI\352\0\0@\1\7\251\n\n\v\1\n\n\n\2\0\0\20|
   \1\301\0\1Yj\373^"..., iov_len=192}], msg_iovlen=1,
   msg_control=[{cmsg_len=32, cmsg_level=SOL_SOCKET,
   cmsg_type=SCM_TIMESTAMP, cmsg_data={tv_sec=1593535065,
   tv_usec=599942}}], msg_controllen=32, msg_flags=0}, 0) = 84
recvmsg(3, {msg_namelen=128}, 0) = -1 EAGAIN (Resource
   temporarily unavailable)
```

Es en el momento de parar el *ping* cuando se obtiene esta ultima sección de *system calls*:

```
rt_sigreturn({mask=[]}) = -1 EINTR (Interrupted system call)
exit_group(0)
```

En la última sección de *system calls* son llamas *rt sigreturn* y *exit group*. *rt sigreturn* cumple el propósito de verificar los procesos pendientes del *kernel* para así guardar el contexto del proceso para que en un futuro se reanude la ejecución en el punto donde fue interrumpido

por una señal. El valor de retorno de esta *system call* es `-1 EINTR`, lo cual indica que fue interrumpida por una señal.

La última *system call* que obtenemos por medio de `strace` al analizar el *ping* es *exit group*, la cual termina todos los hilos del grupo del proceso de llamada.

4.1.2. Envio de mensaje mediante TCP/IP

La siguiente prueba envía un mensaje mediante el uso de sockets TCP en Python, en esta prueba se usan dos programas, uno para el cliente en la máquina virtual 1 y otro para el servidor en la máquina virtual 2.

Durante esta prueba se analizan ambas máquinas virtuales *(cliente y servidor)*.

4.1.2.1. Cliente (send.py)

El script de Python utilizado ejecutado del lado del cliente para el envío de paquetes es el siguiente:

```
import socket

ip = '10.10.11.2'
port = 5005
buffer = 1024
msg = 'Trying out TCP socket'
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((ip,port))

for i in range (1000):
   s.send(msg)
   data = s.recv(buffer)
   print ('ack:', data)

s.close()
```

De forma similar a la prueba del *ping*, la salida en terminal que se obtiene al venir realizando la ejecución del programa *send.py* es la siguiente:

```
1 tesis@dell-inspiron:/ strace python send.py
2 execve("/usr/bin/python",["python","send.py"],0x7ffd0e71c958 /* 19 vars
   */) = 0
3 brk(NULL) = 0x55f701856000
4 access("/etc/ld.so.preload", R_OK) = -1
   ENOENT (No such file or directory)
5 openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
6 fstat(3, {st_mode=S_IFREG|0644, st_size=54201, ...}) = 0
7 mmap(NULL, 54201, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7f6b62159000
8 close(3) = 0
```

En esta sección se ejecuta el programa del primer argumento de `execve` y su último argumento es un apuntador al entorno del nuevo programa. Después el proceso solicita información de la ubicación donde termina su memoria acumulada por medio de *brk*. El valor de retorno de *system call access* indica que el componente `pathname = /etc/ld.so.preload` no existe. Por medio de la *system call openat* se indica que `pathname = /etc/ld.so.cache` será relativo al actual directorio del proceso, y la bandera `O_RDONLY` indica que la petición será de solo lectura, mientras la bandera `O_CLOEXEC` indica que se evitarán funciones que impiden el funcionamiento de los programas "multihilo", el valor de retorno de esta *system call* es `3`, por lo que se referirá como `3` a esta *system call*. La siguiente *system call* es `fstat`, la cual

tiene como valor `fd = 3`, esto indica que el primer argumento se refiere a `/etc/ld.so.cache`, de su segundo argumento se sabe que `st_mode = S_IFREG` es un archivo regular. Por último, `mmap` indica que desde la dirección `0x7f6b62159000` a `0x7f6b62159000 + 54201 bytes` el mapeo será privado (donde múltiples procesos pueden solicitar recursos iguales), de solo lectura.

Para analizar las siguientes *system calls* se utilizará la estructura:

```
System call               Argumento(s)

openat(*pathname)             .
fstat(*statbuf)[0]            .
mprotect(*addr, len)          .
mmap(*addr, length)           .

Bytes protegidos:
Logitud total del mapeo:
```

Primer sección:

```
System call               Argumento(s)

openat(*pathname)             (/lib/x86_64-linux-gnu/libpthread.so.0)
fstat(*statbuf)[0]            (st_mode=S_IFREG)
mprotect(*addr, len)          No usa mprotect
mmap(*addr, length)           (0x7f6b62157000,8192)
mmap(*addr, length)           (0x7f6b62136000,132288)
mmap(*addr, length)           (0x7f6b6213c000,61440)
mmap(*addr, length)           (0x7f6b6214b000,24576)
mmap(*addr, length)           (0x7f6b62151000,8192)
mmap(*addr, length)           (0x7f6b62153000,13504)

Bytes protegidos:         0 bytes
Logitud total del mapeo:  248192 bytes
```

Segunda sección:

```
System call               Argumento(s)

openat(*pathname)             (/lib/x86_64-linux-gnu/libdl.so.2)
fstat(*statbuf)[0]            (st_mode=S_IFREG)
mprotect(*addr, len)          No usa mprotect
mmap(*addr, length)           (0x7f6b62131000,16656)
mmap(*addr, length)           (0x7f6b62132000,4096)
mmap(*addr, length)           (0x7f6b62133000,4096)
mmap(*addr, length)           (0x7f6b62134000,8192)

Bytes protegidos:         0 bytes
Logitud total del mapeo:  33040 bytes
```

Tercera sección:

```
System call               Argumento(s)

openat(*pathname)             (/lib/x86_64-linux-gnu/libutil.so.1)
fstat(*statbuf)[0]            (st_mode=S_IFREG)
mprotect(*addr, len)          No usa mprotect
mmap(*addr, length)           (0x7f6b6212c000,16656)
mmap(*addr, length)           (0x7f6b6212d000,4096)
mmap(*addr, length)           (0x7f6b6212e000,4096)
```

```
mmap(*addr, length)           (0x7f6b6212f000,8192)

Bytes protegidos:         0 bytes
Logitud total del mapeo:  33040 bytes
```

Cuarta sección:

```
System call               Argumento(s)

openat(*pathname)             (/lib/x86_64-linux-gnu/libz.so.1)
fstat(*statbuf)[0]            (st_mode=S_IFREG)
mmap(*addr, length)           (0x7f6b61f0e000,2216336)
mprotect(*addr, len)          (0x7f6b61f2a000,2097152)
mmap(*addr, length)           (0x7f6b6212a000,8192)

Bytes protegidos:         2097152 bytes
Logitud total del mapeo:  2224528 bytes
```

Quinta sección:

```
System call               Argumento(s)

openat(*pathname)             (/lib/x86_64-linux-gnu/libm.so.6)
fstat(*statbuf)[0]            (st_mode=S_IFREG)
mprotect(*addr, len)          No usa mprotect
mmap(*addr, length)           (0x7f6b61d8b000,1581384)
mmap(*addr, length)           (0x7f6b61d98000,651264)
mmap(*addr, length)           (0x7f6b61e37000,872448)
mmap(*addr, length)           (0x7f6b61f0c000,8192)

Bytes protegidos:         0 bytes
Logitud total del mapeo:  3113288 bytes
```

Sexta sección:

```
System call               Argumento(s)

openat(*pathname)             (/lib/x86_64-linux-gnu/libc.so.6)
fstat(*statbuf)[0]            (st_mode=S_IFREG)
mmap(*addr, length)           (0x7f6b61bca000,1837056)
mprotect(*addr, len)          (0x7f6b61bec000,1658880)
mmap(*addr, length)           (0x7f6b61bec000,1343488)
mmap(*addr, length)           (0x7f6b61d34000,311296)
mmap(*addr, length)           (0x7f6b61d81000,24576)
mmap(*addr, length)           (0x7f6b61d87000,14336)

Bytes protegidos:         1658880 bytes
Logitud total del mapeo:  3530752 bytes
```

No se describen las secciones pasadas ya que son muy parecidas al ejmplo anterior del *ping*. Después de ejecutar estas acciones, se usa *munmap* para eliminar el mapeo desde `addr` a `addr + length`, esta sección de memoria es la sección usada por el primer mmap que obtenemos al ejecutar el programa *send.py*.

```
munmap(0x7f6b62159000, 54201) = 0

set_tid_address(0x7f6b61bc5a10) = 580
set_robust_list(0x7f6b61bc5a20, 24) = 0
rt_sigaction(SIGRTMIN, {sa_handler=0x7f6b6213c6b0, sa_mask=[],
```

```
    sa_flags=SA_RESTORER|SA_SIGINFO, sa_restorer=0x7f6b62148730},
    NULL, 8) = 0
rt_sigaction(SIGRT_1, {sa_handler=0x7f6b6213c740, sa_mask=[],
    sa_flags=SA_RESTORER|SA_RESTART|SA_SIGINFO, sa_restorer=0x7f6b62148730},
    NULL, 8) = 0
rt_sigprocmask(SIG_UNBLOCK, [RTMIN RT_1], NULL, 8) = 0
prlimit64(0, RLIMIT_STACK, NULL, {rlim_cur=8192*1024,
    rlim_max=RLIM64_INFINITY}) = 0
ioctl(0, TCGETS, {B38400 opost isig icanon echo ...}) = 0
brk(NULL) = 0x55f701856000
brk(0x55f701877000) = 0x55f701877000
```

Posteriormente se utiliza *set-tid-address*, la cual establece un valor `clear_child_tid` al hilo en `0x7f6b61bc5a10` cuyo ID es `580`. El valor que es otorgado al hilo ID `580` permite realizar un *futex* en la memoria. La siguiente *system call* utilizada es *set-robust-list*, la cual tiene como parámetro un apuntador al *head* de la lista robusta de futex, la cual se diferencia solo con 16 bytes respecto a la dirección del hilo `ID=580` en `0x7f6b61bc5a10`. Después se utiliza dos veces *rt-sigaction*. En la primera *system call rt-sigaction*, se utiliza como primer parámetro `signum = SIGRTMIN`, la cual es una señal que determina el rango de señales en tiempo real. En su segundo argumento se hace referencia a la acción asociada con `signum = SIGRTMIN` mediante `sa_handler = 0x7f6b6213c6b0` y usando las banderas `SA_RESTORER` y `SA_SIGINFO` que indican que el campo `sa_restorer` contiene la dirección de la "signal trampoline" (`SA_RESTORER`) y permiten al manejador de la señal tomar tres parámetros en vez de solo 1 (`SA_SIGINFO`). Por último, `sa_restorer` indica la dirección de la "signal trampoline".

La segunda *system call rt-sigaction* tiene como argumento `signum = SIGRT_1`, la cual es una señal utilizada para realizar cambios de credenciales a lo largo del proceso, en su segundo argumento se utiliza otra dirección en `sa_handler`, la razón es porque ahora se refiere a la acción asociada con `signum`. A diferencia de la anterior `rt_sigaction`, adicionalmente a `SA_RESTORER` y `SA_SIGINFO` se utiliza `SA_RESTART`, la cual la hace compatible con señales BSD. Por último, el parámetro `sa_restorer` es igual a la *system call* anterior, que indica que usan la misma "signal trampoline".

La *system call rt-sigprocmask* se utiliza para cambiar la máscara de la señal del hilo de la llamada. El parámetro `how = SIG_UNBLOCK`, indica que las señales del argumento `set = [RTMIN RT_1]` son eliminadas por el conjunto actual de señales bloqueadas. La *system call prlimit* permite establecer los limites de los recursos del proceso, su argumento `pid = 0`, indica que la llamada se aplicará al proceso actual, mientras que el argumento `resource = RLIMIT_STACK` indica el máximo valor para el tamaño de la pila del proceso. Una vez alcanzado este límite, una señal SIGSEGV será enviada. El último argumento es la estructura `old_limit = {rlim_cur=8192*1024, rlim_max=RLIM64_INFINITY}`, la cual indica los valores que se colocarán para los límites en la estructura. Es importante mencionar que para el valor de `rlim_max (hard limit)` se utiliza el valor `RLIM64_INFINITY`, el cual indica que el recurso no tiene límites.

```
mmap(NULL, 262144, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1,
 0) = 0x7f6b61b85000
 stat("/usr/bin/python", {st_mode=S_IFREG|0755, st_size=3689352, ...})
    = 0
 stat("/usr/lib/python2.7/os.py", {st_mode=S_IFREG|0644, st_size=25910, ...})
     = 0
 stat("/usr/lib/python2.7/lib-dynload", {st_mode=S_IFDIR|0755, st_size=4096,
   ...}) = 0
mmap(NULL, 262144, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0)
 = 0x7f6b61b45000
 rt_sigaction(SIGPIPE, {sa_handler=SIG_IGN, sa_mask=[], sa_flags=SA_RESTORER,
 sa_restorer=0x7f6b62148730}, {sa_handler=SIG_DFL, sa_mask=[], sa_flags=0},
     8) = 0
```

```
rt_sigaction(SIGXFSZ, {sa_handler=SIG_IGN, sa_mask=[], sa_flags=SA_RESTORER,
sa_restorer=0x7f6b62148730}, {sa_handler=SIG_DFL, sa_mask=[], sa_flags=0},
     8) = 0
 getpid() = 580
  .
  .
  .
stat("/usr/lib/python2.7", {st_mode=S_IFDIR|0755, st_size=20480, ...}) = 0
openat(AT_FDCWD, "/usr/lib/python2.7/site.py", O_RDONLY) = 3
fstat(3, {st_mode=S_IFREG|0644, st_size=19947, ...}) = 0
openat(AT_FDCWD, "/usr/lib/python2.7/site.pyc", O_RDONLY) = 4
fstat(4, {st_mode=S_IFREG|0644, st_size=19535, ...}) = 0
fstat(4, {st_mode=S_IFREG|0644, st_size=19535, ...}) = 0
  close(4) = 0
```

La sección anterior de *system calls* comienza haciendo un mapeo desde la dirección `0x7f6b61b85000` y hasta `0x7f6b61b85000 + 262144[bytes]`, este mapeo protege la memoria mediante `prot = PROT_READ/PROT_WRITE`, lo que indica que puede ser leída y también se puede escribir en ella. La forma en la cual las actualizaciones del mapeo serán visibles para otros proceso en la región se determina por `flags = MAP_PRIVATE/MAP_ANONYMOUS`, las cuales indican que el mapeo será privado (lo que indica que las actualizaciones no son visibles) y que el mapeo no es respaldado por ningún archivo. Después se solicita información del archivo `pathname = /usr/bin/python` mediante la *system call stat*, y se extrae el tipo de archivo de la estructura `statbuf`, el cual es del tipo `S_IFREG`, es decir, es un archivo regular. Posteriormente, dos *system calls stat* se ejecutan solicitando información de `/usr/lib/python2.7/os.py` y `/usr/lib/python2.7/lib-dynload` cuya descripción es `st_mode=S_IFREG` (archivo regular) y `st_mode=S_IFDIR` (representa un directorio), respectivamente. La siguiente *system call* en ser ejecutada es `mmap`, la cual hace un mapeo con las mismas caracteristicas del primer `mmap` de esta sección de *system calls*, a diferencia que el mapeo inicia en `0x7f6b61b45000`. Las siguientes dos *system calls* son `rt-sigaction`, las cuales cambian la acción de un proceso al recibir una señal, las señales que se presentan son `SIGPIPE` y `SIGXFSZ`, las cuales indican *"pipe"* roto: escribir en el *"pipe"* pero sin lectores y que el limite del tamaño del archivo es excedido, con lo cual se termina el proceso. Las acciones a las que están asociadas las señales anteriores son `sa_handler=SIG_DFL`, la cual indica que la acción por defecto asociada a la señal ocurrirá. El `ID = 580` del hilo de llamada mencionado anteriormente se obtiene mediante el identificador del proceso usando `getpid\verb`. Después de una serie de *system calls* `rt_sigaction` ocurren (representado en la sección de *system calls* como tres puntos) al terminar `getpid`. Todas estas *system calls* tienen la estructura `sigaction` como se muestra a continuación:

```
rt_sigaction(SIGHUP, NULL, {sa_handler=SIG_DFL, sa_mask=[], sa_flags=0},
   8) = 0
```

El único parámetro que cambia entre todas ellas es el primer parámetro (`signum`). Para todas ellas el primer elemento de la estructura `sigaction` es `sa_handler=SIG_DFL`, que indica que la acción asociada a la señal ocurrirá. Las señales de cada *system call* son:

```
SIGHUP:    Se detecta el fin del proceso de control.
SIGINT:    Interrupción desde el teclado.
SIGQUIT:   Salida desde el teclado.
SIGILL:    Instrucción ilegal.
SIGTRAP:   Trampa de rastreo/interrupción.
SIGABRT:   Abortar la señal desde "abort(3)"
SIGBUS:    Mal acceso de memoria.
SIGFPE:    Excepción de punto flotante.
SIGKILL:   Destruye la señal.
SIGUSR1:   Señal definida por el usuario 1.
SIGSEGV:   Referencia de memoria inválida.
```

```
SIGUSR2:   Señal definida por el usuario 2.
SIGPIPE:   Pipe roto.
SIGALRM:   Señal de timer.
SIGTERM:   Señal de terminación.
SIGSTKFLT: Falla en el stack.
SIGCHLD:   Child detenido o terminado.
SIGCONT:   Señal para continuar en caso de estar detenido algún proceso.
SIGSTOP:   Detener el proceso.
SIGTSTP:   Stop escrito desde terminal.
SIGTTIN:   Entrada a terminal por el proceso en background.
SIGTTOU:   Salida en terminal por el proceso en background.
SIGURG:    Condición urgente en el socket.
SIGXCPU:   El límite de tiempo de CPU fue excedido.
SIGXFSZ:   El límite de tamaño del archivo fue excedido.
SIGVTALRM: Alarma de reloj virtual.
SIGPROF:   El temporizador "profiling" expiró.
```

Las últimas *system calls* `sigaction` tienen como argumento `signum = SIGRT_#`, donde `#` empieza desde 2 a 32. Estas *system calls* indican un cambio de credencial a lo largo del proceso. Después se solicita información del archivo en la ruta de `stat`, la cual es de tipo `st_mode = S_IFDIR`, lo que indica que es un directorio. Por último se utiliza `openat` para abrir los archivos :
`/usr/lib/python2.7/site.py` y `/usr/lib/python2.7/site.pyc` en modo "solo lectura", después se obtiene la informaión de los archivos mediante *fstat*, la cual gracias a `st_mode=S_IFREG` nos indica que son archivos regulares.

Las siguiente sección de *system calls* hacen uso de `openat` para abrir archivos y después `fstat` para obtener qué tipo de archivo es. A continuación se muestran los archivos que fueron usados por estas dos *system calls*:

```
La estructura en la que se muestran los archivos es la siguiente:

pathname                                    dirfd           st_mode
/usr/lib/python2.7/file                      .              .


os.py | os.pyc                              AT_FDCWD     st_mode=S_IFREG
   sysinfo({uptime=502, loads=[0, 768, 0], totalram=514121728,
      freeram=367149056, sharedram=819200, bufferram=9162752,
      totalswap=536866816, freeswap=536866816, procs=70,
      totalhigh=0, freehigh=0, mem_unit=1}) = 0
posixpath.py | posixpath.pyc                AT_FDCWD     st_mode=S_IFREG
stat.py | stat.pyc                          AT_FDCWD     st_mode=S_IFREG
genericpath.py | genericpath.pyc            AT_FDCWD     st_mode=S_IFREG
warnings.py | warnings.pyc                  AT_FDCWD     st_mode=S_IFREG
linecache.py | linecache.pyc                AT_FDCWD     st_mode=S_IFREG
types.py | types.pyc                        AT_FDCWD     st_mode=S_IFREG
UserDict.py | UserDict.pyc                  AT_FDCWD     st_mode=S_IFREG
_abcoll.py | _abcoll.pyc                    AT_FDCWD     st_mode=S_IFREG
abc.py | abc.pyc                            AT_FDCWD     st_mode=S_IFREG
_weakrefset.py | _weakrefset.pyc            AT_FDCWD     st_mode=S_IFREG
copy_reg.py | copy_reg.pyc                  AT_FDCWD     st_mode=S_IFREG
traceback.py | traceback.pyc                AT_FDCWD     st_mode=S_IFREG
sysconfig.py | sysconfig.pyc                AT_FDCWD     st_mode=S_IFREG
re.py | re.pyc                              AT_FDCWD     st_mode=S_IFREG
sre_compile.py | sre_compile.pyc            AT_FDCWD     st_mode=S_IFREG
sre_parse.py | sre_parse.pyc                AT_FDCWD     st_mode=S_IFREG
sre_constants.py |sre_constants.pyc         AT_FDCWD     st_mode=S_IFREG
```

```
_sysconfigdata.py | _sysconfigdata.py       AT_FDCWD     st_mode=S_IFREG
plat-x86_64-linux-gnu/_sysconfigdata_nd.py  AT_FDCWD     st_mode=S_IFREG
plat-x86_64-linux-gnu/_sysconfigdata_nd.pyc AT_FDCWD     st_mode=S_IFREG
/usr/local/lib/python2.7/dist-packages      AT_FDCWD     S_IFDIR|S_ISGID
dist-packages                               AT_FDCWD     S_IFDIR
sitecustomize.py | sitecustomize.py         AT_FDCWD     st_mode=S_IFREG
lib-tk                                         ·         S_IFDIR
lib-dynload                                    ·         S_IFDIR
dist-packages                                  ·         S_IFDIR
encodings                                      ·         S_IFDIR
encodings/__init__.py                       AT_FDCWD     S_IFREG
encodings/__init__.pyc                      AT_FDCWD     S_IFREG
codecs.py/codecs.pyc                        AT_FDCWD     S_IFREG
encodings/aliases.py | aliases.pyc          AT_FDCWD     S_IFREG
encodings/ascii.py | ascii.pyc              AT_FDCWD     S_IFREG
/root/send.py                               AT_FDCWD     st_mode=S_IFREG
```

En la sección anterior se obtuvo la *system call* `sysinfo`, esta *system call* nos indica que el total de memoria ram utilizable es de `514121728 bytes`, la memoria ram libre es de `367149056 bytes`, la memoria ram compartida es de `819200 bytes` y el total de memoria ram en buffers es de `9162752 bytes`. También nos indica que el total de espacio de swap es de `536866816 bytes`, el espacio de swap libre es de `536866816 bytes` y por último, nos muestra la cantidad total de procesos actuales `procs = 70`. Se puede observar que la mayoria de los archivos en la sección anterior son de tipo `S_IFREG` (archivos regulares), mientras que la minoría son directorios (`IFDIR`).

```
openat(AT_FDCWD, "send.py", O_RDONLY)   = 3
read(3, "import socket\n\nip='10.10.11.2'\np"..., 198) = 198
read(3, " ', data)\n\ns.close() \n", 4096) = 22

openat(AT_FDCWD, "/usr/lib/python2.7/socket.py", O_RDONLY) = 3
fstat(3, {st_mode=S_IFREG|0644, st_size=20615, ...}) = 0
openat(AT_FDCWD, "/usr/lib/python2.7/socket.pyc", O_RDONLY) = 4
fstat(4, {st_mode=S_IFREG|0644, st_size=16092, ...}) = 0
close(4)                                = 0

openat(AT_FDCWD, "/usr/lib/python2.7/functools.py", O_RDONLY) = 4
fstat(4, {st_mode=S_IFREG|0644, st_size=4806, ...}) = 0
openat(AT_FDCWD, "/usr/lib/python2.7/functools.pyc", O_RDONLY) = 5
fstat(5, {st_mode=S_IFREG|0644, st_size=6569, ...}) = 0
close(5)                                = 0
close(4)                                = 0

openat(AT_FDCWD, "/usr/lib/python2.7/lib-dynload/_ssl.x86_64-linux-gnu.so",
   O_RDONLY) = 4
fstat(4, {st_mode=S_IFREG|0644, st_size=110408, ...}) = 0
futex(0x7f6b621350c8, FUTEX_WAKE_PRIVATE, 2147483647) = 0
openat(AT_FDCWD, "/usr/lib/python2.7/lib-dynload/_ssl.x86_64-linux-gnu.so",
   O_RDONLY|O_CLOEXEC) = 5

mmap(NULL, 112672, PROT_READ, MAP_PRIVATE|MAP_DENYWRITE, 5, 0) = 0x7f6b61a38000
mprotect(0x7f6b61a40000, 61440, PROT_NONE) = 0
mmap(0x7f6b61a40000, 36864, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_FIXED|
   MAP_DENYWRITE, 5, 0x8000) = 0x7f6b61a40000
mmap(0x7f6b61a49000, 20480, PROT_READ, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE,
   5, 0x11000) = 0x7f6b61a49000
```

```
mmap(0x7f6b61a4f000, 20480, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|
   MAP_DENYWRITE, 5, 0x16000) = 0x7f6b61a4f000

openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 5
fstat(5, {st_mode=S_IFREG|0644, st_size=54201, ...}) = 0
mmap(NULL, 54201, PROT_READ, MAP_PRIVATE, 5, 0) = 0x7f6b62159000
openat(AT_FDCWD, "/usr/lib/x86_64-linux-gnu/libssl.so.1.1", O_RDONLY|
   O_CLOEXEC) = 5
mmap(NULL, 596272, PROT_READ, MAP_PRIVATE|MAP_DENYWRITE, 5, 0) = 0x7f6b619a6000
mprotect(0x7f6b619c3000, 425984, PROT_NONE) = 0
mmap(0x7f6b619c3000, 315392, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_FIXED|
   MAP_DENYWRITE, 5, 0x1d000) = 0x7f6b619c3000
mmap(0x7f6b61a10000, 106496, PROT_READ, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 5,
   0x6a000) = 0x7f6b61a10000
mmap(0x7f6b61a2b000, 53248, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|
   MAP_DENYWRITE, 5, 0x84000) = 0x7f6b61a2b000

openat(AT_FDCWD, "/usr/lib/x86_64-linux-gnu/libcrypto.so.1.1", O_RDONLY|O_CLOEXEC)
   = 5
fstat(5, {st_mode=S_IFREG|0644, st_size=3031904, ...}) = 0
mmap(NULL, 3051424, PROT_READ, MAP_PRIVATE|MAP_DENYWRITE, 5, 0) = 0x7f6b616bd000
mprotect(0x7f6b61742000, 2285568, PROT_NONE) = 0
mmap(0x7f6b61742000, 1695744, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_FIXED|
   MAP_DENYWRITE, 5, 0x85000) = 0x7f6b61742000
mmap(0x7f6b618e0000, 585728, PROT_READ, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE,
   5, 0x223000) = 0x7f6b618e0000
mmap(0x7f6b61970000, 204800, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|
   MAP_DENYWRITE, 5, 0x2b2000) = 0x7f6b61970000
mmap(0x7f6b619a2000, 16288, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|
   MAP_ANONYMOUS, -1, 0) = 0x7f6b619a2000

mprotect(0x7f6b61970000, 196608, PROT_READ) = 0
mprotect(0x7f6b61a2b000, 36864, PROT_READ) = 0
mprotect(0x7f6b61a4f000, 4096, PROT_READ) = 0
munmap(0x7f6b62159000, 54201)           = 0
mmap(NULL, 262144, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0)
   = 0x7f6b6167d000
socket(AF_INET, SOCK_STREAM, IPPROTO_IP) = 3
connect(3, {sa_family=AF_INET, sin_port=htons(5005), sin_addr=
   inet_addr("10.10.11.2")}, 16) = 0
sendto(3, "Trying out TCP socket", 21, 0, NULL, 0) = 21
recvfrom(3, "A C K, king", 1024, 0, NULL, NULL) = 11
close(3)                                = 0
rt_sigaction(SIGINT, {sa_handler=SIG_DFL, sa_mask=[], sa_flags=SA_RESTORER,
   sa_restorer=0x7f6b62148730}, {sa_handler=0x55f700f75ea0, sa_mask=[],
   sa_flags=SA_RESTORER, sa_restorer=0x7f6b62148730}, 8) = 0
exit_group(0)                           = ?
```

La sección anterior comienza abriendo el archivo *send.py* con el modo `O_RDONLY` que indica una petición para solo lectura. Después lee 198 bytes del archivo descriptor `send.py` en el buffer `import socket\n\nip='10.10.11.2`, el cual corresponde a la biblioteca *socket* y a la dirección IP de la máquina virtual 2. Después se realiza el mismo procedimiento con los archivos:

```
/usr/lib/python2.7/socket.py y socket.pyc
/usr/lib/python2.7/functools.py y functools.pyc
```

`/usr/lib/python2.7/lib-dynload/_ssl.x86_64-linux-gnu.so`

Posteriormente se utiliza `futex` que despierta el proceso en la dirección indicada en su primer argumento.

Las siguientes *system calls* usadas son para hacer mapeos y proteger regiones de memoria uilizando `mmap` y `mprotect`. El total de memoria mapeada es de *190496* bytes y *61440* bytes protegidos con la bandera `PROT_NONE` que indica que la sección de memoria no podrá ser accedida.

Las siguientes *system calls* siguen un procedimiento similar, donde se abre el archivo `/etc/ld.so.cache` con permisos de lectura y ejecución para después hacer un mapeo de la memoria de *54201* bytes. Después se abre el archivo `/usr/lib/x86_64-linux-gnu/libssl.so.1.1` con permisos de lectura y ejecución y un mapeo total de memoria de *787555* bytes y *425984* bytes protegidos usando la bandera `PROT_NONE`. Después se realiza el mismo procedimiento con el archivo `/usr/lib/x86_64-linux-gnu/libssl.so.1.1`, con un total de *5553984* bytes de mapeo y *2285568* bytes protegidos con `PROT_NONE`.

En las siguientes *system calls* se utiliza una serie de *mprotect* para proteger un total de *237568* bytes en tres regiones de memoria no consecutivas pero cercanas. Después se elimina el mapeo en la dirección `0x7f6b62159000` que fue usada previamente en el mapeo posterior a abrir el archivo `/etc/ld.so.cache` y se hace un mapeo de *262144* bytes en otra dirección. Después se utiliza la *system call* `socket` para la creación de un *endpoint*. En su primer argumento se hace uso de la bandera `domain = AF_INET` que especifica el uso del protocolo IPv4. El segundo argumento `type = SOCK_STREAM` proporciona un flujo de bytes secuenciados y bidireccionales basados en conexión ya que este argumento especifica la semántica de la comunicación. Después el socket debe ser conectado usando `connect`, el socket se conecta a la dirección descrita por `10.10.11.2` y el puerto `5005`. El siguiente paso es transmitir el mensaje utilizando *sendto*, el parámetro `buf = Trying out TCP socket` es el mensaje con una longitud `len = 21`. Luego, el socket recibe un mensaje con *recvfrom*, el mensaje es `buf = A C K` y se cierra el socket usando *close*.

Por ultimo, se usa `rt_sigaction` y `exit-group` para terminar el grupo de hilos del proceso de llamada. En los parámetros de `rt-sigaction` se encuentra `signum = SIGINT` que indica que se presiono `DELETE` o `Control-C` para finalizar un proceso.

4.1.2.2. Servidor

El programa usado para crear al servidor es (rec.py):

```
import socket

TCP_IP = ''
TCP_PORT = 5005
BUFFER_SIZE = 1024

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.bind((TCP_IP, TCP_PORT))
s.listen(1)

conn, addr = s.accept()
print "Connection address: ", addr
while 1:
   data = conn.recv(BUFFER_SIZE)
   if not data: break
   print "received data: ", data
   conn.send('ACK')
conn.close()
```

Al ejecutar el programa anterior se obtienen las siguientes *system calls*:

```
execve("/usr/bin/python", ["python", "rec.py"], 0x7ffdfc15b508 /* 19 vars */) = 0
brk(NULL)  = 0x558fd42c7000
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or directory)
openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
fstat(3, {st_mode=S_IFREG|0644, st_size=55559, ...}) = 0
mmap(NULL, 55559, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7f1534e18000
```

La primer *system call* que se observa en la primer sección es `execve`, que indica que el entorno del nuevo programa será `0x7ffdfc15b508`. Después hace una solicitud de información de la ubicación en donde termina su memoria acumulada usando `brk`. Luego mediante `access` se comprueba la inexistencia de `/etc/ld.so.preload` al igual que en el caso del cliente. Después se utiliza `openat` y `fstat` para abrir `/etc/ld.so.cache` en modo lectura y ejecución, también para obtener información del archivo. Por último en esta sección, se hace un mapeo privado en `0x7f1534e18000` de 55559 bytes.

A continuación se utilizan las *system calls* `openat`, `fstat`, `mprotect` y `mmap` y los archivos que se utilizaron en el cliente, los únicos parámetros que cambian son las direcciones, por lo que esta parte del análisis se puede descartar.

```
set_tid_address(0x7f1534884a10)  = 621
set_robust_list(0x7f1534884a20, 24) = 0
rt_sigaction(SIGRTMIN, {sa_handler=0x7f1534dfb6b0, sa_mask=[], sa_flags=
   SA_RESTORER|SA_SIGINFO, sa_restorer=0x7f1534e07730}, NULL, 8) = 0
rt_sigaction(SIGRT_1, {sa_handler=0x7f1534dfb740, sa_mask=[], sa_flags=
   SA_RESTORER|SA_RESTART|SA_SIGINFO, sa_restorer=0x7f1534e07730}, NULL,
   8) = 0
rt_sigprocmask(SIG_UNBLOCK, [RTMIN RT_1], NULL, 8) = 0
prlimit64(0, RLIMIT_STACK, NULL, {rlim_cur=8192*1024, rlim_max=
   RLIM64_INFINITY}) = 0
```

Al igual que en el programa `send.py`, el servidor `rec.py` utiliza las mismas *system calls*. Estas se ilustran en la sección anterior. Primero se usa `set_tid_address` para establecer un valor `clear_child_tid` al hilo 0x7f1534884a10 con ID 621. La siguiente *system call* `set_roboust_list` apunta al head de una lista robusta de futex. Después se utilizan dos *system calls rt-sigaction* para determinar el rango de señales en tiempo real y para realizar cambios de credenciales a lo largo del proceso, respectivamente. En la primera *rt-sigaction* se hace referencia a la acción asociada con `signum = SIGRTMIN`, `sa_handler = 0x7f1534dfb6b0` y las banderas `SA_RESTORER/SA_SIGINFO` que indican que `sa_restorer` tiene la dirección de la *"signal trampoline"* y permiten al manejador de la señal tomar tres parámetros en vez de solo 1. En la segunda `rt_sigaction` se hace uso de las banderas `SA_RESTORER/SA_SIGINFO` y también de `SA_RESTART` que proporciona un comportamiento compatible con señales BSD. Por último, ambas *system calls* usan la misma *"signal trampoline"*.

Después se usa la *system call rt-sigprocmask* para cambiar la mascara de la señal de hilo de llamada con `SIG_UNBLOCK` para eliminar las señales `RTMIN RT_1` del conjunto de señales bloqueadas. La *system call* `prlimit` establece los limites de los recursos del proceso, `pid = 0` indica que esta llamada se aplicará al proceso de llamada actual, `resource = RLIMIT_STACK` indica un valor máximo del tamaño de la pila del proceso y cuando este sea alcanzado se enviará una señal `SIGSEGV`. Los valores en su último argumento indican los límites, el valor `rlim_max = RLIM64_INFINITY` indica que no hay límites.

Después se utiliza una serie de *system calls* `stat`, `rt_sigaction` y `openat` de la misma forma que en el caso del lado del cliente. En ambos casos se aplicaron a los mismos archivos y señales y la única diferencia se encuentra en las direcciones de memoria, la siguiente *system call* y el siguiente archivo se muestran a continuación:

```
sysinfo({uptime=481, loads=[11072, 4896, 1856], totalram=514121728,
   freeram=228864000, sharedram=827392, bufferram=10067968,
   totalswap=536866816, freeswap=536866816, procs=73, totalhigh=0,
```

```
   freehigh=0, mem_unit=1}) = 0

pathname                                        dirfd           st_mode

/root/rec.py                                    AT_FDCWD        S_IFREG
```

La *system call sysinfo* de la sección anterior indica que el total de memoria ram utilizable es de `514121728 bytes`, la memoria ram libre es de `228864000 bytes`, la memoria ram compartida es de `827392 bytes`, y el total de memoria ram en buffers es de `10067968 bytes`. También indica que el total de espacio de swap es de `536866816 bytes`, el espacio de swap libre es de `536866816 bytes` y por último, muestra la cantidad total de procesos actuales `procs = 73`. El total de memoria ram utilizable y los espacios de swap mantienen la misma cantidad que en el estudio del cliente, por el contrario, los demás parámetros son distintos, siendo el más relevante el número de procesos.

Del lado del servidor, al igual que en el cliente, se utilizan las mismas *system calls* y se abren los mismos archivos de la última sección. La única diferencia ocurre en los parámetros de la siguientes *system calls*:

```
openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 5
fstat(5, {st_mode=S_IFREG|0644, st_size=55559, ...}) = 0
mmap(NULL, 55559, PROT_READ, MAP_PRIVATE, 5, 0) = 0x7f1534e18000

munmap(0x7f1534e18000, 55559) = 0
```

Al igual que en el cliente, en el servidor también se usa la *system call* `openat` para abrir en modo lectura y ejecución el archivo `/etc/ld.so.cache`, pero a diferencia del lado del cliente, aquí se hace un mapeo más grande (`55559 bytes`), por lo que la *system call* `munmap` también usará el mismo parámetro para la cantidad de bytes.

A continuación se muestra la última sección de *system calls*:

```
socket(AF_INET, SOCK_STREAM, IPPROTO_IP) = 3
bind(3, {sa_family=AF_INET, sin_port=htons(5005), sin_addr=inet_addr
   ("0.0.0.0")}, 16) = 0
listen(3, 1)  = 0
accept(3, {sa_family=AF_INET, sin_port=htons(32890), sin_addr=inet_addr
   ("10.10.10.2")}, [16]) = 4
recvfrom(4, "Trying out TCP socket", 1024, 0, NULL, NULL) = 21
write(1, "received data: Trying out TCP so"..., 37received data: Trying out
    TCP socket) = 37
sendto(4, "A C K", 11, 0, NULL, 0) = 11
recvfrom(4, "", 1024, 0, NULL, NULL)    = 0
rt_sigaction(SIGINT, {sa_handler=SIG_DFL, sa_mask=[], sa_flags=SA_RESTORER,
    sa_restorer=0x7f1534e07730}, {sa_handler=0x558fd3d0eea0, sa_mask=[],
    sa_flags=SA_RESTORER, sa_restorer=0x7f1534e07730}, 8) = 0
exit_group(0)                           = ?
```

En la sección anterior se hace uso de `socket` para crear el endpoint para la comunicación con el cliente con las mismas características que se usaron para crear el endpoint en el cliente. Después se utiliza la *system call* `bind` para asignar una dirección temporal al socket `0.0.0.0`. Después se usa la *system call* `listen` para que el socket creado acepte solicitudes de conexión mediante la siguiente *system call* `accept`, la cual conecta el socket con la dirección `10.10.10.2` correspondiente a la máquina 1. El siguiente paso es recibir un mensaje de la conexión usando `recvfrom`. Después se transmite un mensaje usando `sendto` y se recibe un mensaje vació con `recvfrom`.

Por último se usa `rt_sigaction` y `exit_group` para terminar el grupo de hilos del proceso. Los argumentos de `rt_sigaction` son los mismos que se usaron en el lado del cliente excepto por las direcciones de memoria.

Capítulo 5

Ataque DoS y análisis de vulnerabilidades

Uno de los ataques más frecuentes en las redes definidas por software es la negación de servicio o por sus siglas en inglés *Denial of Service* (*DoS*). A grandes rasgos, este ataque afecta la estabilidad de una red, ya que consiste en consumir la mayor parte de los recursos disponibles de un servidor por medio de solicitudes fuera de proporción por parte de un *guest* infectado o subyugado por un tercero a nivel de imposibilitarlo para prestar servicio; de ahí su nombre de negación de servicio.

A continuación se presenta cómo se observaría un ataque DoS en una arquitectura de red definida por software y un análisis de las vulnerabilidades que se podrían explotar a través de los parámetros de las llamadas al sistema ejecutadas por un *guest*, a través de las cuales se podría propiciar una negación de servicio en el Dom0.

5.1. Implementación de un ataque DoS

Para la implementación de este ataque, se exploraron varias herramientas preexistentes para la puesta en marcha de un DoS. Una de estas herramientas es conocida como hInjector [16], sin embargo ha quedado obsoleta para las arquitecturas actuales del servicio de paravirtualización Xen. Dada esta circunstancia, se utilizó la herramienta de testeo de penetración de red llamada *hping3*. Esta herramienta permite hacer una explotación excesiva de los recursos del Dom0 a través del uso de mensajes de sincronía con el *host* a modo de *Flooding*.

A través de la herramienta *hping3* se realizaron múltiples pruebas para generar estadísticas del uso del CPU y de esta manera determinar el punto en el que el *Dom0* entra en estado de negación de servicio. Para poder generar dichas estadísticas, se hizo uso de la herramienta de monitorización del sistema Linux que lleva por nombre *glances*, la cual muestra en tiempo real los procesos en ejecución y el uso tanto de CPU como de memoria que cada uno consume; así como las interfaces de red y el tráfico que pasa a través de ellas. También esta herramienta permite observar en tiempo real el consumo de memoria y CPU de cada una de las máquinas virtuales implementadas en el servidor Xen mediante la paravirtualización.

En segundo lugar, mientras se ejecuta el ataque se extraen las llamadas al sistema y se hace una análisis detallado. Este análisis se enfoca en aquellas llamadas al sistema cuyos parámetros de solicitud de recursos al Dom0 conllevan una clara vulnerabilidad del sistema en caso de cualquier superación en el límite de memoria del *buffer* o cualquier búsqueda en índices inexistentes a nivel memoria.

Finalmente, se conjuntan estos dos escenarios para dar lugar a un análisis del ataque *DoS* llevado a cabo con la herramienta *hping3* con el fin de estudiar las vulnerabilidades del Dom0 y que den lugar a la posibilidad de que un *guest* provoque el decaimiento de una red virtualizada.

La herramienta de análisis de penetración *hping3* trae consigo diversas opciones de ejecución de acuerdo al tipo de ataque que se desee realizar; para los propósitos experimentales

del presente trabajo, se optó por ejecutar el siguiente comando:

```
hping3 192.168.1.89 -q -n -d 1200 -S -p 80 --flood --rand-source
```

La dirección ip inmediata al comando *hping3* corresponde a la dirección del puente xenbr0 asignado al puerto de salida ethernet de la computadora física; hacia el cual se dirige el ataque. La opción `-q` se refiere a que la ejecución del proceso se hará de manera silenciosa, por lo que todas las respuestas del host atacado no se muestran en pantalla. La opción `-n` muestra el número de paquetes lanzados durante el tiempo de ejecución. La opción `-d` se utiliza para determinar el tamaño de los paquetes de información lanzados; en este caso de 1200 bytes. La opción `-p` indica el puerto al que el ataque será dirigido. `--flood` se utiliza para hacer que el envío de los paquetes creados sea lo más rápido posible. Finalmente, `--rand-source` es una forma de que la herramienta enmascare los paquetes del ataque con direcciones de origen aleatorias. Se puede encontrar mayor información relacionada a esta herramienta en [25].

Con el fin de cuantificar el consumo de recursos que representa el sistema víctima, se filtraron los datos proporcionados por la herramienta *glances* descrita anteriormente. Cabe mencionar que en esta herramienta resaltan dos procesos que delatan un cambio en la actividad normal del servidor: `ksoftirqd` y `vif1.0-q0-deall`. El primero de ellos corresponde a un hilo de kernel por cada CPU que corre cada vez que una máquina se encuentra bajo una carga alta de interrupciones por software [26].

Las interrupciones son una parte esencial en el funcionamiento de las computadoras en el nivel más bajo de procesamiento, éstas alertas detienen la ejecución de un proceso para ejecutar el proceso que haya desencadenado la interrupción. De esta manera, `ksoftirqd` corresponde a un *daemon* que manipula las interrupciones por software. Asimismo, para una mejor comprensión de los efectos a nivel kernel, se utilizó la herramienta `irqtop` que muestra en tiempo real los cambios en el archivo */proc/interrupts*, el cual almacena una bitácora de las interrupciones a nivel *kernel*.

Es importante mencionar que con la herramienta `topirqd`, se pueden categorizar todas las interrupciones generadas a partir del proceso `ksoftirqd` en tres clases durante el ataque *DoS* en marcha:

1. *Rescheduling interrupts*: son interrupciones generalizadas que representan delegaciones de un núcleo de CPU hacia otro pidiéndole que realice el proceso de *scheduling* para distribuir la carga de procesamiento.

2. *timer*: las interrupciones de tipo *timer* incurren en la modificación de la frecuencia con la que el procesador realiza el conteo de interrupciones al kernel de forma global al realizar *rescheduling* de ciertos procesos.

3. *Tx*: corresponde a las interrupciones generadas en la interfaz virtual creada entre el host y el conmutador virtual al realizar un intercambio de información.

En segundo lugar, `vif1.0-q0-deall` corresponde a uno de los hilos que corren al iniciar una máquina virtual alojada en el sistema de paravirtualización Xen [27].

Por otro lado, la herramienta `xl top` forma parte de las utilerías contenidas en Xen, la cual proporciona lecturas de consumo de memoria y CPU por parte de todas las máquinas virtuales en Xen.

La figura 5.1 muestra un diagrama de bloques de cómo se realizó el ataque de manera general.

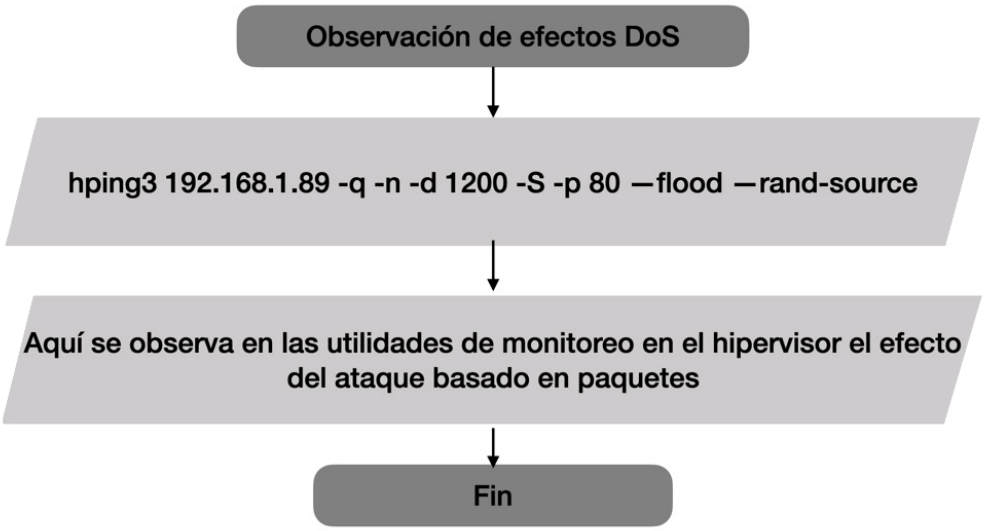

Figura 5.1: Diagrama del proceso de monitorización de los efectos de un DoS.

5.1.1. Resultados del DoS

La figura 5.2 muestra una cronología del ataque con paquetes de información de 1200 bytes en cuanto al consumo del CPU correspondiente a la máquina 1. Se puede observar en la gráfica obtenida por la herramienta *glances* que el *deamon* `ksoftirqd` llega al 100 % durante el periodo de ataque *DoS* de un minuto.

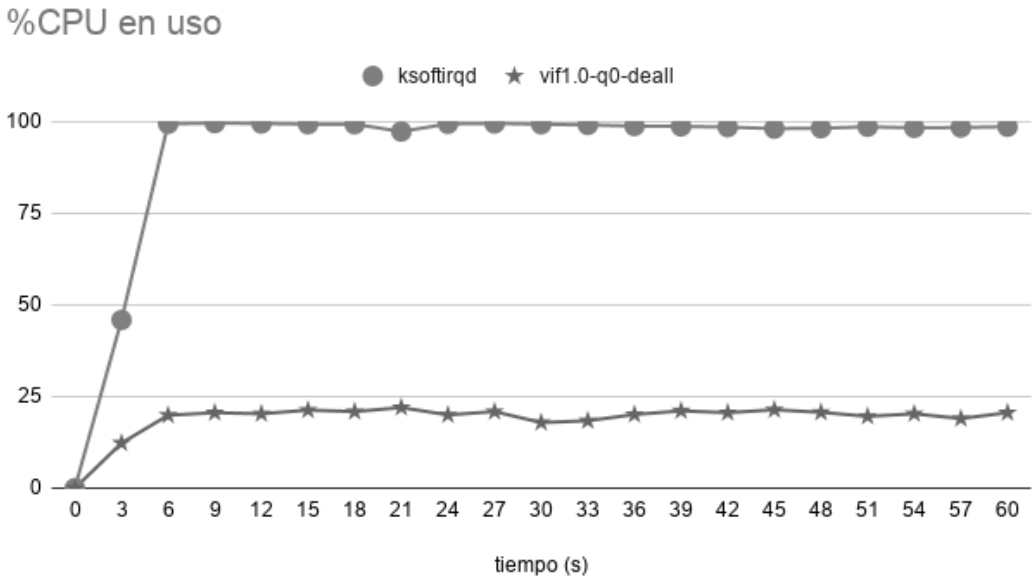

Figura 5.2: Porcentaje de uso de CPU con paquetes de 1200 bytes.

Por otro lado mediante la herramienta `xl top` se puede observar cómo la sobrecarga del CPU sobrepasa el 100 % como se observa en la figura 5.3.

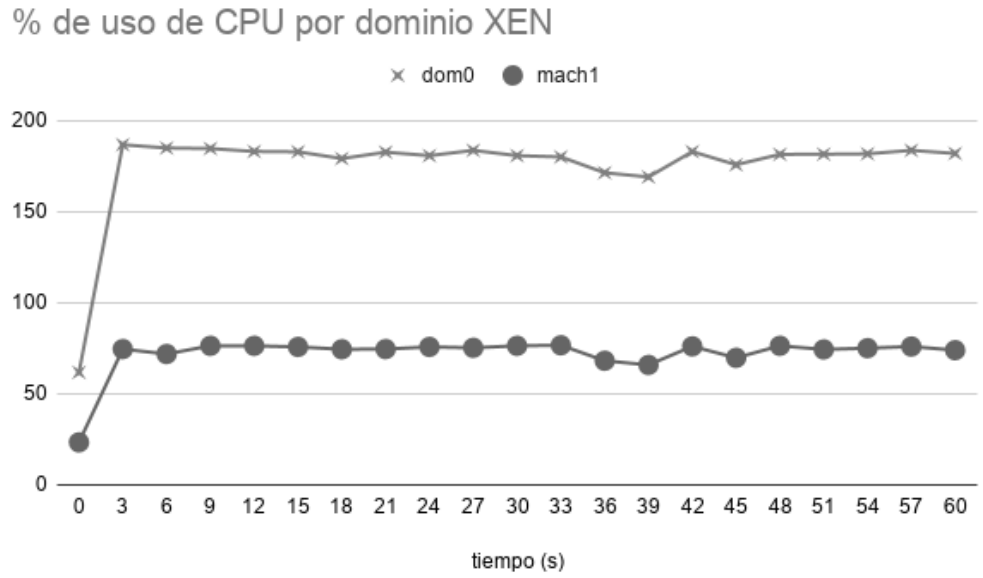

Figura 5.3: Porcentaje de uso de CPU por dominio Xen con paquetes de 1200 bytes.

Es importante aclarar que en las computadoras multi-núcleo la monitorización mediante el comando `top` muestra la suma del porcentaje en cada núcleo [28] (para esta tesis son cuatro núcleos). Por lo que la respuesta a nivel núcleo se puede obtener mediante el comando `htop`, que es una versión mejorada del comando `top`. Esto se puede observar en la figura 5.4.

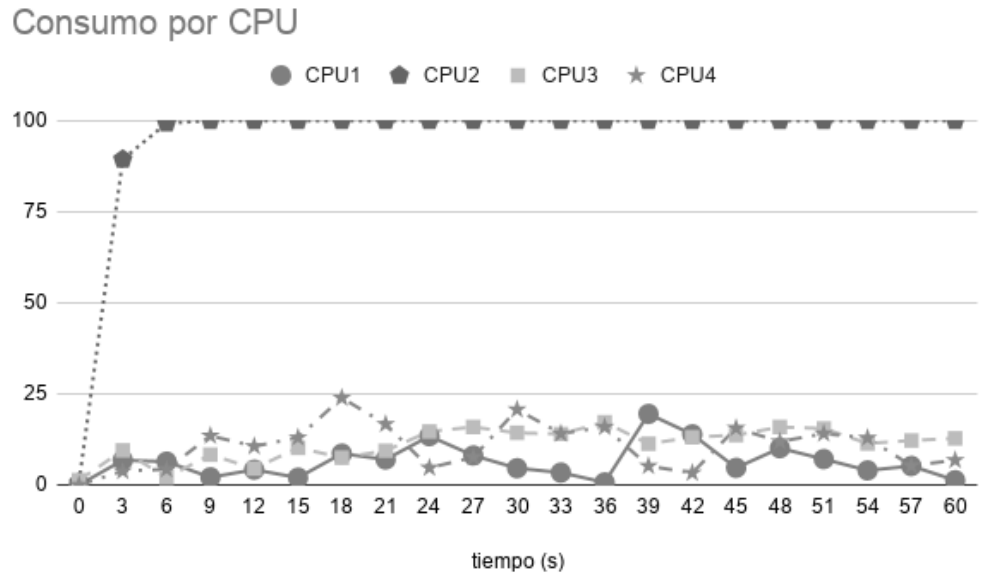

Figura 5.4: Porcentaje de uso de CPU por núcleo con paquetes de 1200 bytes.

Las figuras 5.5, 5.6 y 5.7 muestran los mismos escenarios anteriores pero con un tamaño de paquete diez veces mayor.

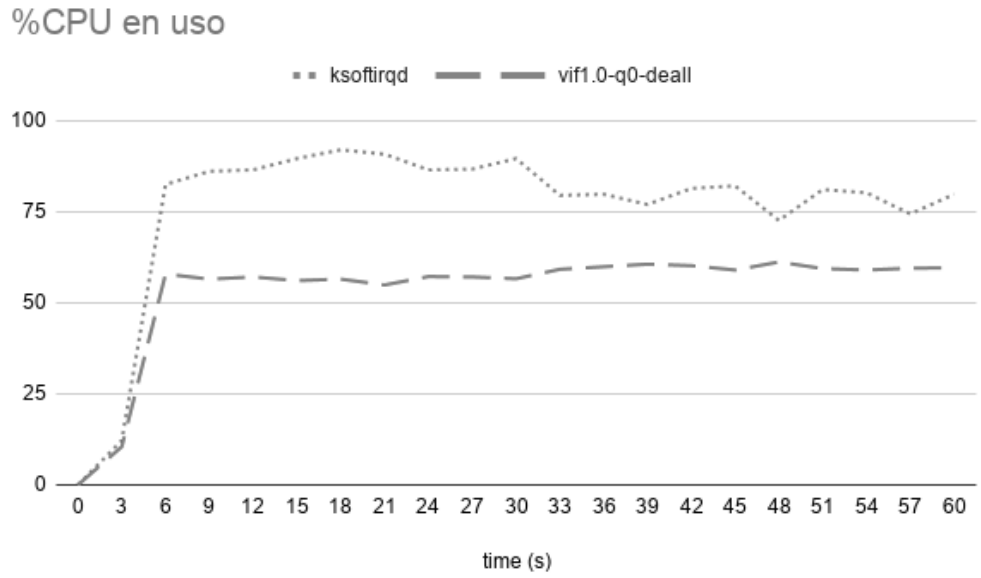

Figura 5.5: Porcentaje de uso de CPU con paquetes de 12000 bytes.

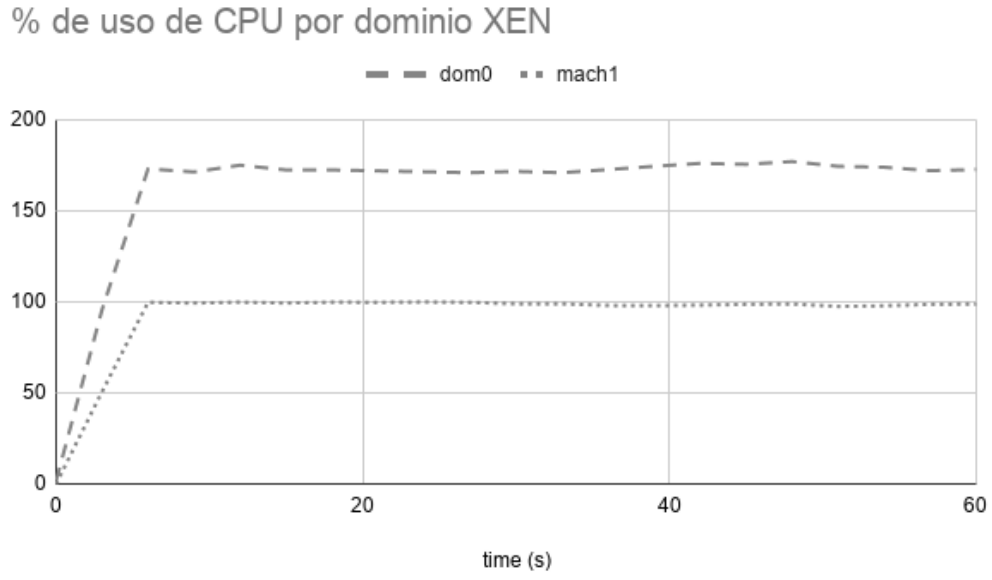

Figura 5.6: Porcentaje de uso de CPU por dominio Xen con paquetes de 12000 bytes.

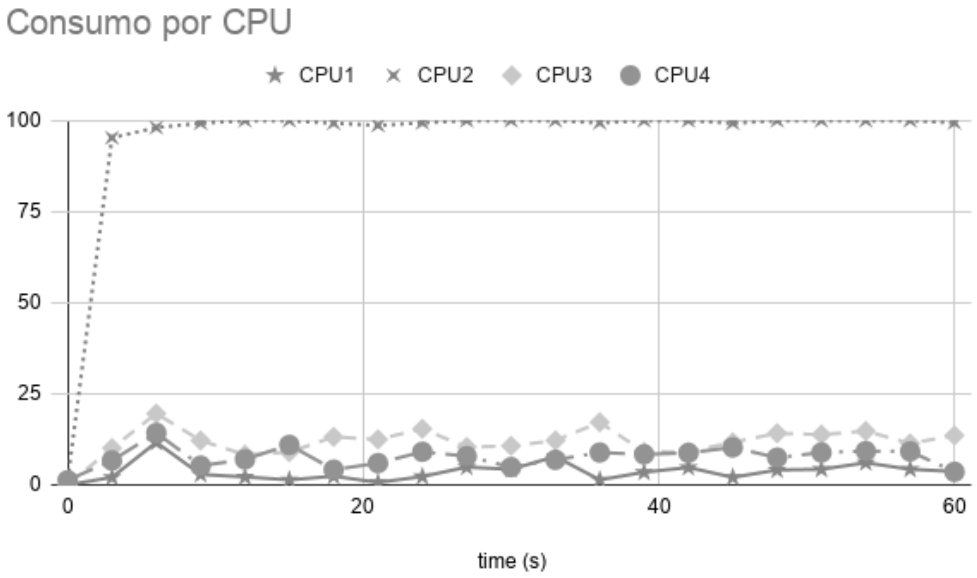

Figura 5.7: Porcentaje de uso de CPU por núcleo con paquetes de 12000 bytes.

Al hacer la comparación de estas últimas gráficas con respecto a las previas obtenidas, se puede observar que:

1. Existe poca afectación en el consumo del CPU cuando se varía el tamaño del paquete.
2. El procesador cuenta por sí mismo con la capacidad de delegar algunos procesos o *threads* en ejecución a núcleos que cuenten con bajo porcentaje de utilización al momento de verse saturados por uno o un conjunto de procesos en particular.

5.1.2. System calls hping3

A continuación se analizan las *system calls* obtenidas mediante el ataque *Do*S utilizando el formato del capitulo 4. El ataque se realizó desde una máquina virtual al Dom0 mediante el siguiente comando en la terminal de Linux:

```
hping3 192.168.1.89 -q -n -d 1200 -S -p 80 --flood --rand-source
```

La figura 5.8 muestra un diagrama de cómo se obtuvieron las llamadas al sistema durante el ataque DoS.

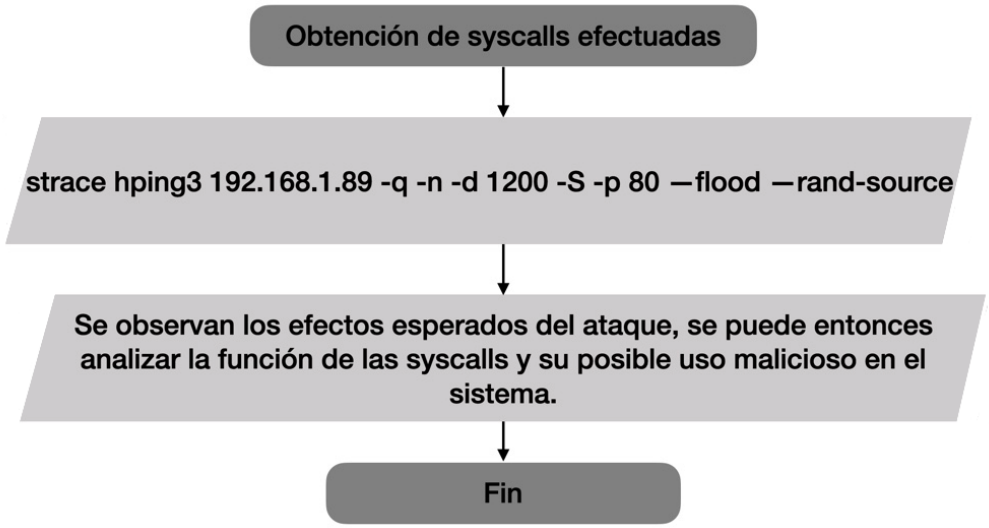

Figura 5.8: Diagrama de extracción de las *syscalls* durante el ataque.

Como se mencionó con anterioridad, la primer *system call* en ser ejeuctada es *execve*:

```
execve("/usr/sbin/hping3", ["hping3", "--rand-source", "--flood",
   "192.168.1.89"], [/* 13 vars */]) = 0
brk(NULL) = 0x5641e1fa9000
open("/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
fstat(3, {st_mode=S_IFREG|0644, st_size=20962, ...}) = 0
mmap(NULL, 20962, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7fce2954f000
close(3) = 0
```

la cual ejecuta el programa asociado con el parámetro `pathname = /usr/sbin/hping3`, después el primer elemento del *array* indica el nombre del archivo asociado a la ejecución (las demás posiciones del *array* son parámetros complementarios de la ejecución del programa). Posteriormente, se hace una solicitud de información de la ubicación en donde termina la memoria acumulada mediante la *system call brk*. Las siguientes *system calls* utilizadas son *openat* y *fstat* para abrir el archivo `/etc/ld.so.cache` para su lectura y ejecución. Para terminar esta sección se hace un mapeo privado en la dirección `0x7fce2954f000` de 20962 bytes.

La siguiente sección muestra las *system calls* posteriores, las cuales se estructuran bajo el mismo formato usado en el capítulo 4.

```
open("/usr/lib/x86_64-linux-gnu/libtcl8.6.so", O_RDONLY|O_CLOEXEC) = 3
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0)
   = 0x7fce2954d000
mmap(NULL, 2365344, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0)
   = 0x7fce290f0000
mprotect(0x7fce29130000, 2093056, PROT_NONE) = 0
```

```
mmap(0x7fce2932f000, 12288, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|
   MAP_DENYWRITE, 3,0x3f000) = 0x7fce2932f000
close(3)
```

La primera *system call* de esta sección es `open` que abre el archivo descrito por su primer parámetro `pathname` con la bandera `O_CLOEXEC` que evita operaciones adicionales, después se hacen dos mapeos de memoria, el primero es de 8192 bytes en modo lectura y escritura, el segundo es de 2365344 en modo lectura y ejecución. La siguiente *system call* protege una sección de memoria de 2093956 bytes, y por último se hace un mapeo de 12288 bytes en modo de lectura y escritura.

Los siguientes bloques de *system calls* solo se mostrarán y no se explicarán dado su similitud con los procesos analizados en el Capítulo 4.

```
System call                 Argumento(s)

openat(*pathname)               .
mprotect(*addr, len)            .
mmap(*addr, length)             .

Bytes protegidos:
Logitud total del mapeo:
```

Segunda sección:

```
System call                 Argumento(s)

openat(*pathname)           /lib/x86_64-linux-gnu/libm.so.6
mmap(*addr, length)         (0x7fce28a35000, 3158248)
mprotect(*addr, len)        (0x7fce28b38000, 2093056)
mmap(*addr, length)         (0x7fce28d37000, 8192)

Bytes protegidos:           2093056
Logitud total del mapeo:    3166440
```

Tercer sección:

```
System call                 Argumento(s)

openat(*pathname)           /lib/x86_64-linux-gnu/libpthread.so.0
mmap(*addr, length)         (0x7fce28818000, 2212936)
mprotect(*addr, len)        (0x7fce28830000, 2093056)
mmap(*addr, length)         (0x7fce28a2f000, 8192)
mmap(*addr, length)         (0x7fce28a31000, 13384)

Bytes protegidos:           2093056
Logitud total del mapeo:    2234512
```

Cuarta sección:

```
System call                 Argumento(s)

openat(*pathname)           /lib/x86_64-linux-gnu/libc.so.6
mmap(*addr, length)         (0x7fce28479000, 3795296)
mprotect(*addr, len)        (0x7fce2860e000, 2097152)
mmap(*addr, length)         (0x7fce2880e000, 24576)
mmap(*addr, length)         (0x7fce28814000, 14688)

Bytes protegidos:           2097152
Logitud total del mapeo:    3834560
```

Quinta sección:

```
System call                 Argumento(s)

openat(*pathname)           /lib/x86_64-linux-gnu/libdl.so.2
mmap(*addr, length)         (0x7fce28275000, 2109680)
mprotect(*addr, len)        (0x7fce28278000, 2093056)
mmap(*addr, length)         (0x7fce28477000, 8192)

Bytes protegidos:           2093056
Logitud total del mapeo:    2117872
```

Sexta sección:

```
System call                 Argumento(s)

openat(*pathname)           /lib/x86_64-linux-gnu/libz.so.1
mmap(*addr, length)         (0x7fce2954b000, 8192)
mmap(*addr, length)         (0x7fce2805b000, 2200072)
mprotect(*addr, len)        (0x7fce28074000, 2093056)
mmap(*addr, length)         (0x7fce28273000, 8192)

Bytes protegidos:           2093056
Logitud total del mapeo:    2216456
```

El siguiente bloque de *system calls* realiza una mapeo en `0x7fce29549000` de 8192 bytes, luego se protegen diferentes secciones de memoria y finalmente se utiliza `munmap` para eliminar el mapeo en `0x7fce2954f000` que es la dirección de memoria correspondiente al primer mapeo de todo el proceso:

```
mprotect(0x7fce2880e000, 16384, PROT_READ) = 0
mprotect(0x7fce28273000, 4096, PROT_READ) = 0
mprotect(0x7fce28477000, 4096, PROT_READ) = 0
mprotect(0x7fce28a2f000, 4096, PROT_READ) = 0
mprotect(0x7fce28d37000, 4096, PROT_READ) = 0
mprotect(0x7fce290df000, 57344, PROT_READ) = 0
mprotect(0x7fce2932f000, 8192, PROT_READ) = 0
mprotect(0x5641e02ac000, 4096, PROT_READ) = 0
mprotect(0x7fce29555000, 4096, PROT_READ) = 0

Bytes protegidos:   106496

munmap(0x7fce2954f000, 20962) = 0
```

Las siguientes *system calls* son `set_id_address` y `set_robust_list` que sirven para establecer un valor *"clear child tid"* al hilo en la dirección `0x7fce295499d0` cuyo ID es el 448, y para obtener un apuntador al head de la lista robusta de futex en la dirección `0x7fce295499e0`. Posteriormente, se hace uso de dos llamadas a `rt sigaction`. En la primera llamada, se establece el parámetro `signum = SIGRTMIN`, el cual indica que se trata de una señal usada para determinar el rango de señales en tiempo real, mientras su parámetro `sa_handler = 0x7fce2881dbd0` indica la dirección asociada con el parámetro `signum`, y se utilizan las banderas `SA_RESTORER` y `SA_SIGINFO` que indican que el campo `sa_restorer` contiene la dirección de la *"signal trampoline"* que permiten al manejador de la señal tomar tres parámetros. En la segunda `rt sigaction`, el parámetro `signum = SIGRT_1` indica que la señal es usada para realizar cambios de credenciales a lo largo del proceso, adicionalmente a las banderas utilizadas en la llamada anterior, en ésta se utiliza la bandera `SA_RESTART` que proporciona un comportamiento compatible con señales BSD. La siguiente *system call* es `rt_sigprocmask` que es usada para cambiar la mascara de la señal del hilo de

la llamada. Su primer parámetro `how = SIG_UNBLOCK` indica que las señales del parámetro `set = [RTMIN RT_1]` son eliminadas por un conjunto de señales bloqueadas. Por último, se utiliza la *system call* `getrlimit`, la cual obtiene los límites de los recursos desde su estructura `rlimit`. Su primer parámetro `resource = RLIMIT_STACK` indica el máximo tamaño de la pila del proceso y una vez llegado a ese límite, se genera una señal `SIGSEGV`. La estructura `rlimit` tiene como valores `{rlim_cur=8192*1024, rlim_max=RLIM64_INFINITY}`, donde el primer argumento de la estructura indica el valor que el *kernel* impone para el recurso correspondiente, mientras el segundo argumento funciona como un tope para *soft limit*, en este caso el valor `RLIM_INFINITY` indica que el recurso no tiene límites.

```
set_tid_address(0x7fce295499d0)         = 448
set_robust_list(0x7fce295499e0, 24)     = 0
rt_sigaction(SIGRTMIN, {sa_handler=0x7fce2881dbd0, sa_mask=[],
   sa_flags=SA_RESTORER|SA_SIGINFO, sa_restorer=0x7fce288290e0},
   NULL,8) = 0
rt_sigaction(SIGRT_1, {sa_handler=0x7fce2881dc60, sa_mask=[],
   sa_flags=SA_RESTORER|SA_RESTART|SA_SIGINFO,sa_restorer =
   0x7fce288290e0}, NULL, 8) = 0
rt_sigprocmask(SIG_UNBLOCK, [RTMIN RT_1], NULL, 8) = 0
getrlimit(RLIMIT_STACK, {rlim_cur=8192*1024, rlim_max=RLIM64_INFINITY})
   = 0
```

La siguiente sección de *system calls* utiliza `socket`, `setsockopt`, `connect` y `getsockname`. La primera *system call* se usa para crear el *socket* con `domain = AF_INET`, lo cual indica que se usará el protocolo IPv4, el parámetro `type = SOCK_DGRAM` indica la semántica de comunicación (en este caso el *socket* soportara datagramas). La siguiente *system call* tiene como primer parámetro la referencia al *socket* creado, luego los siguientes parámetros son `level = SOL_SOCKET` y `optname = SO_BROADCAST` que permite al *socket* enviar mensajes broadcast. El siguiente paso es conectar el *socket*, para eso se usa la *system call* `connect`, que conecta al *socket* con el puerto `11111` y la dirección `192.168.1.89`. Finalmente se obtiene información del *socket* usando `getsockname` en donde podemos obtener el puerto y dirección del *socket*.

```
socket(AF_INET, SOCK_DGRAM, IPPROTO_IP) = 3
setsockopt(3, SOL_SOCKET, SO_BROADCAST, [1], 4) = 0
connect(3, {sa_family=AF_INET, sin_port=htons(11111), sin_addr =
   inet_addr("192.168.1.89")}, 16) = 0
getsockname(3, {sa_family=AF_INET, sin_port=htons(55408), sin_addr=
   inet_addr("10.10.10.2")}, [16]) = 0
close(3)   = 0
```

En la siguiente sección de *system calls* se vuelve a utilizar `socket` para crear un *socket*, ahora de tipo *raw*. En seguida se utiliza dos veces `setsockopt`, donde se manipulan las opciones del nuevo *socket* creado, primero se utiliza `optname = SO_BROADCAST` que permite al *socket* enviar mensajes *broadcast*, después en la segunda *system call* `setsocketopt` se habilita `optname = IP_HDRINCL`.

Nota: Los *sockets raw* permiten la implementación del protocolo IPv4 en el espacio de usuario, un *socket raw* recibe y envía datagramas *raw* sin incluir `link level headers`. La capa IPv4 genera el header IP cuando envía un paquete, a menos que la opción del *socket* `IP_HDRINCL` sea habilitada. Cuando esta opción es habilitada, el paquete debe contener un header IP y es capaz de enviar cualquier protocolo IP que esté especificado en el header.

Después se crea otro *socket* de tipo *raw*, pero a diferencia del anterior, este *socket* tiene como parámetro `domain = AF_PACKET`, lo que indica que el dominio de comunicación se compone de una interfaz de paquete de bajo nivel. Luego se utiliza la *system call* `bind` para asignar los parámetros al *socket*.

Por último, en esta sección se utilizan las *system calls* `getsockopt` y `setsockop` para leer y asignar opciones del *socket*, ya que el *socket* tiene un nivel `SOL_SOCKET` y `SOL_PACKET`.

```
socket(AF_INET, SOCK_RAW, IPPROTO_RAW)  = 3
setsockopt(3, SOL_SOCKET, SO_BROADCAST, [1], 4) = 0
setsockopt(3, SOL_IP, IP_HDRINCL, [1], 4) = 0

socket(AF_PACKET, SOCK_RAW, 768)        = 4
bind(4, {sa_family=AF_PACKET, sll_protocol=htons(ETH_P_ALL),
   sll_ifindex = if_nametoindex("eth0"),sll_hatype=ARPHRD_NETROM,
   sll_pkttype=PACKET_HOST, sll_halen=0}, 20) = 0

getsockopt(4, SOL_SOCKET, SO_ERROR, [0], [4]) = 0
setsockopt(4, SOL_PACKET, PACKET_AUXDATA, [1], 4) = 0
setsockopt(4, SOL_PACKET, PACKET_RX_RING, 0x7ffea3744560, 28) = 0
```

La siguiente sección comienza con un mapeo en la dirección `0x7fce27e5b000` de 2097152 bytes de lectura y escritura que también será compartido. Después se usa la *system call* `uname` para obtener información del sistema, lo que nos indica que el sistema es Linux y estamos en la *mach1*. Después se utiliza una serie de *system calls* `rt_sigaction`, esta *system call* cambia la acción tomada por el proceso al recibir la señal en `signum`. La primer *system call* `rt_sigaction` utiliza la señal `SIGTSTP` que indica la acción `stop` desde terminal, la siguiente *system call* utiliza la señal `SIGINT` que indica una interrupción desde el tecaldo, por último la siguiente señal utilizada es `SIGTERM` que indica el fin de una señal. Todas estas *system calls* tienen como `sa_handler` una dirección de memoria que es un apuntador a una función de manejo de señales.

```
mmap(NULL, 2097152, PROT_READ|PROT_WRITE, MAP_SHARED, 4, 0)=0x7fce27e5b000
uname({sysname="Linux", nodename="mach1", ...}) = 0

rt_sigaction(SIGTSTP, {sa_handler=0x5641e008ff10, sa_mask=[], sa_flags=
   SA_RESTORER|SA_RESTART, sa_restorer=0x7fce288290e0}, {sa_handler=
   SIG_DFL, sa_mask=[], sa_flags=0}, 8) = 0
rt_sigaction(SIGINT, {sa_handler=0x5641e0092740, sa_mask=[], sa_flags=
   SA_RESTORER|SA_RESTART, sa_restorer=0x7fce288290e0}, {sa_handler=
   SIG_DFL, sa_mask=[], sa_flags=0}, 8) = 0
rt_sigaction(SIGTERM, {sa_handler=0x5641e0092740, sa_mask=[], sa_flags=
   SA_RESTORER|SA_RESTART, sa_restorer=0x7fce288290e0}, {sa_handler=
   SIG_DFL, sa_mask=[], sa_flags=0}, 8) = 0
getpid()   = 448
kill(448, SIGALRM)  = 0
```

Las siguientes *system calls* se repiten indefinidamente y son las que se usan para realizar el ataque.

```
sendto(3, "E\0\0(\270\223\0\0@\6\0\0\347\t\362[\300\250\1Y\10\233\0\0J\316
   \rE:Fg\10"..., 40,0, {sa_family=AF_INET, sin_port=htons(11111), sin_addr=
   inet_addr("192.168.1.89")}, 16) = 40
rt_sigaction(SIGALRM, {sa_handler=0x5641e0094100, sa_mask=[], sa_flags=
   SA_RESTORER|SA_INTERRUPT, sa_restorer=0x7fce288290e0}, {sa_handler=
   0x5641e0094100,sa_mask=[], sa_flags=SA_RESTORER|SA_INTERRUPT,sa_restorer
   =0x7fce288290e0}, 8) = 0
rt_sigreturn({mask=[]}) = 0

write(2, "hping in flood mode, no replies "..., 46) = 46
(Esta system call solo
   se utiliza en la primer sección)
```

Como se puede observar en la sección anterior, la primer *system call* es `sendto`, la cual tiene como finalidad transmitir un mensaje de un *socket* a otro, en este caso el mensaje será transmitido usando el *socket* `3` ya que el primer parámetro es `sockfd = 3`, este *socket*

se creó anteriormente como `socket(AF_INET, SOCK_RAW, IPPROTO_RAW) = 3`. El segundo parámetro contiene el mensaje, el tercer parámetro es la longitud del mensaje, el siguiente parámetro indica las banderas que se pueden adoptar, en este caso no hay ninguna bandera activa. Los siguientes dos parámetros contienen información del *socket* al que se enviará el mensaje. Cuando el *socket* no se encuentra en estado *"conectado"*, la dirección de la tarjeta se proporciona mediante estos dos últimos parámetros. La siguiente *system call* es `rt_sigaction` que recibe una señal para cambiar la acción tomada por un proceso. La señal recibida es `SIGALRM`, esta señal indica que el *"timer"* asociado finaliza, en el segundo argumento `act` se utilizan las banderas `sa_flags = SA_RESTORER SA_INTERRUPT`, la primer bandera indica que el campo `sa_restorer` contiene la dirección de la *"signal trampoline"* y la segunda bandera un *"operador"* de interrupciones rápido. La última *system call* de esta sección es `rt_sigreturn` que verifica los procesos pendientes por el *kernel*, de este modo guarda *"segmentos"* del contexto del proceso entre los cambios del espacio de usuario y así es posible reanudar ejecuciones en el punto donde fueron interrumpidas.

```
setsockopt(4, SOL_PACKET, PACKET_RX_RING, {block_size=0, block_nr=0,
   frame_size=0, frame_nr=0}, 16) = -1 EINVAL (Invalid argument)
munmap(0x7fce27e5b000, 2097152)         = 0
close(4)                                = 0
write(2, "\n--- 192.168.1.89 hping statisti"..., 38) = 38
write(2, "152 packets transmitted, 0 packe"..., 62) = 62
write(2, "round-trip min/avg/max = 0.0/0.0"..., 40) = 40
exit_group(1)                           = ?
```

La sección anterior de *system calls* muestra como se intentó manipular las opciones del *socket* `4` usando `setsockopt`, sus prámetros `level = SOL_PACKET`, `optname = PACKET_RX_RING` y `SOL_PACKET` indica el nivel del *socket* y `PACKET_RX_RING` crea un *buffer* de memoria mapeada para los paquetes de recepción asíncronos.
El valor de retorno de esta *system call* es `-1 EINVAL`, en algunos casos este error puede ocurrir por un valor invalido en el parámetro `optval`, el cual corresponde a la estructura `{block_size=0, block_nr=0, frame_size=0, frame_nr=0}` que contiene solo valores igual a `0`.

Por último, se utiliza `munmap` para eliminar el mapeo en la dirección `0x7fce27e5b000`, cierra el *socket* `4` y usa `exit group` para terminar el grupo de hilos del proceso de llamada.

Capítulo 6

Conclusiones

En este capítulo se describen las conclusiones generales y las perspectivas de investigación de esta tesis, así como el análisis general de los resultados obtenidos mediante las pruebas del ataque DoS.

6.1. Conclusiones generales

Esta tesis tuvo como finalidad explorar las posibles vulnerabilidades de un ambiente *paravirtualizado* a nivel *kernel* cunado éste es atacado por un DoS. Los resultados obtenidos en este estudio permiten entender el manejo de recursos, interrupciones al *kernel* y cuáles son las *system calls* que podrían ser usadas de forma maliciosa para efectuar un ataque tipo DoS en una red virtualizada. Por medio del ataque DoS implementado en el Capítulo 5 y mediante el uso de herramientas de análisis de penetración y de monitorización de sistemas fue posible confirmar lo estipulado en la hipótesis inicial de este trabajo:

Un ataque DoS mediante inundación de solicitudes SYN permite extraer las llamadas al sistema para crear un ataque a nivel kernel.

Como se mostró en el capítulo 5, fue posible hacer una traducción de un ataque *DoS* mediante inundación de solicitudes SYN y transformarlo a llamadas al sistema. Esto con el fin de hacer un ataque a nivel *kernel*, y que un IDS no pueda detectarlo. Más aún, se muestran otras posibles vulnerabilidades en este tipo de sistemas. Por ejemplo, una red definida por software, si bien limita sus debilidades a problemas directamente dentro de un equipo y no a las interfaces o equipos conectados a ésta, identificar peticiones de solicitudes de recursos legítimos (*system calls*) de las que no lo son representa un enorme reto para el desarrollo de este tipo de sistemas.

Se aprovechó la herramienta `hping3` ya que a nivel llamadas del *kernel*, se mostró cómo estas pasarían como una tarea normal en la red. Asimismo, los procesos que ésta satura son procesos que suelen estar presentes en una implementación típica de este tipo de sistema, como lo son `ksoftirqd` y los procesos propios del ambiente de paravirtualización Xen, como lo es `vif1.0-q0-deall`. Por ejemplo, las interrupciones de *timer* y *rescheduling* son generadas no sólo por éstos, sino por otros múltiples procesos que se realizan de forma regular en una red, lo que haría muy difícil analizarlos en un IDS, algo ya realizado en trabajos como [15]. Algunos de estos procesos pueden manipular el flujo de señales, lo cual les permite tener un control profundo de los recursos (asignación de memoria, control de procesos, etc).

La forma más directa de manipular las señales es por medio de las *system calls*, en este caso existen distintas *system calls* cuyas propiedades son un riesgo potencial, ya que son las mismas *system calls* que los procesos comunes utilizan y que pueden implicar un potencial riesgo para el sistema.

Por medio del estudio de las *system calls* en un ataque *DoS* se logró identificar algunas de ellas que pueden ser usadas para afectar el rendimiento de los procesos o recursos de un ambiente *paravirtualizado*. Una de las *system calls* que se analizaron en el estudio del ataque *DoS* fue *getrlimit*, la cual tiene como contraparte *setrlimit*, que puede ser usada para acortar la

memoria virtual en procesos y subprocesos mediante su parámetro `RLIMIT_AS`. Otras *system calls* que pueden ser usadas para comprometer los procesos de una máquina virtual son `mmap` y `rt _sigaction`, ya que estas *system calls* tienen la capacidad de realizar operaciones del sistema para el control de procesos, asignación dinámica de memoria y el manejo de señales.

Es claro que existen muchas otras *system calls* que pueden comprometer a un sistema; sin embargo, existen prácticas como el uso de filtros que nos permiten limitar el uso de ciertas *system calls*, la finalidad de estos filtros es que un proceso solo pueda acceder a las *system calls* indispensables para su correcto desempeño. Uno de estos filtros puede encontrarse en la biblioteca `libseccomp-golang` de *Seccom* disponible desde versiones Linux 2.6.12 [29].

Con todo esto se puede ver que las *system calls* analizadas no son el único bemol de las redes definidas por software, sino la dificultad o reto que representa el determinar la legitimidad de todas y cada una de las funciones que se realizan en paralelo en un sistema. Las llamadas al sistema pueden ser, como se explora en este trabajo, puestas en marcha de manera maliciosa y asimismo editarse para complicar aún más la tarea de un supervisor del sistema. De igual forma, las interrupciones al *kernel* y los procesos internos de la computadora se van haciendo cada vez más complejas, lo que conlleva una mayor capacidad de procesamiento. De esta manera, crear un sistema único que pueda revisar la legitimidad de cada elemento del sistema a nivel *kernel*, software y hardware podría parecer una tarea muy difícil.

Sin embargo, existe aún mucho campo por explorar en cuanto a la protección y monitorización de sistemas de redes definidas por software, que si bien comienzan a ser sistemas muy populares en la industria, el ir migrando más y más servicios a este tipo de plataformas, implica que los intentos de ataque a estas plataformas no tardarán en hacerse presentes, por lo que el determinar sus vulnerabilidades será el reto más importante.

6.2. Perspectivas de la investigación

Las redes definidas por software son un nicho de oportunidad no sólo para múltiples giros de negocio, sino para la investigación científica y la utilidad de ésta para múltiples ámbitos académicos. Uno de los elementos más populares en las SDN son los balanceadores de carga, sin embargo estos resultan inútiles si no se analiza el tipo de tráfico, ya que puede ser debido a un ataque DoS. Por ello es de suma importancia proporcionar un segundo sistema para las redes definidas por software, el cual en caso de detectar anomalías en el consumo de recursos por parte de un proceso pueda detener y reparar el daño hecho sin permitir el decaimiento en el rendimiento de la red. Dicho tipo de sistema de monitorización debe ser capaz de reconocer procesos que vienen con la intención de perjudicar el rendimiento de la red. Ejemplos de estas herramientas para filtrado son *Collabra* o *MAC/HAT* [16] que actualmente ya no son del todo seguros en versiones actuales de Xen. Sin embargo, podrían ampliarse y/o fortalecerse para que el sistema de red garantice una mejor estabilidad y alta disponibilidad. Esta meta podría alcanzarse a través del análisis estadístico y probabilístico más profundo implicando una matemática más avanzada y una capacidad de análisis mayor de todos los hilos, interrupciones y eventos que se suscitan en el *kernel* de una computadora, así como acceso a herramientas de experimentación que permitan reconocer el alcance de todas las limitaciones en los sistemas de este tipo y otras vulnerabilidades no exploradas que pudieren comprometer la integridad de la red.

Bibliografía

[1] M. A. OmarSefraoui and M. Eleuldj, "Openstack: Toward an open-source solution for cloud computing," *International Journal of Computer Applications*, vol. 55, no. 3, pp. 38–42, 2012.

[2] J. Shrosphire, "Hyperthreats: Hypercall-based dos attacks," in *Proceedings of the IEEE SoutheastCon 2015*, 2015.

[3] A. A. Barakabitze, A. Ahmad, and A. H. Rashid Mijumbi, "5g network slicing using sdn and nfv: A survey of taxonomy, architectures and future challenges," *Computer Networks*, vol. 167, 2020.

[4] W. Zhuang, Q. Ye, F. Lyu, and N. Chengand Ju Ren, "Sdn/nfv-empowered future iov with enhanced communication, computing,and caching," in *Proceedings of the IEEE*, 2019, pp. 1–18.

[5] Khondoker and Rahamatullah, in *SDN and NFV Security Security Analysis of Software-Defined Networking and Network Function Virtualization*, vol. 30, 2018, pp. 1–5.

[6] Y. Li and M. Chen, "Software-defined network function virtualization: A survey," *IEEE Access*, vol. 3, pp. 2542–2553, 2015.

[7] K. T. Raghavendra, S. Vaddagiri, N. Dadhania, and J. Fitzhardinge, "Paravirtualization for scalable kernel-based virtual machine (kvm)," in *2012 IEEE International Conference on Cloud Computing in Emerging Markets (CCEM)*, 2012, pp. 1–5.

[8] H. Zang, K. Gu, Y. Sun, and D. Meng, "Optimizing network paravirtualization with static shared memory pipe," in *2010 WASE International Conference on Information Engineering*, vol. 4, 2010, pp. 6–9.

[9] J. Guarch i Calvó, "Perfomance analysis of the xen hypervisor for virtualizing network devices," Ph.D. dissertation, UPC, Escola Tècnica Superior d'Enginyeria de Telecomunicació de Barcelona, Jul 2010. [Online]. Available: http://hdl.handle.net/2099.1/10455

[10] P. Emmerich, D. Raumer, F. Wohlfart, and G. Carle, "Performance characteristics of virtual switching," in *2014 IEEE 3rd International Conference on Cloud Networking (CloudNet)*, 2014, pp. 120–125.

[11] H. Fang and R. Obermaisser, "Virtual switch for integrated real-time systems based on sdn," in *2019 Sixth International Conference on Internet of Things: Systems, Management and Security (IOTSMS)*, 2019, pp. 344–351.

[12] "How to port open vswitch to new software or hardware," [Web; accedido el 15-08-2020]. [Online]. Available: URL{https://www.openvswitch.org/support/dist-docs-2.5/PORTING.md.txt}

[13] C. H. H. Le, "Protecting xen hypercalls: intrusion detection prevention in a virtualization environment," Ph.D. dissertation, University of British Columbia, 2009. [Online]. Available: https:\/\/open.library.ubc.ca\/collections\/ubctheses\/24\/items\/1.0051653

[14] K. Wall, "System programming," in *Linux Programming by Example*, J. Koch and G. Ganser, Eds. Indianapolis, Indiana: QUE, 2000, pp. 119–123.

[15] F. Wang, P. Chen, B. Mao, and L. Xie, "Randhyp: Preventing attacks via xen hypercall interface," in *Information Security and Privacy Research*, D. Gritzalis, S. Furnell, and M. Theoharidou, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 138–149.

[16] A. Milenkoski, B. D. Payne, N. Antunes, M. Vieira, and S. Kounev, "Experience report: An analysis of hypercall handler vulnerabilities," pp. 1–12, 2015.

[17] P. K. Mojtaba Mostafavi, "Detection of repetitive and irregular hypercall attacks from guest virtual machines to xen hypervisor," *Iran Journal of Computer Science*, vol. 1, pp. 89–97, 2018.

[18] T. S. Omar Jammal, A. Shami, R. Asal, and Y. Li, "Software-defined networking: State of the art and research challenges," *Elsevier's Journal of Computer Networks*, pp. 1–24, 2014.

[19] X. Project. Xen project beginners guide. [Online]. Available: https://wiki.xenproject.org/wiki/Xen_Project_Beginners_Guide

[20] L. F. C. Projects. Open vswitch on linux, freebsd and netbsd. [Online]. Available: https://docs.openvswitch.org/en/latest/intro/install/general/

[21] M. Kerrisk, "syscalls(2) — linux manual page," urlhttps://man7.org/linux/man-pages/man2/syscalls.2.html, 2020.

[22] E. Vicente, R. Matias, L. Borges, and A. Macêdo, "Evaluation of compound system calls in the linux kernel," in *2011 Brazilian Symposium on Computing System Engineering*, 2011, pp. 164–169.

[23] D. P. B. . M. Cesati, *Understanding the Linux Kernel*. O'Reilly, 2006.

[24] G. K.-H. Jonathan Corbet, Alessandro Rubini, *Linux Device Drivers*. O'Reilly Media, 2005, vol. 3.

[25] "hping3(8) - linux man page," [Web; accedido el 21-11-2020]. [Online]. Available: URL{https://linux.die.net/man/8/hping3}

[26] "Manpages: ksoftirqd – softirq daemon," [Web; accedido el 2-1-2021]. [Online]. Available: URL{https://man.cx/ksoftirqd}

[27] L. Liu, H. Wang, A. Wangand Mengbai Xiaoand Yue Cheng, and S. Chen, "vcpu as a container: Towards accurate cpuallocation for vms," in *15th ACM SIGPLAN/SIGOPSInternational Conference on Virtual Execution Environments (VEE'19)*, 2019, pp. 1–14.

[28] "top command on ubuntu multicore cpu shows cpu usage >100%," [Web; accedido el 2-1-2021]. [Online]. Available: URL{https://askubuntu.com/questions/707203/top-command-on-ubuntu-multicore-cpu-shows-cpu-usage-100}

[29] "Seccom - libseccomp-golang," [Web; accedido el 29-12-2020]. [Online]. Available: URL{https://github.com/seccomp/libseccomp-golang}