

Trabajo fin de grado

A Simple Linux Deception System



Javier Sánchez-Pascual Castromonte

Escuela Politécnica Superior
Universidad Autónoma de Madrid
C\Francisco Tomás y Valiente nº 11

UNIVERSIDAD AUTÓNOMA DE MADRID
ESCUELA POLITÉCNICA SUPERIOR



Grado en Ingeniería Informática

TRABAJO FIN DE GRADO

A Simple Linux Deception System

Monitorización oculta de intrusos en sistemas Linux

Autor: Javier Sánchez-Pascual Castromonte
Tutor: Óscar Delgado Mohatar

marzo 2020

Todos los derechos reservados.

Queda prohibida, salvo excepción prevista en la Ley, cualquier forma de reproducción, distribución comunicación pública y transformación de esta obra sin contar con la autorización de los titulares de la propiedad intelectual.

La infracción de los derechos mencionados puede ser constitutiva de delito contra la propiedad intelectual (*arts. 270 y sgts. del Código Penal*).

DERECHOS RESERVADOS

© 13 de Noviembre de 2019 por UNIVERSIDAD AUTÓNOMA DE MADRID
Francisco Tomás y Valiente, nº 1
Madrid, 28049
Spain

Javier Sánchez-Pascual Castromonte

A Simple Linux Deception System

Javier Sánchez-Pascual Castromonte

IMPRESO EN ESPAÑA – PRINTED IN SPAIN

A mi familia y amigos.

*El amor al saber, la curiosidad, el asombro; es lo que me sostiene.
Y creo que puede sostener a cualquier ser humano.*

Antonio Escohotado

RESUMEN

Durante años de carrera, el estudiante del grado de ingeniería informática se cultiva con una gran cantidad de información e intenta progresar en su capacidad de trabajo, individual y colectivo, en el ámbito de las tecnologías informáticas. Sin embargo, esto no es suficiente. En la sociedad actual existen una serie de cuestiones realmente demandadas, como la seguridad informática, que el estudiante, al terminar el grado, desconoce casi completamente. El objetivo de este trabajo es introducirse, aunque sea de forma mínima, en el mundo de la *ciberseguridad*.

En este sentido se ha desarrollado una aplicación que se comporta como un sistema de engaño, en inglés *Deception System*. Esta aplicación ha sido desarrollada en el sistema operativo Linux y su funcionamiento es muy sencillo: detecta y monitoriza toda la actividad posible de un usuario que hace el papel de atacante en el sistema.

Tanto los procesos de monitorización como los registros que contienen la actividad del intruso han sido ocultados, de tal forma que nadie excepto el administrador puede acceder a ellos. Asimismo, toda la información que se recolecta es enviada, mediante una conexión cifrada, a un ordenador central que hace el papel de cerebro del sistema de engaño. Este ordenador central simplemente muestra la información que recibe en tiempo real de la actividad del intruso, pudiendo trazar los movimientos del mismo. Así pues, por un lado, el sistema registra cierta información en una zona oculta del disco y, por otro, envía otros datos al centro de mando. La plataforma de engaño es invisible, o por lo menos difícil de detectar, para el usuario intruso.

PALABRAS CLAVE

Deception System, Ciberseguridad, Monitorización de usuarios, Ocultación de datos, Ocultación de procesos

ABSTRACT

During years of career, the student of the computer engineering degree cultivates himself with a large amount of information and tries to progress in his ability to work, individually and collectively, in the field of computer technologies. However, this is not enough. In today's society, there are a series of really demanded questions, such as computer security, which the student, after completing the degree, is almost completely unaware of. The objective of this work is to enter, albeit minimally, in the world of cybersecurity.

In this sense, an application has been developed that behaves like a Deception System. This application has been developed in the Linux operating system and its operation is very simple: it detects and monitors all the possible activity of a user who plays the role of attacker in the system.

Both the monitoring processes and the records that contain the activity of the intruders have been hidden, so that nobody except the administrator can access them. Also, all the information that is collected is sent, through an encrypted connection, to a central computer that acts as the brain of the deception system. This central computer simply displays the information it receives in real time from the activity of the intruder, being able to trace its movements. Thus, on the one hand, the system records certain information in a hidden area of the disk and, on the other, sends other data to the command center. The deception platform is invisible, or at least difficult to detect, to the intruder user.

KEYWORDS

Deception System, CiberSecurity, User Monitoring, Stealth, Data Concealment, Process Concealment

ÍNDICE

1 Introducción	1
1.1 Motivación	1
1.2 Primeras ideas	1
1.3 Profundizando	2
1.3.1 Qué es <i>Deception Technology</i> o Tecnología de Engaño	2
1.3.2 Porqué usar <i>Deception Technology</i>	2
1.3.3 Arquitectura de un sistema de engaño real	3
2 Análisis y aspectos generales	5
2.1 Descripción del trabajo	5
2.2 Descomposición en subsistemas	5
2.2.1 Subsistema de Monitorización	7
2.2.2 Subsistema de Ocultación de Evidencias	7
2.2.3 Subsistema de Comunicación	7
2.3 Especificación de requisitos	7
2.3.1 Requisitos funcionales	7
2.3.2 Requisitos no funcionales	9
3 Diseño	11
3.1 Diseño del subsistema de Monitorización	11
3.1.1 Monitorización del sistema de ficheros	11
3.1.2 Monitorización de las Llamadas al Sistema	14
3.2 Diseño del subsistema de Ocultación	17
3.2.1 Ocultación de procesos en Linux	17
3.2.2 Ocultación de datos en Linux	21
3.2.3 Ocultación de puertos en Linux	23
3.3 Diseño del subsistema de Comunicación	26
3.3.1 Comunicación cifrada entre director y agente: Protocolo <i>TLS</i>	26
4 Implementación	29
4.1 Implementación del Agente de engaño	29
4.1.1 Implementación subsistema de monitorización	29
4.1.2 Implementación subsistema de ocultación	31
4.1.3 Implementación subsistema de comunicación (I)	36

4.2	Implementación del Director de engaño	38
4.2.1	Implementación del subsistema de comunicación (II)	38
5	Conclusiones y mejoras	39
5.1	Posibles mejoras de la aplicación	39
5.2	Conclusiones finales	39
	Bibliografía	42
	Acrónimos	43
	Apéndices	45
A	Código definitivo de <i>agente.py</i>	47
B	Código definitivo de la librería compartida.	55
C	Código para la creación del certificado auto-firmado.	61
D	Código del hilo de la aplicación <i>Flask</i>.	65
E	Manual de usuario y contenido de la entrega	67

LISTAS

Lista de códigos

4.1	Primera versión de <i>readdir()</i> para la ocultación del proceso agente de engaño (I)	31
4.2	Primera versión de <i>readdir()</i> para la ocultación del proceso agente de engaño (II)	32
4.3	Función <i>is_sudo()</i>	33
4.4	Fichero Makefile para la compilación de la librería compartida <i>libprocesshider.c</i> usando <i>gcc</i>	34
4.5	Compilación e instalación de la librería compartida.	35
A.1	Versión definitiva de <i>agente.py</i> (I)	47
A.2	Versión definitiva de <i>agente.py</i> (II)	48
A.3	Versión definitiva de <i>agente.py</i> (III)	49
A.4	Versión definitiva de <i>agente.py</i> (IV)	50
A.5	Versión definitiva de <i>agente.py</i> (V)	51
A.6	Versión definitiva de <i>agente.py</i> (VI)	52
A.7	Versión definitiva de <i>agente.py</i> (VII)	53
B.1	Código librería compartida. (I)	55
B.2	Código librería compartida. (II)	56
B.3	Código librería compartida. (III)	57
B.4	Código librería compartida. (IV)	58
B.5	Código librería compartida. (V)	59
C.1	Creación certificado auto-firmado (I)	61
C.2	Creación certificado auto-firmado (II)	62
C.3	Creación certificado auto-firmado (III)	63
D.1	Código del hilo de la aplicación Flask (I)	65
D.2	Código del hilo de la aplicación Flask (II)	66

Lista de figuras

1.1	Arquitectura básica de un sistema de engaño real	4
2.1	Descomposición de la funcionalidad a implementar en tres subsistemas: monitorización, ocultación y comunicación	5

2.2 Relación entre los componentes de la aplicación con los subsistemas y con las herramientas utilizadas para su implementación.	6
3.1 Descomposición del subsistema de monitorización	11
3.2 Eventos inotify generados por la creación de un fichero	12
3.3 Descomposición del subsistema de ocultación	17
3.4 Funcionamiento del comando <i>lsof</i>	18
3.5 Salida del comando ' <i>strace ps</i> '	19
3.6 Cadena de llamadas que se producen con el comando <i>ps</i>	20
3.7 Información relativa a conexiones TCP que muestra <i>strace</i> sobre el comando <i>netstat</i>	24
3.8 Contenido del fichero <i>/proc/net/tcp</i>	24
3.9 <i>/proc/net</i> es un enlace simbólico a <i>/self/net</i>	25
3.10 <i>/proc/self</i> es un enlace simbólico a <i>/proc/PID</i>	25
3.11 Cadena de enlaces simbólicos de <i>/proc/net/tcp</i>	25
3.12 Descomposición del subsistema de comunicación	26
3.13 Ubicación de TLS en la pila de protocolos	26
4.1 Ejecución del comando <i>cat</i> sobre el fichero <i>/proc/net/tcp</i> usando la nueva versión de <i>read()</i>	36
E.1 Crear y activar el entorno virtual e instalar la aplicación <i>flaskr</i>	67
E.2 Exportar <i>flaskr</i> , iniciar la base de datos y ejecutar la aplicación	68

Lista de tablas

3.1 Tabla de traducción de eventos	13
--	----

INTRODUCCIÓN

Como su propio nombre indica, la intención de este primer capítulo es servir como una pequeña presentación. Inicialmente se explicará la motivación del proyecto. Seguidamente se describirán las ideas iniciales que surgieron. En la última sección, se profundizará en la investigación y el desarrollo de estas ideas.

1.1. Motivación

Sería del todo incorrecto no decir que la única motivación que impulsó este proyecto fue simple y llanamente la curiosidad. Durante años de carrera, se han abordado multitud de temas: gestión de la información en bases de datos, funcionamiento de colas en sistemas distribuidos, inteligencia artificial, estudio del funcionamiento de sistemas operativos, estudio de algoritmos de ordenación y de búsqueda eficientes, paradigmas de programación de multitud de lenguajes, y un largo etc.

Sin embargo, en la actualidad existen cuestiones realmente demandadas, como la seguridad informática o la popular tecnología *Blockchain* (entre otras), con las que el estudiante apenas se encuentra de oídas.

No ha sido sino esa **falta de contacto** la causa de este trabajo.

1.2. Primeras ideas

Así pues, surge la necesidad de tener una pequeña toma de contacto con la seguridad en el ámbito de la informática. Para una persona completamente ajena al tema en cuestión, surgen **tres frentes** de actuación que deben ser atendidos:

1.- Prevención

Abarcaría todas las medidas cuyo propósito principal fuese impedir el acceso de un intruso en el sistema. Entre ellas se incluirían los conocidos antivirus y cortafuegos. También se consideraría una medida de prevención la formación básica de los empleados del negocio, para no cometer errores evitables que den pie a intrusiones no

deseadas.

2.- Reacción

Sería ingenuo pensar que nadie será capaz de saltarse todos los protocolos de seguridad. Es obligatorio planificar cómo afrontar tal suceso. Se debería empezar por la creación de dispositivos de alarma, colocados estratégicamente para que actúen como disparadores ante una intrusión. Una vez identificado el intruso, el siguiente paso es decidir qué acciones tomar: monitorizarlo, dirigirlo, expulsarlo, interrumpirlo, rastrearlo, etc.

3.- Análisis

Una vez pasada la amenaza, sería hora de evaluar daños. Se debería realizar un análisis resolviendo las siguientes cuestiones: quién era el intruso, cómo ha entrado, qué quería, qué ha conseguido, y cómo podemos evitar que vuelva a ocurrir.

Como se puede suponer, cada una de estas tres ramas de investigación son extremadamente densas de información y contenido. Este proyecto se centra exclusivamente en la parte de reacción. Se busca, por tanto, una metodología que permita detectar, analizar y defenderse de ataques avanzados en tiempo real. La tecnología que se ajusta a esta descripción es conocida como 'Deception Technology'.

1.3. Profundizando

1.3.1. Qué es *Deception Technology* o Tecnología de Engaño

El objetivo de la tecnología de engaño es prevenir de cualquier daño significante que pueda provocar un atacante que se haya introducido en una red privada. Los productos comerciales que implementan esta tecnología son automáticos, precisos, y proporcionan información sobre actividad maliciosa dentro de las redes internas que otros sistemas de defensa no pueden ver. Esto se consigue creando trampas o señuelos que se colocan estratégicamente por toda la infraestructura. Estos señuelos simulan ser activos o bienes tecnológicos importantes dentro de la organización. Una vez que se activa una trampa, se transmiten una serie de notificaciones a un servidor central, que registra el señuelo afectado y los métodos de ataque utilizados por el intruso, y desde donde se podrá hacer un seguimiento detallado de toda la actividad del atacante.

1.3.2. Porqué usar *Deception Technology*

Detección temprana y retroalimentación

En seguridad informática no existe una solución que permita neutralizar todas las amenazas que se producen en una red. La tecnología de engaño se pone en marcha una vez un señuelo es activado. Los atacantes se confían y creen que se han afianzado en la red ajena, cuando en realidad están siendo

monitorizados. La información registrada sobre el comportamiento y técnicas que emplea el atacante servirán para mejorar aún más el sistema de seguridad.

Reducción de falsos positivos

Una gran cantidad de falsos positivos pueden agotar los recursos de análisis si se requiere el examen de cada alerta. Por otro lado, demasiado ruido puede provocar que los sistemas de seguridad sean complacientes a la hora de diagnosticar alertas e ignoren una posible amenaza legítima. La tecnología de engaño permite reducir drásticamente los falsos positivos con alertas de gran confianza: sólo se produce una alarma al activarse un señuelo, que sólo se debería activar si es accedido por alguien que no sabe que es un señuelo; es decir, un intruso.

Escalable y Automatizable

Las alertas automáticas eliminan la necesidad de esfuerzo e intervención manuales, además de que el diseño de la tecnología permite escalarlo fácilmente a medida que crece la organización y el nivel de amenaza.

Bajo riesgo

La tecnología de engaño también es de muy bajo riesgo debido a que no supone ningún peligro para los datos ni provoca un gran impacto en los recursos. Cuando un atacante accede al sistema de engaño, se genera una alerta real que le indica al administrador que debe tomar medidas.

1.3.3. Arquitectura de un sistema de engaño real

Existen multitud de productos y herramientas comerciales que implementan soluciones 'Deception Technology'. En la mayoría de ellas se sigue la misma estructura para la formación del sistema de engaño:

- ***Deception Director* o director de engaño**

Es la consola y el corazón de toda la plataforma o sistema de engaño. El director permite manejar todos los aspectos del entorno de engaño, y todos los eventos son supervisados y analizados desde la interfaz web.

- ***Deception Hosts* o huéspedes de engaño**

Estos son los servidores y/o ordenadores de escritorio que están desplegados y que tienen el agente de engaño instalado. Estos sistemas pueden funcionar bajo cualquier sistema operativo, incluidos Windows y Linux. Pueden ser dispositivos físicos, máquinas virtuales, o incluso estar situados en la nube.

- **Agentes**

Son la pieza de software encargada de monitorizar toda la actividad del dispositivo en el que se encuentren instalados, y reportar toda esa información al director. Los agentes están completamente ocultos, siendo muy

difíciles de detectar para los intrusos, puede que, incluso, imposible.

- **Activos de engaño**

Son la base del entorno de engaño, llamados tarros de miel (en inglés *honeypots*) o migas de pan (*breadcrumbs*). Su función es la de cebo que es descubierto por los posibles atacantes y que los atrae para sumergirlos aún más en el entorno de engaño. Dependiendo del caso, pueden tomar varias formas y tamaños; por ejemplo: servidores con un sistema operativo específico, servicios concretos (carpetas compartidas, aplicaciones web, *SSH*, etc.), documentos (Google Docs, Office, etc.), o, incluso, credenciales para dirigir al atacante a otro huésped del sistema de engaño.

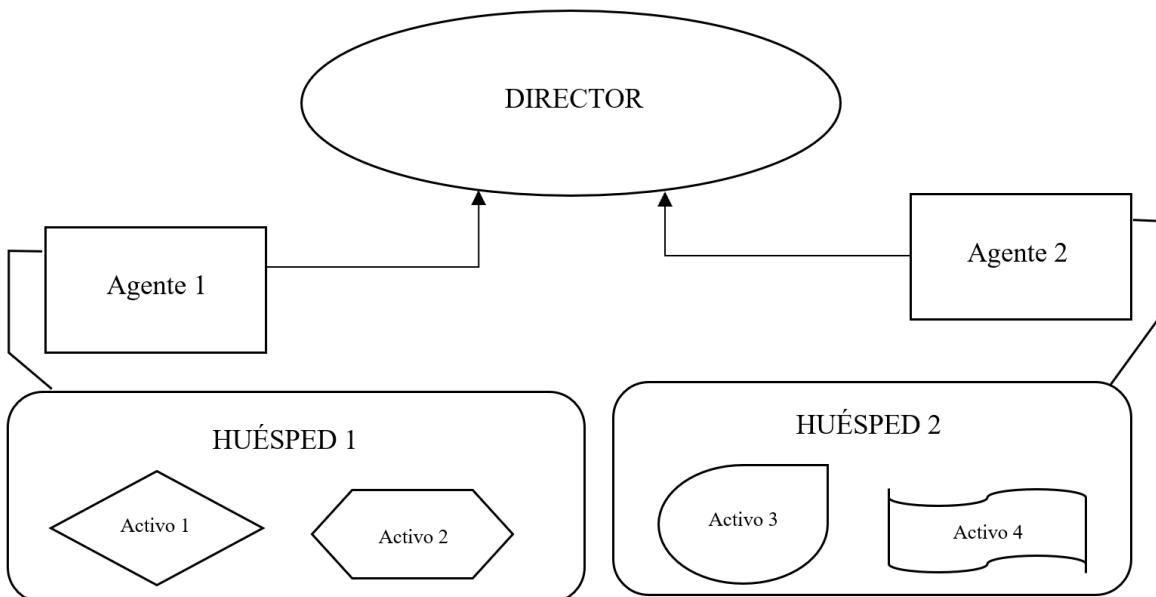


Figura 1.1: Arquitectura básica de un sistema de engaño real.

En la figura 1.1 se muestra un esquema básico de un sistema de engaño. Con esto concluye el capítulo de introducción. En el segundo capítulo se realiza un análisis del problema con una descomposición en subsistemas, así como la especificación de requisitos funcionales y no funcionales. En el tercero se lleva a cabo un estudio del estado de las tecnologías y el diseño de la aplicación. En el cuarto capítulo se explican los detalles del desarrollo para la implementación del sistema de engaño. Por último, en el quinto capítulo, se enuncian las conclusiones y las posibles mejoras del trabajo realizado. [1] [2] [3] [4] [5] [6]

ANÁLISIS Y ASPECTOS GENERALES

2.1. Descripción del trabajo

Como en cualquier otro proyecto, se debe seguir una metodología de desarrollo. En este caso, debido a las características del trabajo, se pretende seguir una metodología similar al desarrollo en cascada, donde cada uno de los pasos esté perfectamente definido. El objetivo es desarrollar un sistema de engaño (*Deception System*) plenamente funcional en Linux, siguiendo la arquitectura descrita en la subsección 1.3.3. Concretamente, se quiere crear un sistema de monitorización que registre toda la actividad de un usuario específico que hará el papel de intruso en el sistema. Este intruso no debe ser capaz, o no al menos de forma sencilla, de detectar que está siendo espiado.

2.2. Descomposición en subsistemas

Para una mayor facilidad a la hora afrontar los retos que van a surgir, y para agrupar las funcionalidades a implementar, se ha dividido el trabajo en tres subsistemas que agruparán los requisitos relacionados: subsistema de monitorización, subsistema de ocultación y subsistema de comunicación.



Monitorización



Ocultación



Comunicación

Figura 2.1: Descomposición de la funcionalidad a implementar en tres subsistemas: monitorización, ocultación y comunicación. Imagen creada con *Microsoft Word*.

A su vez, la aplicación constará de dos partes bien diferenciadas: el agente y el director. El agente de engaño estará involucrado en prácticamente toda la funcionalidad de la aplicación, por lo que

incluirá todo lo relativo a los subsistemas de monitorización, ocultación y parte del subsistema de comunicación. Por otro lado, el director simplemente recibirá las evidencias de seguimiento recolectadas por el agente y las mostrará al administrador, por lo que solamente incluirá su parte correspondiente del subsistema de comunicación.

El huésped o huéspedes de engaño serán máquinas virtuales donde se ejecute el agente. El director se ejecutará en una máquina aparte. Los activos de engaño serán simples ficheros que pondrán en marcha la plataforma de engaño una vez sean accedidos. En la figura 2.2 se muestra un esquema de la arquitectura de la aplicación, donde se señala la relación entre los elementos de la plataforma de engaño y los distintos subsistemas. También se incluye un adelanto de las herramientas utilizadas para su implementación.

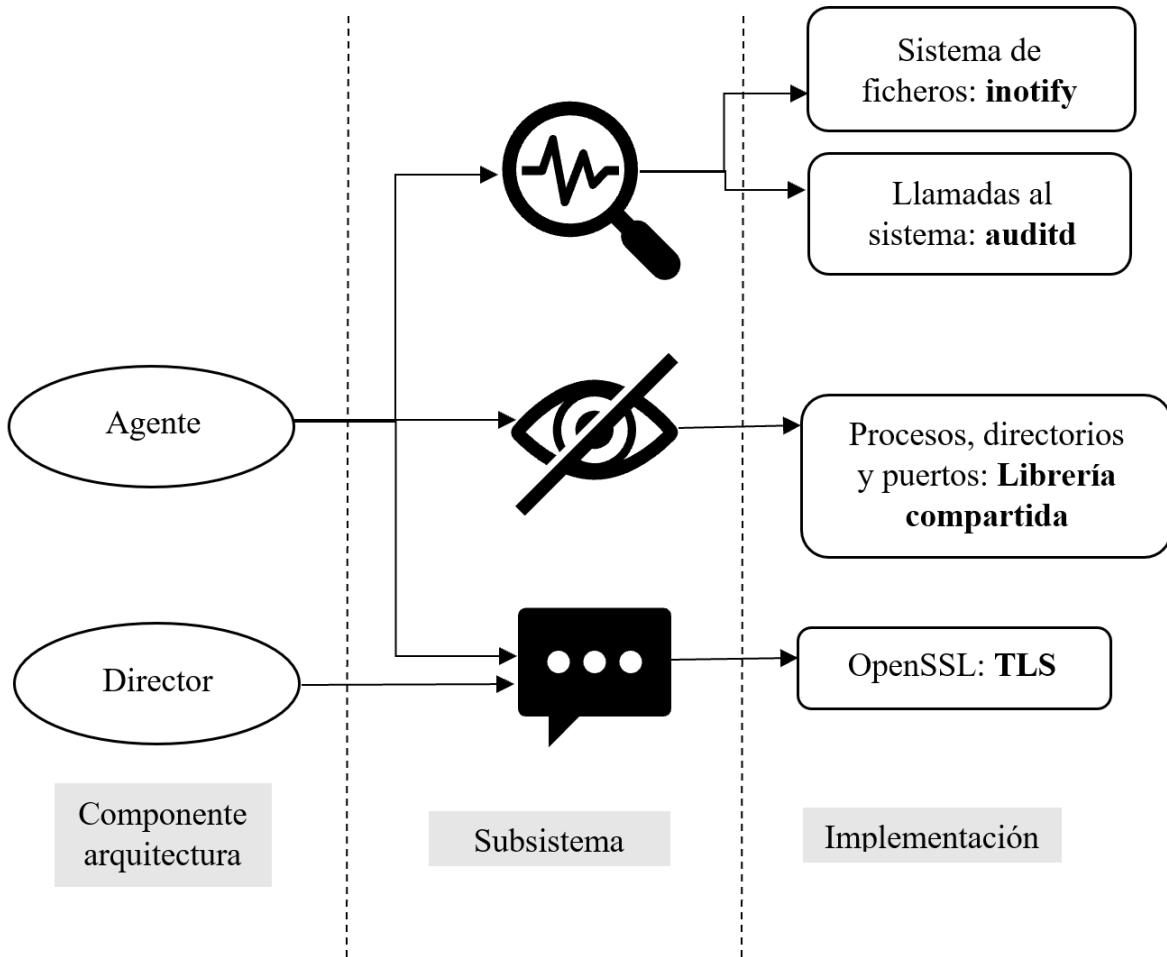


Figura 2.2: Relación entre los componentes de la aplicación con los subsistemas y con las herramientas utilizadas para su implementación. Imagen creada con *Microsoft Word*.

2.2.1. Subsistema de Monitorización

El sistema monitorizará toda la actividad del usuario que se le indique. Se incluyen, por tanto, todos los cambios que dicho usuario haga en el sistema de ficheros y todas las llamadas al sistema que realice. Además, se registrará toda esta información en ficheros de recolección de evidencias. El agente de engaño se encargará de implementar este subsistema en su totalidad.

2.2.2. Subsistema de Ocultación de Evidencias

Con el objetivo de que el atacante no sepa que se siguen todos sus movimientos, el sistema ocultará los procesos de monitorización para que el intruso no pueda descubrirlos de forma sencilla. Los ficheros de recolección de evidencias también deberán ser escondidos para el usuario intruso. Para ello, la aplicación dispondrá de un espacio de memoria secreto donde almacenará todas las evidencias. Además de esto, igualmente será necesario encubrir la comunicación entre los diferentes componentes de la plataforma de engaño. Por ello, se deberán tapar las conexiones que se establezcan entre el director y el agente. Este será, sin duda, el subsistema más elaborado y será también el agente de engaño el encargado de implementarlo.

2.2.3. Subsistema de Comunicación

El director recibirá del agente de engaño la información que este registre y mostrará por pantalla todo el seguimiento en tiempo real que se esté realizando del atacante. Por lo tanto, la funcionalidad de este subsistema se repartirá entre el director y el agente.

2.3. Especificación de requisitos

2.3.1. Requisitos funcionales

RF-1.- La plataforma de engaño estará formada por dos programas: el director y el agente.

RF-2.- Tanto el director como el huésped de engaño funcionarán bajo una distribución del sistema operativo Linux.

RF-3.- El huésped o huéspedes podrán ser una máquina virtual ejecutada por un producto de virtualización como Virtual Box o VMWare.

RF-4.- El activo o activos de engaño harán la función de cebo para activar la plataforma de engaño y podrán tener las siguientes formas:

- Documentos de texto, Google Docs o Microsoft Office.
- Credenciales de acceso a otra máquina huésped del entorno de engaño.
- Carpetas compartidas.
- Aplicaciones Web.

AGENTE

Subsistema de Monitorización

- RF-5.**– El agente será desarrollado en *Python*.
- RF-6.**– El agente sólo podrá ser ejecutado por el administrador.
- RF-7.**– El agente recibirá como argumento el nombre del usuario que se desea monitorizar.
- RF-7.1.**– El agente comprobará que el usuario existe, y si no existiera mostrará un aviso.
- RF-8.**– El agente realizará una copia actual del directorio */home* del usuario seleccionado.
- RF-9.**– El agente monitorizará el sistema de ficheros del usuario seleccionado.
- RF-9.1.**– Realizará una copia de los nuevos ficheros que cree el usuario.
- RF-9.2.**– Realizará una copia de las modificaciones que realice el usuario sobre un fichero.
- RF-9.3.**– Realizará una copia de los nuevos directorios que cree el usuario.
- RF-9.4.**– Registrará si un fichero o directorio ha sido eliminado por el usuario. En este caso, las copias de los ficheros y/o directorios se marcarán para indicar que el usuario ha borrado los originales.
- RF-9.5.**– Registrará si un fichero o directorio ha sido movido por el usuario.
- RF-10.**– El agente monitorizará todas las llamadas al sistema que realice el usuario seleccionado.
- RF-10.1.**– Registrará la información de las llamadas al sistema realizadas por el usuario en un fichero.
- RF-10.2.**– A través de este fichero, permitirá filtrar las llamadas al sistema que haya realizado el usuario. Por ejemplo, mostrar solamente las llamadas *execve* que haya realizado el usuario.

Subsistema de Ocultación de Evidencias

- RF-11.**– El agente será indetectable mediante el comando *ps* de Linux (o similares) para cualquier usuario que no sea el administrador.
- RF-12.**– El agente almacenará los ficheros de recolección de evidencias en un directorio escondido en el sistema.
- RF-12.1.**– El directorio estará oculto a la interfaz de usuario.
- RF-12.2.**– El directorio sólo será accesible por el usuario administrador.
- RF-12.3.**– El directorio será indetectable mediante el comando *ls* de Linux (o similares) para cualquier usuario que no sea el administrador.
- RF-13.**– El agente ocultará o camuflará el puerto o los puertos de conexión con el director de engaño. Estos puertos no deberán aparecer al utilizar comandos como *netstat* o *ss*.

Subsistema de Comunicación I

- RF-14.**– El agente se comunicará con el director mediante una comunicación cifrada.
- RF-15.**– Cada registro que el agente almacene en el directorio de recolección de evidencias será también mandado al director.

DIRECTOR

Subsistema de Comunicación II

- RF-16.**– El director será una aplicación web desarrollada en *Python*, utilizando *Flask*.
- RF-17.**– El director abrirá una conexión segura para establecer comunicación con el agente.

RF-18.– El director almacenará en una base de datos *Sqlite* todas las evidencias que reciba del agente.

RF-19.– El director mostrará por pantalla las evidencias almacenadas en la base de datos.

2.3.2. Requisitos no funcionales

RNF-1.– La plataforma deberá ser estable.

RNF-2.– El agente no debe ocupar más del 20 por ciento de la *CPU*.

RNF-3.– El director no debe ocupar más del 10 por ciento de la *CPU*.

RNF-4.– La comunicación entre director y agente será mediante *TLS*.

RNF-5.– El director mostrará las evidencias recibidas del agente en tiempo real según le vayan llegando.

RNF-6.– Tanto el director como el agente deberán de funcionar de forma fluida y sin errores.

DISEÑO

En esta fase se realizarán los análisis necesarios para saber qué herramientas utilizar en la etapa de implementación. Se respetará la descomposición en subsistemas propuesta en la sección 2.2, realizando un estudio del diseño separado en cada uno de ellos. La prioridad del diseño en todos los subsistemas será seguir el principio Keep It Simple Stupid (KISS) [7].

3.1. Diseño del subsistema de Monitorización

El diseño de este subsistema ha sido dividido a su vez en dos bloques bien diferenciados: monitorización del sistema de ficheros y monitorización de las llamadas al sistema (ver figura 3.1).

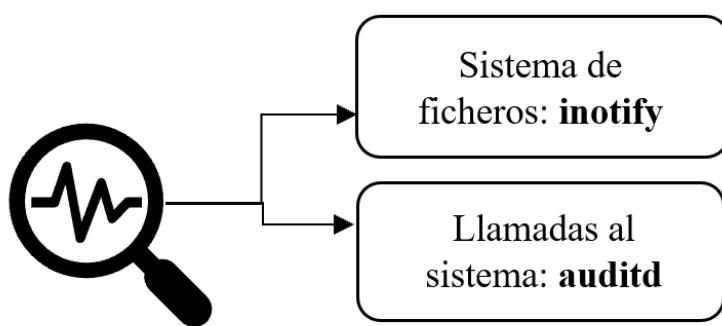


Figura 3.1: Descomposición del subsistema de monitorización. Imagen creada con *Microsoft Word*.

3.1.1. Monitorización del sistema de ficheros

Probablemente la forma más sencilla de monitorizar un sistema de ficheros sea con *inotify*. Esta funcionalidad se encuentra disponible en el *kernel* de Linux, y permite registrar uno o más directorios para ser observados. Esto es obviamente más eficiente que examinar periódicamente uno o más directorios para comprobar si algo ha cambiado. *inotify* está disponible en el *kernel* de Linux a partir de la versión 2.6. [8]

Existe una librería de *Python* que funciona como un generador de observadores *inotify*. Lo único que hay que hacer es crear un bucle infinito, donde se registrará un evento en cada iteración. Una sola acción realizada sobre un directorio monitorizado con *inotify* genera varios eventos; por ejemplo, el hecho de crear un fichero en un directorio monitorizado provocará varios eventos que el observador *inotify* capturará. Esto se pueden ver en la figura 3.2.

```
javier@javier-Lenovo-YOGA-900-13ISK:~/Escritorio/pruebas$ python3 prueba.py
PATH=[/home/javier/Escritorio/pruebas] FILENAME=[hola.txt] EVENT_TYPES=['IN_CREATE']
PATH=[/home/javier/Escritorio/pruebas] FILENAME=[hola.txt] EVENT_TYPES=['IN_OPEN']
PATH=[/home/javier/Escritorio/pruebas] FILENAME=[hola.txt] EVENT_TYPES=['IN_ATTRIB']
PATH=[/home/javier/Escritorio/pruebas] FILENAME=[hola.txt] EVENT_TYPES=['IN_CLOSE_WRITE']
```

Figura 3.2: Eventos *inotify* capturados al crear un fichero *hola.txt* en el directorio monitorizado *pruebas*

En la web de *Python* se muestran ejemplos sencillos sobre cómo usar el paquete de *python-inotify*. Una de las ventajas de esta librería es que permite la observación recursiva: si se crean directorios dentro de directorios con un observador *inotify*, esos directorios anidados también serán observados. Se puede consultar toda la información sobre la *API* de *inotify* en el manual del programador de Linux. A continuación, se detallan todos los eventos *inotify*:

IN_ACCESS (+) Se ha accedido al archivo (por ejemplo con *read(2)* o *execve(2)*)

IN_ATTRIB (*) Se han modificado los metadatos. Por ejemplo: permisos (con *chmod(2)*), *timestamps* (con *utimensat(2)*), atributos extendidos (*setxattr(2)*), número de enlaces (con *link(2)* y *unlink(2)*), y/o el identificador de usuario/grupo (*chown(2)*).

IN_CLOSE_WRITE (+) Se ha cerrado el archivo que anteriormente fue abierto para escritura.

IN_CLOSE_NOWRITE (*) Se ha cerrado el archivo que anteriormente fue abierto pero no para escritura.

IN_CREATE (+) Se ha creado un archivo o directorio en el directorio observado.

IN_DELETE (+) Se ha eliminado un archivo o directorio en el directorio observado.

IN_DELETE_SELF El archivo o directorio observado se ha auto-eliminado. (Este evento ocurre si el objeto es movido a otro sistema de ficheros, debido a que *mv(1)* en realidad copia el archivo a otro sistema y luego elimina el original).

IN MODIFY (+) Se ha modificado el archivo.

IN_MOVE_SELF El directorio o archivo observado se ha auto-movido.

IN_MOVED_FROM (+) Generado para el directorio que contiene el nombre del archivo viejo cuando este es renombrado.

IN_MOVED_TO (+) Generado para el directorio que contiene el nombre del archivo nuevo cuando este es renombrado.

IN_OPEN (*) Se ha abierto un archivo o directorio

Los eventos anteriores que están marcados con un asterisco (*) pueden ocurrir tanto para el directorio observado como para objetos dentro del propio directorio. Los eventos marcados con un símbolo de suma (+) ocurren solamente para los objetos internos contenidos en el directorio observado (y no para el directorio en sí).

Traducción de eventos

Al generarse varios eventos para una sola acción del usuario sobre un directorio monitorizado con *inotify*, es necesario establecer una traducción de eventos. Ésta permitirá convertir una secuencia de eventos en una acción particular del usuario. Obviando la apertura, hay únicamente cuatro acciones posibles sobre un fichero (crear, modificar, mover y eliminar) y tres sobre un directorio (crear, mover y eliminar). Nótese que renombrar un fichero o un directorio es lo mismo que moverlo.

Como se puede ver en la figura 3.2, al crear un fichero se generan varios eventos aparte de IN_CREATE. Son estos los que dificultan la lectura en crudo de los datos capturados por *inotify*, y los que fuerzan a establecer una traducción más legible.

Para poder traducir una secuencia de eventos en una de las cuatro acciones posibles, es indispensable saber cuáles son los eventos que forman cada acción. Así pues, una vez se sabe cuáles son los eventos de todas las cadenas, se puede realizar la traducción deseada. En la tabla 3.1 se muestran la traducción resumida de los eventos correspondientes a las cuatro acciones posibles del usuario.

Acción usuario	Cadena de eventos
FICHERO CREADO	IN_CREATE + IN_OPEN + IN_ATTRIB + IN_CLOSE_WRITE
FICHERO ELIMINADO	IN_DELETE
FICHERO MOVIDO	IN_MOVED_FROM + IN_MOVED_TO
FICHERO MODIFICADO	IN_OPEN + IN_MODIFY + IN_CLOSE_WRITE
DIRECTORIO CREADO	igual que FICHERO CREADO + IN_ISDIR
DIRECTORIO ELIMINADO	igual que FICHERO ELIMINADO + IN_ISDIR
DIRECTORIO MOVIDO	igual que FICHERO MOVIDO + IN_ISDIR

Tabla 3.1: Tabla que muestra la traducción de cadenas de eventos a acciones particulares de un usuario

Una vez identificada la cadena de eventos, se puede proceder a tomar la respuesta que se desee. Por ejemplo, cuando se reconozca la secuencia correspondiente a la acción *FICHERO CREADO* se puede proceder a copiar el fichero en el directorio secreto donde se guarden las evidencias. Por tanto, se puede relacionar cada acción del usuario con una respuesta que debe tomar el subsistema de monitorización. Con este fin, se muestra cuál es la respuesta que llevará a cabo la aplicación para cada acción identificada de las cadenas de eventos:

FICHERO CREADO Copiar el nuevo fichero en la copia del directorio `/home` contenida en la carpeta oculta de evidencias.

FICHERO ELIMINADO Marcar la copia del fichero como «*DELETED*».

FICHERO MOVIDO Mover la copia del fichero a la ruta correspondiente en la copia del directorio `/home` del usuario.

FICHERO MODIFICADO Copiar los cambios del fichero en la copia del directorio oculto.

DIRECTORIO CREADO Igual que *FICHERO CREADO*.

DIRECTORIO ELIMINADO Igual que *FICHERO ELIMINADO*.

DIRECTORIO MOVIDO Igual que *FICHERO MOVIDO*.

Con esto concluiría la monitorización del sistema de ficheros.

3.1.2. Monitorización de las Llamadas al Sistema

Se dedicó un tiempo considerable a averiguar cuál es la mejor forma, en este caso, la más sencilla; de monitorizar llamadas al sistema que hace un usuario concreto. A las llamadas al sistema también se las denomina *syscalls* (del inglés *system calls*). Inicialmente, la intención fue capturar todas y cada una de las *syscalls* que efectuaba el usuario atacante. Sin embargo, a lo largo de esta subsección se verá que esto no será posible sin perjudicar notablemente el rendimiento del sistema. Por ese motivo, será necesario monitorizar las llamadas al sistema de una forma más selectiva.

ptracer

La primera opción que se estudió fue *ptracer*. Según la web de *Python*, es una librería que permite monitorizar o rastrear bajo demanda llamadas al sistema en programas *Python*. A priori es exactamente lo que se buscaba: capturar *syscalls* de un usuario con un pequeño programa de *Python*.

Sin embargo, esta interpretación es errónea. En la documentación de la librería *ptracer* se explica que sólo monitoriza las llamadas al sistema que realiza un programa de *Python* concreto, es decir, no captura todas las llamadas al sistema, ni tampoco las que hace un usuario específico. *ptracer* podría ser útil para optimizar el uso de memoria que hacen ciertos programas escritos en *Python*. Por ejemplo, se podría utilizar para comprobar cuántos descriptores de ficheros quedan abiertos viendo si se realizan el mismo número de llamadas *open* y *close*. No obstante, para el objetivo de este subsistema, *ptracer* no es útil.

La herramienta *sysdig* [9] [10]

La segunda opción que se consideró fue hacer uso de *sysdig*. Es una herramienta para la resolución de problemas, análisis y exploración de sistemas. Se puede usar para capturar llamadas al sistemas y

otros eventos del sistema operativo. También puede ser usada para inspeccionar sistemas en tiempo real o para generar archivos de trazado que más tarde se pueden analizar. Además, incluye un lenguaje de filtrado formado por una serie de comandos (denominados *chisels*) que permiten personalizar la salida. Toda la información relativa a *sysdig* y las opciones que ofrece se pueden consultar con «*man sysdig*» (debe estar instalada la herramienta).

Para capturar todas las llamadas al sistema que realiza un usuario se puede utilizar la siguiente combinación: «`$ sysdig -w evidencias.scap user.name=juan`». En el momento en el que se ejecuta esto, *sysdig* comenzará a capturar todas las llamadas al sistema relacionadas con el usuario *juan*, y las registrará en el fichero *evidencias.scap*. Una vez termine el proceso de captura, este fichero de trazado podrá ser leído con «`$ sysdig -r evidencias.scap`». Con esto ya se habría encontrado una forma relativamente sencilla de monitorizar todas las *syscalls* que realizará el usuario intruso.

Sin embargo, lo cierto es que se encontraron algunos problemas al hacer las pruebas de monitorización con esta herramienta. El principal inconveniente es que *sysdig* consume demasiados recursos del procesador. Registrar todas las llamadas al sistema que provoca un sólo usuario ocupa entorno a un 30 % de la capacidad de la *CPU*. La principal consecuencia de esto es la sobrecarga y la ralentización del sistema. En casi todas las pruebas que se realizaron el sistema colapsó y se bloqueó, siendo necesario el reinicio del mismo.

Para intentar solventar este problema, se aumentó el área de intercambio de la máquina virtual donde se realizaron las pruebas, sin embargo, los resultados fueron los mismos. Es por esto que, a pesar de que *sysdig* hace lo que tiene que hacer, en este caso no es viable para monitorizar todas las llamadas al sistema del atacante. Se buscó, por tanto, una alternativa que no afectara tanto al rendimiento.

El demonio de auditoría de Linux: *auditd* [11] [12]

Finalmente se optó por utilizar esta herramienta, aunque es una opción algo más compleja. *auditd* es el componente en el espacio de usuario del Sistema de Auditoría de Linux. Se encarga de escribir registros de auditoría en disco. Está formado por varias herramientas y ficheros para configurarlo al gusto. A continuación, se realiza una pequeña descripción de cada componente de *auditd*. Para ver toda la información de cada uno de ellos se puede utilizar el comando *man* de Linux.

auditd Es el proceso demonio. Tiene algunas opciones de configuración directas, como `-f` para poner el proceso en primer plano.

auditd.conf Fichero de configuración del demonio de auditoría. El archivo `/etc/audit/auditd.conf` contiene información de configuración específica para el demonio de auditoría. Cada línea debe tener una clave de configuración, un signo de igual, y a continuación la información de configuración apropiada para la clave.

Por ejemplo, la clave *log_file* especifica el nombre de la ruta completa del fichero donde se almacenarán los registros de auditoría.

ausearch Es una herramienta para realizar consultas en los registros de auditoría (también llamados *logs*). Se pueden realizar consultas con diferentes criterios de búsqueda, como por ejemplo el *UID* del usuario.

aureport Es una herramienta que produce informes resumidos de los *logs*.

auditctl Es un programa que se usa para configurar las opciones del *kernel* relacionados con la auditoría, para ver el estado de la configuración y para cargar reglas de auditoría discretionales.

audit.rules Es un fichero que contiene las reglas de auditoría que serán cargadas cuando el demonio se pone en marcha. *auditctl* lee de *audit.rules* las reglas durante el inicio y las carga en el *kernel*.

augenrules Es un *scrypt* que fusiona todos los ficheros de reglas de auditorías en un único fichero. Concretamente, coge todos los ficheros de reglas del directorio */etc/audit/rules.d*, los agrupa, y escribe el resultado en */etc/audit/audit.rules*. Todos los ficheros de reglas deben acabar en *.rules* para ser procesados por *augenrules*. Cualquier otro fichero contenido en */etc/audit/rules.d* será ignorado.

audispd Es un multiplexor de eventos. Tiene que ser iniciado por el demonio de auditoría para captar eventos. Captura los eventos y los distribuye a programas que quieren analizar eventos en tiempo real.

audispd.conf Es el fichero de configuración de *audispd*. Funciona de manera similar a *auditd.conf*.

A pesar de que aprender en detalle cómo funcionan cada uno de los componentes anteriores puede parecer una tarea pesada, lo cierto es que es bastante sencillo. Con una sola regla se puede configurar al demonio para que capture todas las *syscalls* de un usuario: «\$ *auditctl -a always,exit -S all -F uid=juan*». Y con esto, ya se tendría al demonio de *auditd* capturando y registrando todas las llamadas al sistema que hace el usuario *juan*.

No obstante, de nuevo se encontraron problemas de rendimiento. Sin llegar a colapsar, el sistema se ralentizaba de la misma forma que con la herramienta *sysdig*. Por este motivo, se tomó la decisión de que la aplicación debía ser menos exigente a la hora de monitorizar las llamadas al sistema con el objetivo de no afectar tanto al rendimiento. Para ello, el agente recibirá como argumento las llamadas al sistema que debe monitorizar. Se explicará esto con más detalle en el apartado de implementación correspondiente.

3.2. Diseño del subsistema de Ocultación

Al igual que en el diseño del subsistema de Monitorización, este subsistema se ha dividido en tres bloques diferenciados: la ocultación de los procesos de monitorización, la ocultación del espacio de memoria donde se archivarán todos los registros de seguimiento, que en este caso será un directorio, y la ocultación del puerto que utilizará el agente para comunicarse con el director (ver figura 3.3).

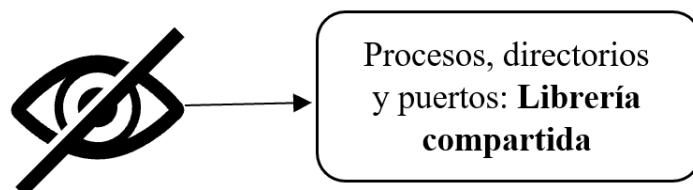


Figura 3.3: Descomposición del subsistema de ocultación. Imagen creada con Microsoft Word.

3.2.1. Ocultación de procesos en Linux

El objetivo es que el usuario intruso no sea capaz de detectar fácilmente los procesos que están registrando su actividad. En particular, se pretende ocultar estos procesos a los comandos tradicionales de Linux que realizan un chequeo de los procesos que se están ejecutando en el sistema, como son *ps*, *top*, *lsof*, etc. Para poder llegar a ocultar información a estos comandos es necesario entender cómo funcionan.

El directorio */proc* [13] [14] [15]

En las máquinas Unix, casi todo se representa como un descriptor de archivo: ficheros, directorios, conexiones de red, tuberías, eventos, y mucho más. Como consecuencia, comandos como *lsof* o *ps* pueden ver mucha información y pueden ser usados para resolver muchas cuestiones. Por ejemplo:

- ¿Qué usuario ha ejecutado cierto proceso?
- ¿Qué conexiones de red tiene el proceso X?
- ¿Qué procesos están consumiendo más recursos de la CPU?

Para ello, estos comandos hacen uso de una propiedad importantísima del kernel de Linux: el directorio */proc*. El kernel exporta una gran cantidad de información a este directorio, que puede ser listado con *ls* y manipulado con cualquier editor de texto. Si se lista el contenido del directorio */proc* se pueden ver varios subdirectorios cuyos nombres son números. Estos números son los *PID* de los procesos en ejecución. Por lo tanto, para saber qué procesos están en ejecución, simplemente hay que listar las entradas del directorio */proc*, que es precisamente lo que hace el comando *ps*. Dentro de los subdirectorios se encuentra toda la información referente a cada proceso asociado. Entre otras cosas,

se encuentran las listas de los descriptores de fichero asignados a cada proceso. En este punto, es fácil entender qué hace el comando *lsof* (ver figura 3.4):

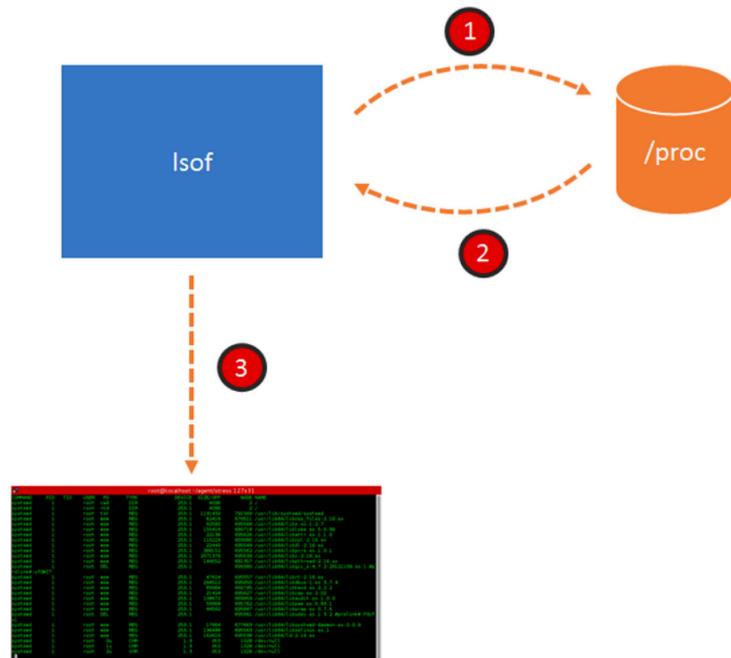


Figura 3.4: Esquema que resume el funcionamiento del comando *lsof* de Linux. Imagen tomada de <https://sysdig.com/blog/ps-lsof-netstat-time-travel/>

- 1.– Accede al directorio */proc*.
- 2.– Recolecta toda la información sobre los descriptores de ficheros asignados.
- 3.– Muestra esta información por la consola.

Muchos otros comandos de Linux (*ps*, *top*, *netstat*, y más) funcionan de la misma forma y siguen el esquema de la figura 3.4. Se podría decir que todos ellos constituyen una interfaz de usuario para extraer la información del directorio */proc*.

El funcionamiento de *ps* en detalle [14]

Para estudiar en detalle cómo funcionan estos comandos, se ha escogido el comando *ps* para su análisis. Aunque se podría haber escogido cualquier otro, ya que todos funcionan de manera similar. El objetivo es comprobar qué llamadas al sistema realiza este comando y a qué ficheros del directorio */proc* accede. Para averiguarlo, se ha hecho uso de la herramienta *strace* de Linux. Esta utilidad muestra todas las llamadas al sistema del proceso o comando que se le pase como argumento. También se podría usar la herramienta *sysdig*: el formato de salida habría sido diferente, pero la lectura sería la misma. Los resultados se muestran en la figura 3.5.

Primero, se abre el directorio */proc* mediante la llamada al sistema *openat()*. Después, el proceso realiza en el directorio abierto la llamada *getdents()*, que es una *syscall* que devuelve un listado de

```

openat(AT_FDCWD, "/proc/meminfo", O_RDONLY) = 4
lseek(4, 0, SEEK_SET)                      = 0
read(4, "MemTotal:      8056484 kB\nMemF...", 8191) = 1419
stat('/proc/self/task', {st_mode=S_IFDIR|0555, st_size=0, ...}) = 0
openat(AT_FDCWD, "/proc", O_RDONLY|O_NONBLOCK|O_CLOEXEC|O_DIRECTORY) = 5
fstat(5, {st_mode=S_IFDIR|0555, st_size=0, ...}) = 0
mmap(NULL, 135168, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7fdbf3dbd000
mmap(NULL, 135168, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7fdbf3d9c000
getdents(5, /* 317 entries */, 32768) = 8440
stat("/proc/1", {st_mode=S_IFDIR|0555, st_size=0, ...}) = 0
openat(AT_FDCWD, "/proc/1/stat", O_RDONLY) = 6
read(6, "1 (systemd) S 0 1 1 0 -1 4194560...", 1024) = 196
close(6)                                     = 0
openat(AT_FDCWD, "/proc/1/status", O_RDONLY) = 6
read(6, "Name:\tsystemd\nUmask:\t0000\nState:'..., 1024) = 1024
read(6, "00000000,00000000,00000000,00000000'..., 1024) = 275
close(6)                                     = 0
stat("/proc/2", {st_mode=S_IFDIR|0555, st_size=0, ...}) = 0
openat(AT_FDCWD, "/proc/2/stat", O_RDONLY) = 6
read(6, "2 (kthreadd) S 0 0 0 0 -1 212998'..., 2048) = 150
close(6)                                     = 0
openat(AT_FDCWD, "/proc/2/status", O_RDONLY) = 6
read(6, "Name:\tkthreadd\nUmask:\t0000\nState'..., 2048) = 941
close(6)                                     = 0
stat("/proc/3", {st_mode=S_IFDIR|0555, st_size=0, ...}) = 0
openat(AT_FDCWD, "/proc/3/stat", O_RDONLY) = 6
read(6, "3 (rcu_gp) I 2 0 0 0 -1 6923880'..., 2048) = 151
close(6)                                     = 0
openat(AT_FDCWD, "/proc/3/status", O_RDONLY) = 6
read(6, "Name:\trcu_gp\nUmask:\t0000\nState:\t'..., 2048) = 933
close(6)                                     = 0
stat("/proc/4", {st_mode=S_IFDIR|0555, st_size=0, ...}) = 0
openat(AT_FDCWD, "/proc/4/stat", O_RDONLY) = 6
read(6, "4 (rcu_par_gp) I 2 0 0 0 -1 6923'..., 2048) = 155
close(6)                                     = 0
openat(AT_FDCWD, "/proc/4/status", O_RDONLY) = 6
read(6, "Name:\trcu_par_gp\nUmask:\t0000\nSta'..., 2048) = 937
close(6)                                     = 0
stat("/proc/9", {st_mode=S_IFDIR|0555, st_size=0, ...}) = 0
openat(AT_FDCWD, "/proc/9/stat", O_RDONLY) = 6
read(6, "9 (mm_percpu_wq) I 2 0 0 0 -1 69'..., 2048) = 157
close(6)                                     = 0
openat(AT_FDCWD, "/proc/9/status", O_RDONLY) = 6
read(6, "Name:\tm_percpu_wq\nUmask:\t0000\nS'..., 2048) = 939
close(6)                                     = 0

```

Figura 3.5: Parte de la salida que resulta al ejecutar el comando *strace ps*.

los ficheros y directorios contenidos en un directorio concreto, que en este caso es `/proc`. En esta lista obtenida por `getdents()`, `ps` iterará sobre un par de ficheros en cada subdirectorio. Estos ficheros, como se puede leer en la figura 3.5, son `/proc/PID/stat` y `/proc/PID/status`. Por tanto, toda la información que muestra `ps` de cada proceso está contenida en estos ficheros.

Es importante saber que `ps` no llama directamente a `getdents()` y `openat()`, ya que estas son llamadas al sistema abstractas por la biblioteca estándar de C: `libc`. Esta biblioteca proporciona dos funciones, `opendir()` y `readdir()`, y son estas funciones las que llaman respectivamente a las `syscalls` anteriores, proporcionando una API algo más simple para el desarrollador. Por lo tanto, son estas últimas funciones las que son llamadas directamente por el comando `ps`. En la figura 3.6 se muestra un esquema que describe esta cadena de llamadas.

```
ps → readdir( ) [libc] → openat( )
ps → opendir( ) [libc] → getdents( )
```

Figura 3.6: Esquema que resume las cadenas de llamadas a `openat()` y `getdents()` que se producen con el comando `ps`

Ocultando el proceso [14] [16]

Después de esta pequeña explicación sobre cómo funciona el comando `ps`, parece obvio que si se pretende esconder un proceso, es necesario encontrar alguna forma de ocultar los ficheros implicados del directorio `/proc`. Observando el esquema mostrado en la figura 3.6, se pueden deducir las siguientes posibilidades:

- 1.– **Modificar los ficheros binarios de `ps`, `lsof`, `top`, etc.** Se podría modificar el código fuente de cada uno de estos comandos, añadiendo la lógica necesaria para ocultar procesos. Una vez hecho esto, se compilarían y se sustituirían los archivos binarios. Pese a ser efectivo, sería un procedimiento muy inefficiente y lento.
- 2.– **Modificar `libc`.** Se podría modificar la función `readdir()` de `libc` e introducir el código necesario para impedir el acceso a ciertos ficheros de `/proc`. Sin embargo, recompilar `libc` es un lastre enorme, además de que el código que compone esta biblioteca es bastante difícil de entender.
- 3.– **Modificar la llamada al sistema en el kernel de Linux.** Esta sería la opción más avanzada. Funcionaría interceptando y modificando la llamada `getdents()` directamente en el kernel mediante la creación de un módulo personalizado. A pesar de que funcionaría, esta opción se considera demasiado compleja y no cumpliría con el principio *KISS*.

Además de estas opciones para ocultar archivos de `/proc`, se encontró una cuarta alternativa. Esta consistiría en hacer uso de un *framework* apropiado, como por ejemplo *SELinux* o *Grsecurity*. Si bien podría haber sido una alternativa viable, no se profundizó más por esta vía.

La solución que se propone consiste en aprovecharse de un enlazador dinámico de Linux (*linker* en inglés) llamado *preloading*. Este se encarga de cargar en memoria todas las bibliotecas necesarias para

la ejecución de los procesos. Gracias al *preloading*, en Linux es posible cargar una librería compartida personalizada antes de que las otras librerías normales del sistema sean cargadas. Esto permite sobrescribir funciones de las librerías del sistema, creando funciones con el mismo nombre en las librerías compartidas personalizadas. Como consecuencia, todos los procesos ejecutarán el código de la nueva librería compartida. Simplemente habría que sobrescribir *readdir()* para que no se lea cierta información de */proc*, y de esta forma conseguir esconder un proceso. La forma más sencilla es esconder todo el subdirectorio de */proc* asociado al proceso que se quiera tapar. De esta manera, ninguno de los comandos nombrados anteriormente será capaz de detectar el proceso oculto. [17]

En la fase de Implementación se explicarán todos los detalles: cómo sobrescribir la función *readdir()* para esconder un subdirectorio de */proc*, cómo crear la librería compartida personalizada, y cómo decirle al *preloading* que la use.

3.2.2. Ocultación de datos en Linux

Una vez visto cómo ocultar un proceso en Linux, se busca una solución semejante para ocultar datos, ficheros e información. En particular, lo que se pretende es crear un espacio de disco que sea invisible para el usuario intruso. En este espacio se almacenarán todas las evidencias de monitorización que recolecten los agentes de la plataforma de engaño.

El ocultamiento de datos en informática abarca multitud de disciplinas, desde la criptografía a la esteganografía. Se muestran a continuación todas las propuestas que surgieron en relación a esta parte del diseño de la aplicación.

Directarios ocultos [18]

Este es uno de los métodos más básicos para ocultar datos. Se basa en el no descubrimiento del directorio que contiene los datos. Los archivos que contienen los datos no se disfrazan de ninguna manera, todo el esfuerzo se centra únicamente en esconder el directorio. Hay dos enfoques para lograr esto. El primer enfoque se centra en darle al directorio un nombre extraño para que pase desapercibido en los listados de ficheros, mientras que el segundo implica crear el directorio en un lugar del sistema donde sea poco probable que un usuario lo encuentre.

Con respecto al primer enfoque, existen varios nombres de directorios que los ciberdelincuentes utilizan con frecuencia para esconder información. Por ejemplo «...» (tres puntos) y «.. » (dos puntos y un espacio). En un listado normal, si se usa el comando «*ls -l*», estos dos directorios no aparecerán ya que ambos tienen nombres que empiezan con un punto. En un listado completo, con «*ls -a*», sí aparecen pero pueden llegar a pasar desapercibidos debido a su similitud con los directorios actual y padre, representados por un solo punto y dos puntos respectivamente.

En cuanto al segundo enfoque, se podría crear el directorio en algún sector del directorio */etc*. Este

es uno de los lugares más usados para el ocultamiento de un directorio, ya que puede pasar inadvertido entre los cientos de ficheros y directorios que hay dentro de `/etc`.

Archivos camuflados [18]

Este es otro método básico para encubrir datos, que se centra exclusivamente en el nombre del fichero. Consiste en dar nombres que camuflen los ficheros y los hagan pasar desapercibidos. Concretamente, se modifica la extensión del fichero; por ejemplo, cambiar las extensiones de archivos con contenido audiovisual por un «`.doc`». Sin embargo, Linux proporciona un comando llamado `file` que puede ser usado para determinar el tipo de fichero. Por lo tanto, este método no es muy útil para el objetivo establecido.

Cifrado con negación plausible o *plausible deniability* [19]

Ya existen leyes en varios países que obligan a descifrar los archivos encriptados, suponiendo la negativa una severa pena de prisión en algunos casos. La mejor protección, a veces, es no tener datos cifrados visibles. Para ello, algunas herramientas permiten ocultar discretamente datos detrás de otros archivos. Sin embargo, cuando se tienen muchos archivos que ocultar, o su tamaño es demasiado grande, la tarea de esconder la información detrás de otros ficheros se complica considerablemente. En este caso, sería necesario hacer uso de sistemas de ficheros esteganográficos. Estos sistemas van un paso más allá: permiten crear capas de ocultación en una sola partición de disco, con claves diferentes para cada una de las capas. De esta forma, cuando una autoridad obligue a dar la contraseña para una partición, el usuario puede dar una que dé acceso a datos inservibles o irrelevantes. En las capas pueden ocultarse incluso sistemas operativos enteros. Cada capa no conoce las demás capas, por lo que es imposible demostrar su existencia; de ahí lo de negación plausible. Una herramienta conocida que permitiría este tipo de cifrado es *VeraCrypt*; sin embargo, se ha visto que la negación plausible de *VeraCrypt* es relativamente fácil de detectar [20]. Se busca, por tanto, otra alternativa que, a ser posible, no involucre herramientas más allá de las que aporta el sistema operativo Linux.

De vuelta a la librería compartida

La solución propuesta consiste en reutilizar la librería compartida con la que se ocultarán los procesos. Si se puede ocultar un subdirectorio de `/proc`, se puede ocultar cualquier directorio del sistema de ficheros. En particular, se creará un directorio oculto con un nombre encabezado por un punto y camuflado en algún sector de `/etc`. Además, el administrador impedirá el acceso a este directorio mediante «`chmod 600 /...`». Con el fin de que el usuario intruso no sea capaz de encontrarlo, se añadirá la lógica necesaria a la nueva versión de la función `readdir()` para que el directorio sea invisible a comandos como `ls -a`.

En la fase de Implementación se explicarán todas las modificaciones que habrá que añadir a la

librería compartida para que también oculte el directorio de recolección de evidencias.

3.2.3. Ocultación de puertos en Linux

Además de ocultar los procesos y el directorio donde el agente almacenará las evidencias que registre, se pretende también ocultar el puerto que el propio agente de engaño utilizará para comunicarse con el director. A estas alturas la intención es utilizar el mismo método que en las subsecciones anteriores: hacer uso de la librería compartida.

Como ya se ha dicho, en los sistemas *Unix* todo se representa como un descriptor de archivo. En la subsección 3.2.1 de ocultación de procesos, se planteó la solución de sobrescribir la función *readdir()* para filtrar el subdirectorio */proc* asociado al proceso que se quería ocultar. De esta forma, comandos como *ps* o *lsof* son incapaces de mostrar información del proceso oculto, ya que para ellos no existe. Es por ello que se ha planteado el mismo método para ocultar el directorio de recolección de evidencias.

Hasta ahora solo ha sido necesario sobrescribir la función *readdir()*. La solución propuesta para ocultar puertos es parecida, salvo que ahora es otra función de *libc* la que debe ser sobrescrita.

Comandos para comprobar los puertos en Linux: *netstat*, *ss*, etc...

De la misma forma que se hizo con el comando *ps*, se procede a realizar un pequeño análisis de alguno de los comandos que muestra información relativa a los puertos en Linux.

Es de suponer que la conexión entre agente (cliente) y director (servidor) será una conexión *TCP* encapsulada con algún tipo de protocolo cifrado. El objetivo es encontrar la forma de filtrar esa conexión para que no aparezca en los listados de comandos como *netstat* o *ss*. Para ello es necesario ver qué ficheros del directorio */proc* son consultados por estos comandos. Así pues, se vuelve a hacer uso del comando *strace* de Linux para ver el comportamiento de *netstat*.

En este caso *strace* devuelve muchísima información para ser analizada. Una forma de filtrarla es volcar la salida a un fichero con «*strace -o hola.txt netstat*» y luego utilizar el comando *grep* para mostrar sólo lo relevante a conexiones *TCP*: «*cat hola.txt | grep tcp*». En la figura 3.7, se muestra el resultado del comando *grep*.

Como se puede ver en la figura 3.7, el único fichero que consulta el comando *netstat* para ver las conexiones *TCP* es el fichero */proc/net/tcp*. De hecho si se accede a este fichero, aparece un lista en hexadecimal de todas las conexiones *TCP* que existen en ese momento (ver figura 3.8).

Llegado este punto, la tarea que se debe realizar parece sencilla. Habría que sobrescribir la función *read()* de *libc* en la librería compartida. De tal forma que, cuando se lea el fichero */proc/net/tcp*, se filtre la conexión establecida con el director. Se podría, o bien filtrar la fila entera correspondiente a dicha

```
javier@javier-Lenovo-YOGA-900-13ISK:~$ cat hola.txt | grep tcp
openat(AT_FDCWD, "/proc/net/tcp", O_RDONLY) = 3
read(4, "1/tcp\t\nclearcase\nclearcase\t371/u"..., 4096) = 4096
read(4, "\t1812/tcp\nradius\t\t1812/udp\nradius"..., 4096) = 4096
read(4, "/tcp\t\t\t# french minitel\nxotelw\t\t1"..., 4096) = 2799
write(1, "tcp      0      0 javier-Lenov"..., 80) = 80
openat(AT_FDCWD, "/proc/net/tcp6", O_RDONLY) = 3
javier@javier-Lenovo-YOGA-900-13ISK:~$ 
```

Figura 3.7: Información relativa a conexiones TCP que muestra strace sobre el comando netstat

```
javier@javier-Lenovo-YOGA-900-13ISK:~$ cat /proc/net/tcp
sl local_address rem_address   st tx_queue rx_queue tr tm->when retrnsmt  uid  timeout inode
0: 3500007F:0035 00000000:0000 0A 00000000:00000000 00:00000000 00000000 101    0 63335 1 0000000000000000 100 0 0 10 0
1: 0100007F:0277 00000000:0000 0A 00000000:00000000 00:00000000 00000000 0       0 228782 1 0000000000000000 100 0 0 10 0
2: 3001A8C0:AF98 5561C923:01BB 01 00000000:00000000 02:000001B3 00000000 1000   0 239066 2 0000000000000000 22 4 30 10 -1
3: 3001A8C0:D756 99E84922:01BB 01 00000000:00000000 02:00000346 00000000 1000   0 226696 2 0000000000000000 36 4 31 10 -1
4: 3001A8C0:99A8 BC857D4A:146C 01 00000000:00000000 02:00000A9E 00000000 1000   0 208895 2 0000000000000000 23 4 2 10 -1
5: 3001A8C0:AF92 5561C923:01BB 01 00000000:00000000 02:0000017B 00000000 1000   0 237505 2 0000000000000000 21 4 30 10 -1
6: 3001A8C0:D75C 99E84922:01BB 01 00000000:00000000 02:000008DF 00000000 1000   0 229475 2 0000000000000000 38 4 24 10 -1
7: 3001A8C0:ED60 542ED382:01BB 01 00000000:00000000 02:0000029E 00000000 1000   0 238910 2 0000000000000000 20 4 0 10 -1
8: 3001A8C0:AF9C 5561C923:01BB 06 00000000:00000000 03:00000922 00000000 0       0 0 3 0000000000000000
9: 3001A8C0:D7F8 99E84922:01BB 01 00000000:00000000 02:00000893 00000000 1000   0 237682 2 0000000000000000 36 4 26 10 40
10: 3001A8C0:D154 2201A8C0:1F49 01 00000000:00000000 02:0000031D 00000000 1000   0 208666 2 0000000000000000 22 4 30 10 -1
11: 3001A8C0:AF96 5561C923:01BB 01 00000000:00000000 02:000001B3 00000000 1000   0 239065 2 0000000000000000 22 4 26 10 -1
12: 3001A8C0:9370 0281459F:01BB 01 00000000:00000000 02:00000F6A 00000000 1000   0 238926 2 0000000000000000 24 4 4 10 -1
javier@javier-Lenovo-YOGA-900-13ISK:~$ 
```

Figura 3.8: Contenido del fichero /proc/net/tcp donde aparece un listado en hexadecimal de todas las conexiones TCP del sistema.

conexión, o bien modificar la información de la misma (el puerto o la dirección remota) para que *netstat* muestre información falsa.

Sin embargo, es importante darse cuenta de que el directorio */proc/net/* es un enlace simbólico. Esto se puede comprobar fácilmente ejecutando «*ls -la /proc/net*». En la figura 3.9 se puede ver que es un enlace a */self/net*.

```
javier@javier-Lenovo-YOGA-900-13ISK:~$ ls -la /proc/net
lrwxrwxrwx 1 root root 8 feb 20 09:30 /proc/net -> self/net
javier@javier-Lenovo-YOGA-900-13ISK:~$ 
javier@javier-Lenovo-YOGA-900-13ISK:~$ 
```

Figura 3.9: */proc/net* es un enlace simbólico a */self/net*

```
javier@javier-Lenovo-YOGA-900-13ISK:~$ ls -ls /proc/self
0 lrwxrwxrwx 1 root root 0 feb 20 11:42 /proc/self -> 7217
javier@javier-Lenovo-YOGA-900-13ISK:~$ ls -ls /proc/self
0 lrwxrwxrwx 1 root root 0 feb 20 11:42 /proc/self -> 7218
javier@javier-Lenovo-YOGA-900-13ISK:~$ ls -ls /proc/self
0 lrwxrwxrwx 1 root root 0 feb 20 11:42 /proc/self -> 7219
javier@javier-Lenovo-YOGA-900-13ISK:~$ ls -ls /proc/self
0 lrwxrwxrwx 1 root root 0 feb 20 11:42 /proc/self -> 7220
javier@javier-Lenovo-YOGA-900-13ISK:~$ ls -ls /proc/self
0 lrwxrwxrwx 1 root root 0 feb 20 11:42 /proc/self -> 7221
javier@javier-Lenovo-YOGA-900-13ISK:~$ ls -ls /proc/self
0 lrwxrwxrwx 1 root root 0 feb 20 11:42 /proc/self -> 7222
javier@javier-Lenovo-YOGA-900-13ISK:~$ ls -ls /proc/self
0 lrwxrwxrwx 1 root root 0 feb 20 11:42 /proc/self -> 7223
javier@javier-Lenovo-YOGA-900-13ISK:~$ ls -ls /proc/self
0 lrwxrwxrwx 1 root root 0 feb 20 11:42 /proc/self -> 7224
javier@javier-Lenovo-YOGA-900-13ISK:~$ ls -ls /proc/self
0 lrwxrwxrwx 1 root root 0 feb 20 11:42 /proc/self -> 7225
javier@javier-Lenovo-YOGA-900-13ISK:~$ 
```

Figura 3.10: */proc/self* es un enlace simbólico a */proc/PID*

Si se vuelve a realizar la misma comprobación, se puede ver que */proc/self* también es un enlace simbólico. En la figura 3.10, se ve que es un enlace a uno de los subdirectorios de */proc* asociado a un proceso. Y vemos que, si se ejecuta «*ls -la /proc/self*» varias veces seguidas, el enlace cambia y el *PID* del proceso asociado se va incrementando en uno. Esto quiere decir que */proc/self* es un enlace al proceso que se está ejecutando en ese momento. Por lo tanto, a lo que realmente está apuntando */proc/net/tcp* es a */proc/PID/net/tcp*, donde *PID* es el identificador del proceso que está en ejecución. A modo de resumen, en la figura 3.11 se puede ver la cadena de enlaces simbólicos que se sigue. Es importante tener esto en cuenta a la hora de sobrescribir la función *read()* de *libc*.

/proc/net/tcp → */proc/self/net/tcp* → */proc/PID/net/tcp*

Figura 3.11: Cadena de enlaces simbólicos que va desde */proc/net/tcp* a */proc/PID/net/tcp* donde *PID* es el identificador del proceso en ejecución.

Hasta aquí llegaría el diseño del subsistema de ocultación. Para resumir, se va a crear una librería

compartida donde se sobrescribirán dos funciones de *libc*: *readdir()* y *read()*. La nueva versión de *readdir()* ocultará los subdirectorios de */proc* asociados a los procesos de monitorización y también el directorio de recolección de evidencias. Y la nueva versión de *read()* ocultará la conexión TCP establecida con el director de engaño. En la fase de Implementación se mostrarán los detalles del código desarrollado.

3.3. Diseño del subsistema de Comunicación

Para el diseño del subsistema de comunicación, se presenta un único reto: establecer una comunicación segura, y por tanto cifrada, entre el director y el agente (ver figura 3.12).

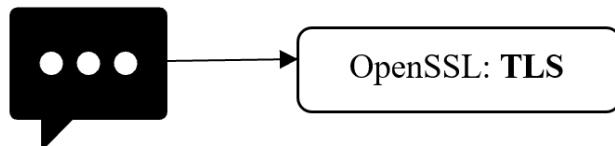


Figura 3.12: Descomposición del subsistema de comunicación. Imagen creada con *Microsoft Word*.

3.3.1. Comunicación cifrada entre director y agente: Protocolo *TLS*

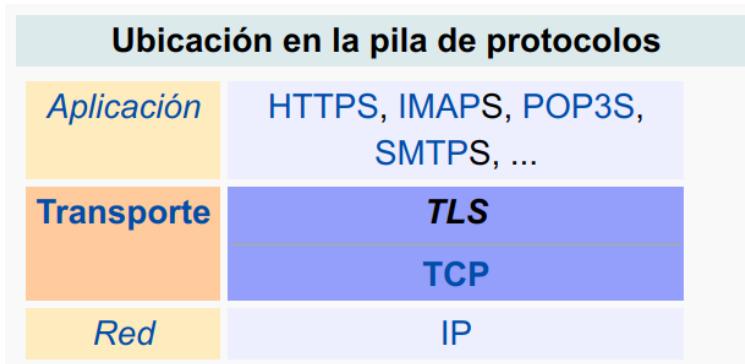


Figura 3.13: Ubicación de TLS en la pila de protocolos. Imagen tomada de https://es.wikipedia.org/wiki/Seguridad_de_la_capa_de_transporte

Transport Layer Security (TLS), en español significa seguridad en la capa de transporte, y su antecesor *Secure Sockets Layer (SSL)*, en español significa capa de puertos seguros, son protocolos criptográficos que proporcionan comunicaciones seguras para cualquier red. En la figura 3.13 se muestra la ubicación del protocolo *TLS* en la pila de protocolos.

TLS utiliza los certificados X.509. Estos proporcionan criptografía asimétrica para intercambiar una

lave simétrica y para autenticar a las partes que se están comunicando. Se utiliza esta sesión para cifrar el flujo de información entre los pares; permitiendo confidencialidad del mensaje, códigos de autenticación para la integridad de los mensajes, y además, autenticación del mensaje.

En *Python* existe la librería *cryptography* que incluye interfaces a bajo nivel para algoritmos criptográficos comunes, además del estándar X.509. Se ha hecho uso de esta librería para cifrar la comunicación entre agente y director, creando un certificado auto-firmado únicamente para el director (que será el servidor). En la parte de implementación se verán los detalles del código desarrollado. [21]

IMPLEMENTACIÓN

En este apartado, se muestran y explican los detalles y problemas que surgieron durante el desarrollo y la implementación del sistema de engaño. Como ya se ha indicado, la estructura de la aplicación consta de dos partes bien diferenciadas: el agente y el director. El agente implementa la mayor parte de la funcionalidad, incluyendo los subsistemas de monitorización, ocultación y la mitad del subsistema de comunicación. Por otra parte, el director solamente implementa la mitad restante del subsistema de comunicación.

4.1. Implementación del Agente de engaño

Desde el inicio, la intención fue crear un único programa en *Python* para el agente. Se escogió el lenguaje *Python* por su sencillez, siguiendo el principio KISS . Se verá, punto por punto, cómo se resolvió cada funcionalidad de cada subsistema por separado y se mostrará la versión final del agente con toda la funcionalidad incorporada.

4.1.1. Implementación subsistema de monitorización

Monitorización del sistema de ficheros

Como se indicó en el apartado de diseño, para monitorizar los cambios que realice el usuario intruso, se ha hecho uso de la librería *python-inotify*, que se puede instalar con «*sudo pip install inotify*». Esta funcionalidad se puede ver implementada en el apéndice A, a partir de la línea 242. Para cualquier cadena de eventos asociada a una acción, el agente llevará a cabo la respuesta correspondiente. La traducción de cadenas de eventos a respuestas del subsistema de monitorización se mostró en la tabla 3.1.

Se muestra por pantalla cada evento capturado. En la versión final, estos eventos serán transmitidos al director. La variable *hiddenD* es la ubicación del directorio oculto donde se almacenarán todas las evidencias de monitorización del usuario. La ubicación elegida es «*/etc/dpkg/origins/...*». Este directorio

debe crearlo el agente antes de ejecutarse el manejador de eventos *inotify* y debe permitir su acceso únicamente al administrador con «*chmod 600 /etc/dpkg/origins/...*». El motivo por el que se ha llamado al directorio de recolección de evidencias «...» (tres puntos) se explicó en la subsección de diseño 3.2.2 y es para que pudiera pasar desapercibido en el caso de que el usuario intruso realizase un listado del directorio */etc/dpkg/origins*. Aunque en este caso sería inútil, debido a que no podrá verlo, ya que este directorio se ocultará con la librería compartida. Y, aunque consiguiera saber de su existencia, no podría acceder a él porque no tendría los permisos de acceso.

Monitorización de las llamadas al sistema

Después de probar varias opciones, se optó por usar el demonio de auditoría de Linux para monitorizar llamadas al sistema. Aunque en este trabajo sólo se use para registrar las *syscalls* que efectúe un usuario, esta herramienta permite hacer un seguimiento de muchas otras cosas. Una vez se ha instalado, con «*sudo apt install auditd*», es necesario configurarla. Para ello, solamente hay que hacer dos cosas: cambiar el fichero de salida donde *auditd* escribe los registros y añadir las reglas necesarias para que el demonio monitorice las llamadas al sistema del usuario que se deseé.

Por defecto, *auditd* escribe todo en */var/log/audit/audit.log*, pretendiéndose que dicho fichero se encuentre en otra ubicación diferente. Cambiar la configuración de *auditd* implica retocar el fichero */etc/audit/auditd.conf*. En este archivo, se puede modificar la ubicación del fichero de registros modificando la opción *log_file*. Además de esta, en */etc/audit/auditd.conf* aparecen muchas otras opciones para poder configurar el demonio de *auditd* al gusto.

Por otro lado, para indicarle qué llamadas al sistema queremos que registre y de qué usuario, es necesario incluir una serie de reglas en el fichero */etc/audit/rules.d/audit.rules*. En este caso, todas las reglas tienen prácticamente la misma forma y se hace una regla por cada llamada al sistema que se quiere monitorizar. Se podría hacer una única regla que incluyera todas las *syscalls* pero, como ya se indicó en la subsección 3.1.2, esto afectaría negativamente al rendimiento de la máquina virtual, dificultando su normal funcionamiento. Por lo tanto, es el administrador el que debe introducir manualmente, cuando ejecute el agente, qué llamadas al sistema quiere monitorizar del usuario intruso. Acto seguido, el agente debe modificar el fichero de configuración *auditd.conf* e incluir las reglas necesarias en *audit.rules*. Todo se implementa en el apéndice A, entre las líneas 72 y 148.

Con esto, ya se tiene toda la parte correspondiente a la monitorización del usuario. La versión final del agente incluye tanto la parte *inotify* como la configuración de *auditd*. Es importante tener en cuenta que, aunque el agente configure *auditd* y lo ponga en marcha, el demonio de auditoría de Linux es un proceso aparte al manejador de eventos. Por lo tanto, dicho agente de engaño está formado por dos procesos: el manejador *inotify* y el demonio *auditd*. Y, como es lógico, el subsistema de ocultación debe conseguir hacer invisibles a ambos.

4.1.2. Implementación subsistema de ocultación

En esta sección se explica cómo crear la librería compartida para ocultar los procesos de monitorización, el directorio de recolección de evidencias y el puerto de comunicación con el director.

Como se puede suponer, es obligatorio escribir la librería compartida en lenguaje C. Será lo único de la aplicación que no estará escrito en *Python*. La idea es muy sencilla: sobreescibir las funciones *readdir()* y *read()* de *libc*. De tal forma que, cuando un proceso llame a estas funciones, se ejecutarán nuestras versiones en vez de las originales. Lo que estas versiones hacen es realizar una serie de comprobaciones y, si todas ellas resultan positivas, se aplicarán los filtros de ocultación del sistema de engaño.

Ocultación del proceso de monitorización

En el caso de la ocultación del proceso de monitorización, se llevan a cabo tres comprobaciones antes de aplicar los filtros. En los códigos 4.1 y 4.2 se muestra la primera versión modificada de *readdir()* que incluyó la librería compartida.

Código 4.1: En esta figura se presenta el código correspondiente a la primera versión de la función *readdir()* de la librería compartida. Parte I.

```

1  /*
2   *Every process with this name will be excluded
3   */
4  static const char* process_to_filter = "agente.py";
5
6  #define DECLARE__REaddir(dirent, readdir) \
7  static struct dirent* (*original_##readdir)(DIR*) = NULL; \
8                                     \
9  struct dirent* readdir(DIR *dirp) \
10 { \
11     if(original_##readdir == NULL) { \
12         original_##readdir = dlsym(RTLD_NEXT, #readdir); \
13         if(original_##readdir == NULL) \
14             { \
15                 fprintf(stderr, "Error in dlsym: %s\n", dlerror()); \
16             } \
17     } \
18                                     \
19     struct dirent* dir; \
20                                     \
21     while(1) \
22     { \
23         dir = original_##readdir(dirp); \
24         if(dir) { \
25             char dir_name[256]; \
26             char process_name[256]; \

```

Código 4.2: En esta figura se presenta el código correspondiente a la primera versión de la función *readdir()* de la librería compartida. Parte II.

```

27     if( is_sudo() == 0 && \
28         ((get_dir_name(dirp, dir_name, sizeof(dir_name)) && \
29          strcmp(dir_name, "/proc") == 0 && \
30          get_process_name(dir->d_name, process_name) && \
31          strcmp(process_name, process_to_filter) == 0) )) { \
32         continue; \
33     } \
34   } \
35   break; \
36 } \
37 return dir; \
38 }
39
40 DECLARE_REaddir(dirent64, readdir64);
41 DECLARE_Readdir(dirent, readdir);

```

La estructura de la función *readdir()* se puede consultar en el manual del programador de Linux con el comando «*man readdir*». En nuestra versión, la función *readdir()* se ha definido como una macro. Primero, se crea un puntero a función al que se le asigna la función *readdir()* original mediante la función *dlsym()*. Una vez se tiene la función original, se ejecuta y se guarda su salida en la variable *dir* que es una estructura *dirent** (del inglés *directory entry*).

Llegados a este punto, se llevan a cabo las tres comprobaciones. Primero, se comprueba con la función *is_sudo()* si el proceso que ha llamado a la función *readdir()* lo ha ejecutado el usuario administrador. En segundo lugar, con la ayuda de la función *get_dir_name()*, se comprueba si el directorio sobre el que se está haciendo *readdir()* es el directorio */proc*. Y tercero, se comprueba si el subdirectorío que va a devolver *readdir()*, esto es, lo que está contenido en la variable *dir*, está asociado al proceso agente. Es decir, se comprueba si el proceso asociado a ese subdirectorío es el proceso agente de engaño de nombre *agente.py*. Si las tres comprobaciones resultan positivas, entonces el bucle itera una vez más y no devuelve la variable *dir* para ese caso. De esta forma, se consigue filtrar el subdirectorío de */proc* asociado al proceso agente.

A continuación, vamos a ver por separado las funciones que llevan a cabo las tres comprobaciones anteriores. En primer lugar, se creó una función que permitiera averiguar si el proceso actual, es decir, el que está llamando a *readdir()*, lo ha lanzado el administrador. Para averiguar qué usuario ha lanzado un proceso en ejecución, simplemente hay que consultar el fichero */proc/PID/status*. En este fichero aparece información relativa al proceso, como su nombre, su *pid*, o el *uid* del usuario que lo puso en marcha. Por lo tanto, abrir este fichero y leer el identificador de usuario sería una forma de averiguar quién fue el causante del proceso. Sin embargo, hay una forma más sencilla. Como sólo interesa saber si el que lanzó el proceso fue el administrador, simplemente hay que intentar acceder a un fichero con

permisos *root*: si el acceso se lleva a cabo, entonces no hay que aplicar los filtros de ocultación. Para no andar accediendo a ficheros sensibles del sistema, el agente crea al principio un fichero vacío */etc/hola* con permisos *root* y es a este fichero al que se intenta acceder para ver si el proceso en ejecución lo lanzó el administrador. En el código 4.3 se muestra la función *is_sudo()*, que realiza este acceso a */etc/hola*. Si el acceso falla, se realizan el resto de comprobaciones.

Código 4.3: Código de la función *is_sudo()*: comprueba si el proceso actual lo ha lanzado el usuario administrador. Para ello intenta abrir el fichero */etc/hola* que tiene permisos *root*. Devuelve 1 si el acceso se lleva a cabo, sino devuelve 0.

```

1  /*
2   * Detect if uid is 0
3   */
4  static int is_sudo(){
5
6      FILE* f = fopen("/etc/hola", "r");
7      if(f == NULL) {
8          return 0;
9      }
10
11     fclose(f);
12     return 1;
13 }
```

Como ya se ha indicado, la segunda comprobación fue averiguar sobre qué directorio se está ejecutando la función *readdir()*. Para ello, se ha creado la función *get_dir_name()*. Esta función se muestra en el apéndice B, entre las líneas 34 y 50.

Una vez se tiene el nombre del directorio, se comprueba si este es */proc*. Si no es el caso, no se aplican los filtros. Pero si lo es, se llevaría a cabo la última comprobación. La tercera y última comprobación es ver si el proceso asociado al subdirectorio que pretende devolver la función *readdir()* original es el proceso agente de engaño *agente.py*. Para ello se ha creado la función *get_process_name()*, que obtiene el nombre del proceso a partir de su *pid*. Se puede ver esta función en el apéndice B, entre las líneas 55 y 79.

Una vez se tiene el nombre del proceso, se comprueba si este es *agente.py*. Si no es el agente de engaño, no se aplican los filtros. Pero si lo es, entonces se itera una vez más en el bucle para ignorar esta entrada del directorio */proc*. De esta forma, se filtra el proceso de monitorización, haciéndose invisible a los comandos *ps*, *lsof*, etc.

Ocultación del directorio de recolección de evidencias

Para ocultar el directorio de recolección de evidencias, se sigue el mismo método. Si se puede filtrar un subdirectorio de */proc*, se puede filtrar cualquier directorio del sistema de ficheros. Para ello,

es necesario fijar una ruta para el directorio de recolección de evidencias. En este caso, se escogió la ruta «`/etc/dpkg/origins/...`» (con tres puntos al final). Pese a que se pueden escoger infinidad de rutas, es importante que este directorio esté fuera del directorio `/home` del usuario monitorizado, ya que en caso contrario, se entraría en un bucle infinito: el agente registraría en el directorio de recolección de evidencias una acción del usuario, pero como este directorio también estaría dentro del directorio `/home`, `inotify` capturaría otro evento de modificación de fichero que el agente registraría de nuevo en el directorio de recolección de evidencias, que a su vez generaría otro evento con su correspondiente registro, y este a su vez otro... Y así hasta el infinito.

Las comprobaciones para decidir si filtrar el directorio son tres: comprobar si fue el administrador el que ejecutó el proceso, comprobar si se está haciendo `readdir()` sobre el directorio `/etc/dpkg/origins`, y comprobar si la entrada que devuelve la función `readdir()` original, esto es, el contenido de la variable `dir`, se corresponde con el subdirectorio «`...`». En el apéndice B se implementa la versión final de la función `readdir()`. En esta versión, se han incluido las comprobaciones necesarias para ocultar el directorio de evidencias. Se aprovecha la comprobación del usuario administrador que ya se hacía con la función `is_sudo()`. Para las otras dos comprobaciones simplemente se han añadido un par de condiciones más, siguiendo el ejemplo de cómo se hacía para el directorio `/proc`. Además de esto, se ha incluido la filtración del proceso de `auditd` de la misma forma a como se ha hecho con el propio proceso agente `agente.py`.

Compilación e instalación de la librería compartida

Una vez tenemos la librería con la versión modificada de `readdir()`, es hora de compilarla e instalarla en la ruta del sistema. Para compilar la librería compartida es necesario utilizar las etiquetas mostradas en el código 4.4.

Código 4.4: Fichero Makefile para la compilación de la librería compartida `libprocesshider.c` usando `gcc`.

```

1 all: libprocesshider.so
2
3 libprocesshider.so: processhider.c
4     gcc -Wall -fPIC -shared -o libprocesshider.so processhider.c -ldl
5
6 .PHONY clean:
7     rm -f libprocesshider.so

```

La etiqueta `-Wall` muestra todos los *warnings*. `-fPIC` (del inglés *Position-Independent Code*) es para indicar que se está generando código de posición independiente. `-shared` para indicar que es compartida. Y `-ldl` para enlazar nuestra librería con las otras librerías cargadas dinámicamente. Esta última etiqueta es obligatoria debido a que estamos usando la función `dlsym()` para obtener la versión `readdir()` original.

Ahora simplemente hay que instalar la librería en la ruta del sistema e indicarle al enlazar dinámico que la use. Queremos instalarla para todo el sistema, de tal forma que todos los nuevos procesos la utilicen de forma automática. Para ello, tan solo hace falta escribirla en el fichero de configuración del *preloading*, tal y como se muestra en el código 4.5. Se implementa como código *Python* para poder incluirlo en el fichero *agente.py*, y así poder recompilar la librería con un simple argumento para el agente de engaño.

Código 4.5: Compilación de la librería compartida, su instalación en la ruta del sistema y su escritura en el fichero de configuración del *preloading*

```

1 ######
2 #COMPILACION E INSERCIÓN LIBRERIA DINAMICA
3 os.system("make")
4 os.system("mv libprocesshider.so /usr/local/lib/")
5 os.system("echo /usr/local/lib/libprocesshider.so >> /etc/ld.so.preload")

```

Una vez hecho esto, si todo ha ido bien, desde este momento los procesos que entren en ejecución utilizarán la nueva versión de *readdir()*. Se ocultarán, a todo usuario excepto al administrador, los procesos llamados *agente.py* y *auditd*, así como el directorio ubicado en la ruta «*/etc/dpkg/origins/...*».

Ocultación del puerto de conexión con el Director

Para empezar con este apartado, es necesario advertir que no se ha conseguido implementar adecuadamente. Se trata del único requisito que no se ha conseguido incluir en la aplicación. A pesar de ello, se van a desarrollar todos los pasos realizados y a detallar el problema en cuestión que está sin resolver.

Como se explicó en la sección 3.2.3, es necesario añadir a la librería compartida una versión de la función *read()* que capture las lecturas del fichero */proc/PID/net/tcp*. Este fichero es el lugar desde el cual comandos como *netstat* o *ss* sacan toda la información relativa a las conexiones TCP del sistema. Para ver qué forma tiene la función *read()* puede consultarse el manual del programador de Linux. En el apéndice B, entre las líneas 169 y 200, se muestra la nueva versión incluida en la librería compartida.

Se sigue el mismo esquema que para *readdir()*. Se crea un puntero a función, y se le asigna la función *read()* original haciendo uso de *dlsym()*. En este caso, se hacen dos comprobaciones: la primera ya es conocida, se utiliza la función *is_sudo()* para comprobar que el usuario que ha lanzado el proceso es el usuario administrador; la segunda comprobación trata de hacer lo siguiente. La función *read()* recibe como argumento un descriptor de fichero *fd*. Se trata de un entero que se identifica con el archivo que se pretende leer. Por lo tanto, lo que se hace es traducir ese descriptor de fichero al nombre real del archivo y compararlo con */proc/PID/net/tcp*, donde *PID* es el identificador del propio proceso que ha llamado a *read()*. Si no coincide, se devuelve el resultado del *read()* original. Pero si coincide, se debe filtrar la conexión con el director en la salida.

Se sabe que esta versión de `read()` funciona. Se puede comprobar ejecutando «`cat /proc/net/tcp`» con el usuario monitorizado. Obviamente se debe recomilar la librería para incluir cualquier cambio. Lo que hace el comando `cat` simplemente es leer ese fichero e imprimir su contenido en la consola. Se puede ver que se ejecuta el código que está dentro del filtro. En este caso, es un simple `printf` que imprime el nombre del comando por la consola (ver figura 4.1).

```
tfg@tfg-VirtualBox:~$ cat /proc/net/tcp
Process name = cat
sl local_address rem_address st tx_queue rx_queue tr tm->when retrnsmt uid timeout inode
0: 3500007F:0035 00000000:0000 0A 00000000:00000000 00:00000000 00000000 101 0 15583 1 0000000000000000 100 0 0 10 0
1: 0100007F:0277 00000000:0000 0A 00000000:00000000 00:00000000 00000000 0 0 16964 1 0000000000000000 100 0 0 10 0
2: 0100007F:1388 00000000:0000 0A 00000000:00000000 00:00000000 00000000 1000 0 47155 1 0000000000000000 100 0 0 10 0
3: 0101007F:04D2 00000000:0000 0A 00000000:00000000 00:00000000 00000000 1000 0 47162 1 0000000000000000 100 0 0 10 0
4: 0101007F:04D2 0100007F:9AA8 01 00000000:00000000 00:00000000 00000000 1000 0 47163 1 0000000000000000 20 4 30 10 -1
5: 0F02000A:B342 27E3D522:01BB 01 00000000:00000000 02:00001F44 00000000 1000 0 32855 2 0000000000000000 46 4 29 10 -1
6: 0F02000A:916C DF29372D:0050 08 00000000:00000000 00:00000000 00000000 1000 0 29650 1 0000000000000000 33 4 26 10 -1
7: 0100007F:9AA8 0101007F:04D2 01 00000000:00000000 00:00000000 00000000 0 0 47728 1 0000000000000000 20 4 14 10 -1
Process name = cat
tfg@tfg-VirtualBox:~$
```

Figura 4.1: Ejecución del comando `cat` sobre el fichero `/proc/net/tcp` usando la nueva versión de `read()`. El comando `cat` ejecuta la función `read()` de la librería compartida e imprime el código que correspondería al filtro para ocultar el puerto de conexión con el director (en este caso es un `printf`).

Sin embargo, los comandos `netstat` y `ss` nunca llegan a ejecutar esta versión de `read()`. Se puede comprobar con `strace` que realizan la misma lectura del fichero `/proc/net/tcp` que el comando `cat`. Sin embargo, parece que `netstat` y `ss` son capaces de saltarse la librería compartida y acceder directamente a la función `read()` original. Se desconoce el motivo por el que estos comandos tienen este comportamiento. El problema no es que no se ejecute el código correspondiente al filtro, sino que no se ejecuta en ningún momento esta versión de `read()` de la librería compartida. Se puede ver esto colocando el mismo `printf` fuera del `if()`, de tal forma que siempre se ejecute. Si se guardan los cambios el resultado es que para todos los programas se imprimen los nombres de los mismos, menos para `netstat` y `ss` (seguramente ocurre para otros, pero estos son los únicos que nos interesan para filtrar el puerto).

Debido a que no funciona la ocultación del puerto, se ha incluido la función `read()` como un comentario en la librería compartida.

4.1.3. Implementación subsistema de comunicación (I)

Como se mencionó en el diseño del subsistema de comunicación, en la sección 3.3.1, se ha hecho uso de la librería `cryptography` de `Python` para cifrar la comunicación entre el director y el agente, utilizando el protocolo `TLS` y el estándar `X.509`.

En la mayoría de los casos, se quiere un certificado que haya sido firmado por una tercera parte, por ejemplo, una autoridad certificadora, que otorgue a dicho certificado un alto nivel de confianza. Sin embargo, a veces con un certificado auto-firmado es suficiente. Los certificados auto-firmados no son emitidos por una autoridad certificadora, sino que están firmados con la clave privada correspondiente a la clave pública que incorporan.

Esto tiene dos consecuencias. Por un lado, nadie confiará en dicho certificado auto-firmado. Pero por otro, estos certificados son mucho más fáciles de emitir, al no necesitar de una tercera parte. En general, el único caso de uso para estos certificados auto-firmados son las pruebas de software locales, en las que no se necesita que nadie confíe en el certificado. Es por esto que se ha creado un certificado auto-firmado para la comunicación del director y el agente.

En el apéndice C se muestra la creación de los certificados: el certificado raíz y el certificado del servidor, que será el director. Hacer un certificado para el cliente (el agente) no es necesario. El resultado de ejecutar sin argumentos este programa de *Python* son cuatro ficheros: la clave privada del certificado raíz (*key.pem*), el certificado raíz (*certificate.pem*), la clave privada del servidor (*server_key.pem*), y el certificado del servidor (*server_cert.pem*).

El director abre un *socket*, lo encapsula en una conexión *TLS* y espera a recibir conexiones. El agente inicializa una conexión *TLS* con el director y procede a mandarle todas las evidencias a través del *socket*. La conexión entre ambos debe ser antes del bucle de eventos para poder mandar las evidencias de monitorización del usuario.

Además de esto, es necesario mandar por la misma vía los datos registrados por *auditd*. Para ello, se ha ubicado la ruta del fichero de registro de *auditd* dentro del directorio */home* del usuario monitorizado. De tal forma que, cada vez que *auditd* modifique este fichero, registrando una nueva llamada al sistema, el proceso *inotify* detectará dicho cambio y capturará un evento. El agente trata los eventos de este fichero de registros de una forma distinta a como lo hace con los eventos del resto del sistema de ficheros, ya que no se pretende notificar de la inclusión de un nuevo registro de *auditd*, sino que se quiere mandar precisamente el contenido del registro. Igualmente, debido a que el formato *en crudo* de los registros de *auditd* son poco legibles, el agente hace uso de la herramienta *aureport* para sintetizar el contenido del nuevo registro. Es este registro abreviado de *aureport* el que manda el agente cuando *auditd* registra una nueva llamada al sistema. En el apéndice A se muestra el código final del agente de engaño, donde se incluyen estas últimas cuestiones y todas las anteriores. El código ha sido comentado para facilitar todo lo posible su comprensión.

Asimismo, se incluye en el Apéndice B la versión final de la librería compartida. Ésta filtra, mediante la función *readdir()*, todo lo que se pretende ocultar al usuario atacante: el proceso *inotify* del agente, el proceso *auditd* que también forma parte del agente, el directorio de recolección de evidencias y el fichero de registros de *auditd*. Se incluye como comentario todo lo relativo a la función *read()*, ya que no se ha conseguido que funcione, para la ocultación del puerto de conexión con el director.

Con esto queda completada la implementación del agente de engaño.

4.2. Implementación del Director de engaño

Con respecto al director, este también será desarrollado en *Python*. Concretamente, el director será una aplicación web desarrollada en el *microframework Flask*. Se ha hecho uso de *Flask* porque se considera la forma más rápida de crear una aplicación web sencilla, siguiendo el principio KISS .

Se ha usado el ejemplo del tutorial que se encuentra en la propia web de *Flask*. Se ha tomado esta aplicación web y se le han hecho pequeñas modificaciones para que incluya la funcionalidad requerida por el director de engaño.

4.2.1. Implementación del subsistema de comunicación (II)

Las modificaciones realizadas a la aplicación *Flask* del tutorial son las siguientes. Se ha importado el módulo *flaskthreads* para que la aplicación pueda lanzar un hilo que se encargue de crear la sesión *TLS* y lea del *socket* los mensajes que le mande el agente de engaño. Estos mensajes son almacenados en la base de datos *sqlite* que incluye la aplicación, pero cuyo diseño ha sido alterado. Ahora la base de datos está compuesta por una tabla de evidencias que incluye únicamente un entero como clave primaria y un segundo atributo de tipo texto que es el cuerpo de la evidencia. En el apéndice D se muestra el código que ejecuta el hilo de la aplicación *Flask*.

También se ha modificado toda la parte *HTML* de la aplicación para que únicamente se dedique a mostrar el contenido de la base de datos. Con esto se cubriría la funcionalidad del director de engaño.

El puerto que se utiliza por defecto en la aplicación *Flask* es el 1234. La dirección *IP* y el puerto deben ser introducidos como argumentos de entrada en el agente de engaño. Una vez se establece la conexión, el agente manda toda la información que recogen los procesos *inotify* y *auditd* a través del *socket* encapsulado en el protocolo *TLS*. El director recibe esta información y la almacena en su base de datos local. Una vez están en la base de datos, las evidencias se muestran por pantalla en la aplicación web.

CONCLUSIONES Y MEJORAS

5.1. Posibles mejoras de la aplicación

La plataforma de engaño desarrollada funciona. Sin embargo, algunos de los requisitos iniciales no se han conseguido implementar, como la ocultación del puerto de conexión con el director por parte del agente de engaño.

Otro aspecto a mejorar es la monitorización de las llamadas al sistema. El hecho de sólo monitorizar unas pocas implica que hay mucha información de la actividad del usuario que no se registra, y que, por lo tanto, se está perdiendo. Para ello, se podría implementar una segunda vía de comunicación, esta vez del director hacia el agente. A través de ella, el director podría modificar la configuración de monitorización del agente para que cambiara las llamadas al sistema a monitorizar o para que le envíe cierto fichero del usuario cuyo contenido se quiere comprobar.

También se podría implementar una mejora en la monitorización de las conexiones que el usuario abre. Actualmente, en la plataforma únicamente se registra la dirección *IP* asociada a la llamada al sistema *connect*. Se podría implementar un sistema que almacene una copia de la información que el usuario transmite o recibe a través las conexiones que abre.

5.2. Conclusiones finales

El principal logro de esta aplicación es que consigue ocultar procesos y directorios a una serie de comandos del sistema operativo mediante el uso de una librería compartida. Para ello, ha sido necesario estudiar y analizar el funcionamiento del sistema operativo de Linux.

El subsistema más complejo, y el que más tiempo de diseño y desarrollo ha requerido, ha sido sin duda el subsistema de ocultación. Es realmente complejo hacer algo en un ordenador o en una red sin dejar huella o ser visto.

Resulta sorprendente que, después de cursar el grado de ingeniería informática, un estudiante, ya considerado ingeniero, salga de la carrera sin tener ni idea de seguridad informática. El hecho de

no tener nociones en este campo implica que el programador desarrollará aplicaciones y sistemas vulnerables, poniendo en riesgo los datos de los usuarios que se sirven de su plataforma. El objetivo de este trabajo fue, desde el principio, servir como una pequeña introducción al mundo de la seguridad en sistemas Linux. Todo esto no ha sido más que la punta del iceberg de un mundo completamente desconocido para el estudiante.

BIBLIOGRAFÍA

- [1] D. Fraunholz, S. D. Antón, C. Lipps, D. Reti, D. Krohmer, F. Pohl, M. Tammen, and H. D. Schotten, “Demystifying deception technology: A survey,” *CoRR*, vol. abs/1804.06196, 2018. (Leer).
- [2] A. Yasinsac and Y. Manzano, “Honeytraps, a network forensic tool,” in *Sixth Multi-Conference on Systemics, Cybernetics and Informatics*, 2002. (Leer).
- [3] B. Schlossberg and W. Wang, “Distributed network security deception system,” May 30 2002. US Patent App. 09/956,942.
- [4] M. Mansoori, O. Zakaria, and A. Gani, “Improving exposure of intrusion deception system through implementation of hybrid honeypot,” *The International Arab Journal of Information Technology*, vol. 9, no. 5, pp. 436–444, 2012. (Leer).
- [5] W. Wang, J. Bickford, I. Murynets, R. Subbaraman, A. G. Forte, and G. Singaraju, “Catching the wily hacker: A multilayer deception system,” in *2012 35th IEEE Sarnoff Symposium*, pp. 1–6, IEEE, 2012. (Leer).
- [6] W. Wang, J. Bickford, I. Murynets, R. Subbaraman, A. G. Forte, G. Singaraju, et al., “Detecting targeted attacks by multilayer deception,” *Journal of Cyber Security and Mobility*, vol. 2, no. 2, pp. 175–199, 2013. (Leer).
- [7] M. Smith, “Keep it simple stupid: Using basic design maxims for effective communication,” in *ANNUAL CONFERENCE-SOCIETY FOR TECHNICAL COMMUNICATION*, vol. 45, pp. 398–401, Citeseer, 1998. (Leer).
- [8] R. Love, *Linux system programming: talking directly to the kernel and C library*. .O'Reilly Media, Inc., 2013. (Leer).
- [9] J.-W. Selij and E. van den Haak, “A visitation of sysdig,” *Project Report*, pp. 2013–2014, 2014. (Leer).
- [10] R. Brondolin, M. Ferroni, and M. Santambrogio, “Performance-aware load shedding for monitoring events in container based environments,” *ACM SIGBED Review*, vol. 16, no. 3, pp. 27–32, 2019. (Leer).
- [11] S. Miclea, “Windows and linux security audit,” *Journal of Applied Business Information Systems*, vol. 3, no. 4, p. 117, 2012. (Leer).
- [12] B. Morisson and I. Stephen Wolthusen, “Analysis of the linux audit system,” *Information Security Group, Royal Holloway University of London*, 2015. (Leer).
- [13] L. Wirzenius, J. Oja, S. Stafford, and A. Weeks, “Linux system administrator’s guide,” 2000. (Leer).
- [14] G. Borello, “Hiding linux processes for fun + profit,” *Sysdig blog*, August 2014. (Leer).
- [15] L. Degioanni, “Sysdig for ps, lsof, netstat + time travel..,” *Sysdig blog*, August 2014. (Leer).
- [16] A. S. Abed, T. C. Clancy, and D. S. Levy, “Applying bag of system calls for anomalous behavior detection of applications in linux containers,” in *2015 IEEE Globecom Workshops (GC Wkshps)*,

- pp. 1–5, IEEE, 2015. (Leer).
- [17] K. Pulo, “Fun with LD_PRELOAD,” in *linux. conf. au*, vol. 153, 2009. (Leer).
- [18] R. F. Rights, “Global information assurance certification paper,” *GIAC*, 2003. (Leer).
- [19] V. Bindschaedler, R. Shokri, and C. A. Gunter, “Plausible deniability for privacy-preserving data synthesis,” *arXiv preprint arXiv:1708.07975*, 2017. (Leer).
- [20] M. Kedziora, Y.-W. Chow, and W. Susilo, “Defeating plausible deniability of veracrypt hidden operating systems,” in *International Conference on Applications and Techniques in Information Security*, pp. 3–13, Springer, 2017. (Leer).
- [21] C. Cremers, M. Horvat, J. Hoyland, S. Scott, and T. van der Merwe, “A comprehensive symbolic analysis of tls 1.3,” in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pp. 1773–1788, 2017. (Leer).

ACRÓNIMOS

KISS Keep It Simple Stupid.

APÉNDICES

CÓDIGO DEFINITIVO DE *agente.py*

Código A.1: En esta figura se presenta el código definitivo *agente.py*. Parte I.

```
1 #!/usr/bin/python3
2
3 import daemon
4 import inotify.adapters
5 import os
6 import sys
7 import OpenSSL
8 import socket
9 import subprocess
10 import OpenSSL
11 from OpenSSL.SSL import TLSv1_2_METHOD
12 import datetime
13
14
15 HOST = '127.0.1.1'
16 PORT = 1234
17 audit_log = '.audit.log'
18
19
20 ##### COMPILACION E INSERCIÓN LIBRERIA DINAMICA #####
21 #COMPILEACION E INSERCIÓN LIBRERIA DINAMICA
22 if len(sys.argv) == 3 and sys.argv[2] == '-make':
23     os.system("make")
24     os.system("mv _libprocesshider.so /usr/local/lib/")
25     if not os.path.isfile("/etc/ld.so.preload"):
26         os.system("echo _/usr/local/lib/libprocesshider.so >> /etc/ld.so.preload")
27     os._exit(0)
28
29 ##### COMPROBAR QUE EL ACTIVO EXISTE #####
30 #COMPROBAR QUE EL ACTIVO EXISTE
31 if not os.path.isfile('/home/' + sys.argv[1] + '/credenciales/credenciales.txt'):
32     print('EL ACTIVO DE ENGAÑO NO EXISTE. Por favor créalo:\n')
33     print('Crea el fichero /home/USUARIO/credenciales/credenciales.txt')
34     os._exit(0)
```

Código A.2: En esta figura se presenta el código definitivo *agente.py*. Parte II.

```

35 def agent():
36     ##### COMPROBAR USUARIO EXISTE #####
37     #COMPROBAR USUARIO EXISTE
38     if len(sys.argv) < 4 or sys.argv[2] != '-ip':
39         print('Número_de_argumentos_incorrecto.\n'
40               'Sigue_el_siguiente_esquema:./agente_username-ip_IP,PORT[-s_kill,execve,etc][-restart]')
41         os._exit(0)
42
43     usuario = sys.argv[1]
44     if not os.path.isdir('/home/'+usuario):
45         print("El_usuario'{'}_NO_existe._Prueba_con_otro.".format(usuario))
46         os._exit(0)
47
48     ip_port = sys.argv[3].split(',')
49     HOST = ip_port[0]
50     PORT = int(ip_port[1])
51
52
53     ##### BUCLE INFINITO PARA LA ACTIVACION DEL ACTIVO #####
54     #BUCLE INFINITO PARA LA ACTIVACION DEL ACTIVO
55     i = inotify.adapters.Inotify()
56     i.add_watch('/home/'+usuario+'/credenciales')
57
58     for event in i.event_gen(yield_nones=False):
59         (_, type_names, path, filename) = event
60
61         if filename == 'credenciales.txt' and path == '/home/'+usuario+'/credenciales':
62             break
63
64
65     ##### CREACION FICHERO PARA COMPROBAR USUARIO ROOT CON is_sudo de processhider.c #####
66     #CREACION FICHERO PARA COMPROBAR USUARIO ROOT CON is_sudo de processhider.c
67     if not os.path.isfile('/etc/hola'):
68         os.system("touch_/etc/hola")
69         os.system("chmod_600_/etc/hola")

```

Código A.3: En esta figura se presenta el código definitivo *agente.py*. Parte III.

```
#####
# Funcion para reiniciar y reconfigurar auditd
def restart_audited():
    os.system('systemctl_stop_audited')
    if os.path.isfile('/home/+'+usuario+'/'+audit_log):
        os.system('rm-/home/+'+usuario+'/'+audit_log)
    with open('/etc/audit/audited.conf', "w") as audit_conf:
        audit_conf.write ('#/n#_This_file_controls_the_configuration_of_the_audit_daemon\n#\n'
                          'local_events_=yes\n'
                          'write_logs_=yes\n'
                          'log_file_=~/home/+'+usuario+'/'+audit_log+'\n'
                          'log_group_=adm\n'
                          'log_format_=ENRICHED\n'
                          'flush_=INCREMENTAL_ASYNC\n'
                          'freq_=50\n'
                          'max_log_file_=8\n'
                          'num_logs_=5\n'
                          'priority_boost_=4\n'
                          'disp_qos_=lossy\n'
                          'dispatcher_=sbin/audispd\n'
                          'name_format_=NONE\n'
                          '##name_=mydomain\n'
                          'max_log_file_action_=ROTATE\n'
                          'space_left_=75\n'
                          'space_left_action_=SYSLOG\n'
                          'verify_email_=yes\n'
                          'action_mail_acct_=root\n'
                          'admin_space_left_=50\n'
                          'admin_space_left_action_=SUSPEND\n'
                          'disk_full_action_=SUSPEND\n'
                          'disk_error_action_=SUSPEND\n'
                          'use_libwrap_=yes\n'
                          '##tcp_listen_port_=60\n'
                          'tcp_listen_queue_=5\n'
                          'tcp_max_per_addr_=1\n'
                          '##tcp_client_ports_=1024-65535\n'
                          'tcp_client_max_idle_=0\n'
                          'enable_krb5_=no\n'
                          'krb5_principal_=audited\n'
                          '##krb5_key_file_=etc/audit/audit.key\n'
                          'distribute_network_=no\n')

    os.system('touch~/home/+'+usuario+'/'+audit_log)
    os.system('chmod_600~/home/+'+usuario+'/'+audit_log)
```

Código A.4: En esta figura se presenta el código definitivo *agente.py*. Parte IV.

```

114     with open('/etc/audit/rules.d/audit.rules', "w") as audit_rules:
115         audit_rules.write('##_First_rule_-_delete_all\n'
116                         '-D\n\n'
117                         '##_Increase_the_buffers_to_survive_stress_events.\n'
118                         '##_Make_this_bigger_for_busy_systems\n'
119                         '-b_8192\n\n'
120                         '##_This_determine_how_long_to_wait_in_burst_of_events\n'
121                         '--backlog_wait_time_0\n\n'
122                         '##_Set_failure_mode_to_syslog\n'
123                         '-f_1\n\n')
124         syscalls = sys.argv[5].split(',')
125         for s in syscalls:
126             audit_rules.write('-a(always,exit,-F,arch=b64,-S,+s+-k,+s+)_rule,-F,-'
127                               'uid='+usuario+'\n')
128         os.system('systemctl_restart_audited')
129
130
131 ######
132 # Aniadir syscalls a audited
133 def add_syscall():
134     os.system('systemctl_stop_audited')
135     with open('/etc/audit/rules.d/audit.rules', "a") as audit_rules:
136         syscalls = sys.argv[5].split(',')
137         for s in syscalls:
138             audit_rules.write('-a(always,exit,-F,arch=b64,-S,+s+-k,+s+)_rule,-F,-'
139                               'uid='+usuario+'\n')
140         os.system('systemctl_restart_audited')
141
142
143 ######
144 # Gestión de audited segun argumentos de entrada
145 if (len(sys.argv) == 7 and sys.argv[6] == '--restart' and sys.argv[4] == '-s') or not
146     os.path.isfile('/home/'+usuario+'/'+audit_log):
147     restart_audited()
148 elif len(sys.argv) == 6 and sys.argv[4] == '-s':
149     add_syscall()
150
151
152 ######
153 #ESTRUCTURACION DIRECTORIO DE EVIDENCIAS
154 hiddenD = "/etc/dpkg/origins..."
155 if not os.path.isdir(hiddenD):
156     os.system("mkdir "+hiddenD)
157
158 os.system("chmod_600_"+hiddenD)
159
160 hiddenD = hiddenD+'/'+usuario
161 os.system("cp_r_/home/{}_{}.format(usuario, hiddenD))

```

Código A.5: En esta figura se presenta el código definitivo *agente.py*. Parte V.

```
162 if not os.path.isfile(hiddenD+='/auditd.txt'):
163     os.system('touch '+hiddenD+='/auditd.txt')
164
165
166
167 ######
168 # AGENTE DE ENGAÑO: monitoriza todos los eventos en el sistema de ficheros
169 # Create context for de TLS session
170 context = OpenSSL.SSL.Context(TLSv1_2_METHOD)
171
172 # Create connection
173 conn = OpenSSL.SSL.Connection(context, socket.socket(socket.AF_INET, socket.SOCK_STREAM))
174 conn.set_connect_state()
175
176 # Connect to server
177 conn.connect((HOST, PORT))
178
179 i = inotify.adapters.InotifyTree('/home/'+usuario)
180 # Variables utiles
181 create = 0
182 modify = 0
183 move = 0
184 move_to = 0
185 delete = 0
186 fichero = ""
187 aux_destiny = ""
188 aux_source = ""
189 IN_ISDIR = 0
190 flag_aux = 0
191 old = ""
192
193 options = {0:"FICHERO", 1:"DIRECTORIO"}
194
195 content_aux = 'Key_
Report\n=====
date_time_key_success_exe_auid_
event\n=====\\n'
196
197 with open(hiddenD+='/auditd.txt', "w") as writer:
198     writer.write(content_aux)
199
200 for event in i.event_gen(yield_nones=False):
201     (extra_info, type_names, path, filename) = event
```

Código A.6: En esta figura se presenta el código definitivo *agente.py*. Parte VI.

```

202 #no_metadatos = ".local" not in path and ".cache" not in path and ".config" not in path and
203 #".swp" not in filename
204 no_metadatos = "." not in path and ".swp" not in filename
205
206 if no_metadatos:
207
208     source = path+"/"+filename
209     destiny = hiddenD+path.split(usr)[1]+"/"+filename
210
211     # Si se ejecuta el director y el agente en la misma maquina
212     if "instance" in path:
213         pass
214
215     # AUDITD LOG HANDLING
216     elif filename == audit_log and 'IN_MODIFY' in type_names:
217
218         cmd_aureport_k = "aureport_-k"
219         p = subprocess.Popen(cmd_aureport_k, shell=True, stdout=subprocess.PIPE)
220         content = p.stdout.read().decode()
221
222         lista_aux = content.split('\n')
223
224         with open(hiddenD+'/auditd.txt', "r") as reader:
225             old = reader.read()
226
227         with open(hiddenD+'/auditd.txt', "a") as writer:
228             for e in lista_aux:
229                 if e not in old:
230                     eventid = e.split('_')[-1]
231                     cmd_aureport_u = "aureport_-u_|_grep_|_" + eventid
232                     p2 = subprocess.Popen(cmd_aureport_u, shell=True,
233                                         stdout=subprocess.PIPE)
234                     content2 = p2.stdout.read().decode()
235
236                     if content2 != '':
237                         e = e + '_' + content2.split('_')[5]
238
239                     conn.write(e)
240                     writer.write(e)
241
242     # REST OF FILESYSTEM HANDLING
243     elif 'IN_CREATE' in type_names:
244         create = 1

```

Código A.7: En esta figura se presenta el código definitivo *agente.py*. Parte VII.

```
244 elif 'IN_CLOSE_WRITE' in type_names:
245     os.system("cp_r_{ }_{ }".format(source, destiny))
246     if create == 1:
247         conn.write(datetime.datetime.now().strftime("%Y-%m-%d_%H:%M:%S")+"_"
248                     FICHERO_{ }\tCREADO_en_la_ruta_{ }\n".format(filename, source))
249     create = 0
250     elif move_to == 1 or modify == 1:
251         conn.write(datetime.datetime.now().strftime("%Y-%m-%d_%H:%M:%S")+"_"
252                     FICHERO_{ }\tMODIFICADO_en_la_ruta_{ }\n".format(filename, source))
253     move_to = modify = 0
254
255 elif 'IN_CLOSE_NOWRITE' in type_names:
256     if create==1 and 'IN_ISDIR' in type_names:
257         os.system("cp_r_{ }_{ }".format(source, destiny))
258         conn.write(datetime.datetime.now().strftime("%Y-%m-%d_%H:%M:%S")+"_"
259                     DIRECTORIO_{ }\tCREADO_en_la_ruta_{ }\n".format(filename, source))
260     create = 0
261
262 elif 'IN_DELETE' in type_names:
263     IN_ISDIR = 'IN_ISDIR' in type_names
264     aux_destiny = destiny
265     destiny = hiddenD+path.split(usuario)[1]+"/DELETED_"+filename
266     os.system("mv_{ }_{ }".format(aux_destiny, destiny))
267     conn.write(datetime.datetime.now().strftime("%Y-%m-%d_%H:%M:%S")+"_"
268                     '{ }\tELIMINADO_en_la_ruta_{ }\n".format(options[IN_ISDIR],filename, source))
269
270 elif 'IN_MOVED_FROM' in type_names:
271     move = 1
272     fichero = filename
273     aux_source = source
274     aux_destiny = destiny
275
276 elif 'IN_MOVED_TO' in type_names:
277     if move == 1:
278         IN_ISDIR = 'IN_ISDIR' in type_names
279         os.system("mv_{ }_{ }".format(aux_destiny, destiny))
280         conn.write(datetime.datetime.now().strftime("%Y-%m-%d_%H:%M:%S")+"_"
281                     '{ }\tMOVIDO_de_la_ruta_{ }_a_la_ruta_{ }\n".format(options[IN_ISDIR],
282                     fichero, aux_source, source))
283     move = 0
284     else:
285         move_to = 1
286
287 elif 'IN MODIFY' in type_names:
288     modify = 1
289
290 #####
291 # DEMONIZACION
292 with daemon.DaemonContext(stdin=sys.stdin, stdout=sys.stdout):
293     agent()
```


CÓDIGO DEFINITIVO DE LA LIBRERÍA COMPARTIDA.

Código B.1: En este figura se muestra el código de la librería compartida. Parte I.

```
1 #define _GNU_SOURCE
2
3 #include <stdio.h>
4 #include <dlfcn.h>
5 #include <dirent.h>
6 #include <string.h>
7 #include <unistd.h>
8
9 /*
10 *Every process with this name will be excluded
11 */
12 static const char* process_to_filter = "agente.py";
13 static const char* process_to_filter2 = "auditd";
14
15 /*
16 *File to filter
17 */
18 static const char* file_to_filter = ".audit.log";
19
20 /*
21 *Port to hide
22 */
23 static const char* port_to_filter = ":04D2";
24 */
25
26 /*
27 */Proc/self
28 */
29
30 /*static const char* proc_self = "/proc/self";*/
```

Código B.2: En este figura se muestra el código de la librería compartida. Parte II.

```

31  /*
32   *Get a directory name given a DIR* handle
33   */
34 static int get_dir_name(DIR* dirp, char* buf, size_t size)
35 {
36     int fd = dirfd(dirp);
37     if(fd == -1) {
38         return 0;
39     }
40
41     char tmp[64];
42     snprintf(tmp, sizeof(tmp), "/proc/self/fd/%d", fd);
43     ssize_t ret = readlink(tmp, buf, size);
44     if(ret == -1) {
45         return 0;
46     }
47
48     buf[ret] = 0;
49     return 1;
50 }
51
52 /*
53 *Get a process name given its pid
54 */
55 static int get_process_name(char* pid, char* buf)
56 {
57     if(strspn(pid, "0123456789") != strlen(pid)) {
58         return 0;
59     }
60
61     char tmp[256];
62     snprintf(tmp, sizeof(tmp), "/proc/%s/stat", pid);
63
64     FILE* f = fopen(tmp, "r");
65     if(f == NULL) {
66         return 0;
67     }
68
69     if(fgets(tmp, sizeof(tmp), f) == NULL) {
70         fclose(f);
71         return 0;
72     }
73
74     fclose(f);
75
76     int unused;
77     sscanf(tmp, "%d (%[^)]s", &unused, buf);
78     return 1;
79 }
```

Código B.3: En este figura se muestra el código de la librería compartida. Parte III.

```
80  /*
81   *Detect if uid is 0
82   */
83 static int is_sudo(){
84
85     FILE* f = fopen("/etc/hola", "r");
86     if(f == NULL) {
87         return 0;
88     }
89
90     fclose(f);
91     return 1;
92 }
93
94 #define DECLARE_REaddir(dirent, readdir) \
95 static struct dirent* (*original_##readdir)(DIR*) = NULL; \
96 \
97 struct dirent* readdir(DIR *dirp) \
98 { \
99     if(original_##readdir == NULL) { \
100         original_##readdir = dlsym(RTLD_NEXT, #readdir); \
101         if(original_##readdir == NULL) \
102             { \
103                 fprintf(stderr, "Error in dlsym: %s\n", dlerror()); \
104             } \
105     } \
106 \
107     struct dirent* dir; \
108 \
109     while(1) \
110     { \
111         dir = original_##readdir(dirp); \
112         if(dir) { \
113             char dir_name[256]; \
114             char process_name[256]; \
115             if( is_sudo() == 0 && \
116                 ((get_dir_name(dirp, dir_name, sizeof(dir_name)) && \
117                  strcmp(dir_name, "/proc") == 0 && \
118                  get_process_name(dir->d_name, process_name) && \
119                  strcmp(process_name, process_to_filter) == 0) || \
120                  (strcmp(dir_name, "/etc/dpkg/origins") == 0 && \
121                  strcmp(dir->d_name, "...") == 0) || \
122                  (strstr(process_name, process_to_filter2) != NULL) || \
123                  (strcmp(dir->d_name, file_to_filter) == 0))) { \
124                     continue; \
125                 } \
126             } \
127             break; \
128     } \
129     return dir; \
130 }
```

Código B.4: En este figura se muestra el código de la librería compartida. Parte IV.

```
131 DECLARE_READDIR(dirent64, readdir64);
132 DECLARE_READDIR(dirent, readdir);
133
134
135 /*
136 static int get_process_pid(char* buf, size_t size)
137 {
138
139     ssize_t ret = readlink(proc_self, buf, size);
140     if (ret == -1)
141         return 0;
142
143     buf[ret] = 0;
144     return 1;
145 }
146
147
148
149 static int get_file_name(int fd, char* buf, size_t size)
150 {
151     if(fd == -1) {
152         return 0;
153     }
154
155     char tmp[64];
156     snprintf(tmp, sizeof(tmp), "/proc/self/fd/%d", fd);
157     ssize_t ret = readlink(tmp, buf, size);
158     if(ret == -1) {
159         return 0;
160     }
161
162     buf[ret] = 0;
163     return 1;
164 }
165 */
```

Código B.5: En este figura se muestra el código de la librería compartida. Parte V.

```
166  /*
167  static ssize_t (*original_read)(int fd, void *buf, size_t count) = NULL;
168
169  ssize_t read (int fd, void *buf, size_t count){
170
171      if(original_read == NULL) {
172          original_read = dlsym(RTLD_NEXT, "read");
173          if(original_read == NULL)
174          {
175              fprintf(stderr, "Error in dlsym: %s\n", dlerror());
176          }
177      }
178
179      char pid_string[256];
180      char filename[256];
181      char tmp[256];
182      char line[256];
183
184      get_process_pid(pid_string, sizeof(pid_string));
185      get_file_name(fd, filename, sizeof(filename));
186
187      snprintf(tmp, sizeof(tmp), "/proc/%s/net/tcp", pid_string);
188
189      get_process_name(pid_string, line);
190
191      if(strcmp(tmp, filename) == 0){
192          printf("Process name = %s\n", line);
193      }
194
195
196
197
198      return original_read(fd, buf, count);
199  }
200  */
```


CÓDIGO PARA LA CREACIÓN DEL CERTIFICADO AUTO-FIRMADO.

Código C.1: En esta figura se presenta el código para crear el certificado auto-firmado. Parte I.

```
1 from cryptography import x509
2 from cryptography.hazmat.backends import default_backend
3 from cryptography.hazmat.primitives import serialization
4 from cryptography.hazmat.primitives.asymmetric import rsa
5 from cryptography.x509.oid import NameOID
6 from cryptography.hazmat.primitives import hashes
7 import datetime
8
9 ROOT_PASS = b"holo"
10 SERVER_PASS = b"director"
11
12 def root_certificate():
13     # Creating new private key
14     key = rsa.generate_private_key(
15         public_exponent=65537,
16         key_size=2048,
17         backend=default_backend()
18     )
19
20     # Writing private key to disk for safe keeping
21     with open("key.pem", "wb") as f:
22         f.write(key.private_bytes(
23             encoding=serialization.Encoding.PEM,
24             format=serialization.PrivateFormat.TraditionalOpenSSL,
25             encryption_algorithm=serialization.BestAvailableEncryption(ROOT_PASS),
26         ))
27
28     # Details for the certificate. Subject and issuer are always the same.
29     subject = issuer = x509.Name([
30         x509.NameAttribute(NameOID.COUNTRY_NAME, u"ES"),
31         x509.NameAttribute(NameOID.STATE_OR_PROVINCE_NAME, u"Madrid"),
32         x509.NameAttribute(NameOID.LOCALITY_NAME, U"Madrid"),
33         x509.NameAttribute(NameOID.ORGANIZATION_NAME, u"Simple_Deception"),
34         x509.NameAttribute(NameOID.COMMON_NAME, u"simple.deception"),
35     ])
```

Código C.2: En esta figura se presenta el código para crear el certificado auto-firmado. Parte II.

```

36     cert = x509.CertificateBuilder().subject_name(
37         subject
38     ).issuer_name(
39         issuer
40     ).public_key(
41         key.public_key()
42     ).serial_number(
43         x509.random_serial_number()
44     ).not_valid_before(
45         datetime.datetime.utcnow()
46     ).not_valid_after(
47         # Our certificate will be valid for 1 year
48         datetime.datetime.utcnow() + datetime.timedelta(days=365)
49     ).add_extension(
50         x509.SubjectAlternativeName([x509.DNSName(u"localhost")]),
51         critical=False,
52     # Sign our certificate with our private key
53     ).sign(key, hashes.SHA256(), default_backend())
54
55     with open("certificate.pem", "wb") as f:
56         f.write(cert.public_bytes(serialization.Encoding.PEM))
57
58 def server_certificate():
59
60     ca_cert = x509.load_pem_x509_certificate(
61         open("certificate.pem", "rb").read(),
62         default_backend()
63     )
64
65     ca_rootkey = serialization.load_pem_private_key(
66         open("key.pem", "rb").read(),
67         password=ROOT_PASS,
68         backend=default_backend()
69     )
70
71     # Creating new private key
72     key = rsa.generate_private_key(
73         public_exponent=65537,
74         key_size=2048,
75         backend=default_backend()
76     )
77
78     # Writing private key to disk for safe keeping
79     with open("server_key.pem", "wb") as f:
80         f.write(key.private_bytes(
81             encoding=serialization.Encoding.PEM,
82             format=serialization.PrivateFormat.TraditionalOpenSSL,
83             encryption_algorithm=serialization.NoEncryption(),
84         ))

```

Código C.3: En esta figura se presenta el código para crear el certificado auto-firmado. Parte III.

```
85 # Details for the certificate.
86 subject = x509.Name([
87     x509.NameAttribute(NameOID.COUNTRY_NAME, u"ES"),
88     x509.NameAttribute(NameOID.STATE_OR_PROVINCE_NAME, u"Madrid"),
89     x509.NameAttribute(NameOID.LOCALITY_NAME, U"Madrid"),
90     x509.NameAttribute(NameOID.ORGANIZATION_NAME, u"JSPC"),
91     x509.NameAttribute(NameOID.COMMON_NAME, u"javier.sanchez"),
92 ])
93
94
95 server_cert = x509.CertificateBuilder().subject_name(
96     subject
97     ).issuer_name(
98         ca_cert.issuer
99     ).public_key(
100        key.public_key()
101     ).serial_number(
102        x509.random_serial_number()
103     ).not_valid_before(
104        datetime.datetime.utcnow()
105     ).not_valid_after(
106        # Our certificate will be valid for 1 year
107        datetime.datetime.utcnow() + datetime.timedelta(days=365)
108     ).add_extension(
109        x509.SubjectAlternativeName([x509.DNSName(u"localhost")]),
110        critical=False,
111    # Sign our certificate with our ca private key
112    ).sign(ca_rootkey, hashes.SHA256(), default_backend())
113
114 with open("server_cert.pem", "wb") as f:
115     f.write(server_cert.public_bytes(serialization.Encoding.PEM))
116
117 print("Making_root_certificate...")
118 root_certificate()
119 print("Done")
120 print("Making_server_certificate...")
121 server_certificate()
122 print("Done")
```


CÓDIGO DEL HILO DE LA APLICACIÓN

Flask.

Código D.1: En esta figura se presenta el código del hilo de la aplicación Flask. Parte I.

```
1 # socket-reading thread
2 def thread_job():
3
4     import OpenSSL
5     from OpenSSL.SSL import TLSv1_2_METHOD, FILETYPE_PEM,
6         VERIFY_FAIL_IF_NO_PEER_CERT
7     import socket
8     from cryptography import x509
9     from cryptography.hazmat.backends import default_backend
10    from cryptography.hazmat.primitives import serialization
11
12    def hola():
13        pass
14
15    # Create context for de TLS session
16    context = OpenSSL.SSL.Context(TLSv1_2_METHOD)
17
18    # Load server private key and cert
19    context.use_privatekey_file(os.path.join(app.instance_path, "server_key.pem"))
20    context.use_certificate_file(os.path.join(app.instance_path, "server_cert.pem"))
21
22    # Add verify mode
23    context.set_verify(VERIFY_FAIL_IF_NO_PEER_CERT, hola)
24
25    # Load root certificate
26    context.load_verify_locations(cafile=os.path.join(app.instance_path, "certificate.pem"))
27
28    # Create the initial connection with the above context and a socket
29    soc = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
30    soc.setblocking(1)
31    soc.bind((HOST, PORT))
32    soc.listen(1)
33    conn_ini = OpenSSL.SSL.Connection(context, soc)
```

Código D.2: En esta figura se presenta el código del hilo de la aplicación Flask. Parte II.

```

33 # Accept client connection
34 while 1:
35     conn, addr = conn_ini.accept()
36     conn.set_accept_state()
37     print("Connected by " + str(addr))
38
39     while 1:
40         try:
41             data = conn.read(1024)
42
43             # Connect to the flask database
44             conn_db = connect(os.path.join(app.instance_path, "flaskr.sqlite"))
45             curs = conn_db.cursor()
46             evidencias = data.decode().split('\n')
47             for e in evidencias:
48                 if e != " and "_rule" in e:
49                     e = e.split('.)[1]
50                     e = "AUDITD:" + e
51                 elif e != "":
52                     e = "INOTIFY:" + e
53
54                 if e != "":
55                     curs.execute("INSERT INTO evidence_(body)_VALUES_(?);",
56                         [e],
57                         )
58                     conn_db.commit()
59                     conn_db.close()
60                     #print(data.decode())
61             except OpenSSL.SSL.SysCallError as e:
62                 #if e[0] == -1 or e[1] == 'Unexpected EOF':
63                 conn.shutdown()
64                 break
65
66             with app.app_context():
67                 t = ApplicationContextThread(target=thread_job)
68                 t.start()

```

MANUAL DE USUARIO Y CONTENIDO DE LA ENTREGA

Junto con este documento se incluyen tres directorios que forman el total de la aplicación: *certs*, *agente* y *director*.

certs

En la carpeta *certs* se encuentra el generador de certificados auto-firmados. Para crear un nuevo certificado raíz y un nuevo certificado del servidor, simplemente hay que ejecutar el programa *self_cert.py*. Para el certificado raíz, se crea una clave RSA. La clave privada se cifra y se escribe en el fichero *key.pem* utilizando como clave de cifrado el valor de la variable *ROOT_PASS*. El certificado raíz (*certificate.pem*) se crea utilizando la clave pública de *key.pem*. Para el certificado del servidor, se crea otra clave RSA. Esta vez la clave privada no se cifra (por comodidad) y se escribe directamente en el fichero *server_key.pem*. El certificado del servidor (*server_cert.pem*) se crea a partir de *certificate.pem* y de la clave pública de *server_key.pem*. Se puede ver el código de *self_cert.py* en el apéndice C.

director

En la carpeta *director* se encuentra la aplicación *Flask* del director de engaño. Primero, es necesario introducir en la directorio *instance* los siguientes ficheros: *certificate.pem*, *server_key.pem* y *server_cert.py*. El siguiente paso es crear un entorno virtual e instalar el paquete *flaskr*, tal y como se muestra en la figura E.1.

```
$ python3 -m venv venv  
$ . venv/bin/activate  
  
$ pip install -e .
```

Figura E.1: En esta figura se muestra cómo crear y activar el entorno virtual y cómo instalar el paquete *flaskr*.

Una vez hecho esto, es necesario exportar la aplicación *flaskr* e iniciar la base de datos, tal y como se muestra en la figura E.2. Es necesario tener instalados los paquetes de *Python* que se muestran en el fichero *reqs.py* antes de ejecutar la aplicación.

```
$ export FLASK_APP=flaskr  
$ export FLASK_ENV=development  
$ flask init-db  
$ flask run
```

Figura E.2: En esta figura se muestra cómo exportar *flaskr*, cómo iniciar la base de datos y cómo ejecutar la aplicación.

Una vez ejecutada la aplicación, el director permanecerá a la espera hasta que el agente se conecte y le empiece a mandar mensajes.

agente

En la carpeta *agente* se encuentran tres ficheros: *agente.py*, *processhider.c* y su fichero *Makefile* correspondiente. Como ya se ha indicado, *agente.py* es el script del agente del sistema de engaño. Para poder ejecutarlo es necesario crear previamente el activo de engaño, que en este caso es un fichero de texto ubicado en */home/USUARIO/credenciales/credentials.txt*, donde *USUARIO* es el nombre del usuario a monitorizar. Además, es necesario también compilar primero la librería compartida *processhider.c* mediante «*./agente.py -make*». Una vez se ha creado el activo y se ha compilado la librería compartida, se puede ejecutar el agente siguiendo el siguiente esquema (es obligatorio que el director esté escuchando por el puerto y que el agente lo ejecute el usuario administrador):

```
./agente.py USUARIO -ip IPPORT -s SYSCALL1,SYSCALL2,etc [-restart]
```

La opción *[-restart]* es opcional y es para reiniciar *auditd* limpiando su fichero de registro; se recomienda usarlo la primera vez. El argumento *USUARIO* es el usuario a monitorizar. El argumento *IP,PORT* (separados por una coma) son la dirección *IP* y el puerto del director. El número de syscalls a monitorizar es a elección (separadas por comas), aunque se recomienda no pasar de tres para no afectar en exceso al rendimiento del sistema. El agente comenzará a enviar datos al director en el momento en que se capture un evento sobre el activo de engaño.

Nota importante: se recomienda que la aplicación se ejecute en entornos de prueba o máquinas virtuales.

