KERNEL MONOLITICI

Vs.

MODULI DI KERNEL

sono una delle features piu` potenti dei sistemi odierni, basati su linux

- **permettono di caricare drivers, od un qualunque "*programma*" particolare a livello kernel quando "*serve*"**

cosa succede se non si puo` piu` credere neanche al proprio kernel ???

L'uso di server linux cresce di giorno in giorno (in realta' di ora in ora)...

'hackerare' un sistema linux diventa sempre piu` interessante.

Una delle migliori tecniche e` attaccare un sistema linux utilizzando **kernel code.**

Gli **LKM** diventano un punto di ingresso privilegiato per i nostri hack al sistema.

E` possibile scrivere codice che gira a livello kernel, che ci permettera` di accedere a parti molto delicate e sensibili del cuore del nostro sistema.

andare contro il lato tradizionale del kernel

grossa teoria dietro la programmazione

vs.

gettare le basi per una buona conoscenza di base nel sistema degli LKM per muoversi adeguatamente

Cosa e` un LKM - Per Iniziare:

- sono moduli di kernel, usati per espanderne le sue funzionalita`.
- possono essere caricati/scaricati dinamicamente
- non richiedono una ricompilazione massiccia del kernel (a meno di particolari casi)
- vengono utilizzati tipicamente per scopi particolari (drivers, hacking, hidesystem, etc)

Ogni modulo consiste di un minimo di due funzioni:

int init_module(void) */* usata per tutti i paramentri e inizializzazioni */ { ... }*

void cleanup_module(void) /* usata per scaricare in maniera pulita il modulo */ { ... }

Il caricamento di un modulo, e` tipicamente ristretto a **root**

Due comandi, ke lavorano in sinergia tra loro:

- **insmod module.o**
- **modprobe module**

quello ke succede e` qualkosa del tipo:

- caricare l`objectfile (nel nostro caso module.o)
- chiamare la systemcall create_module per l`allocazione della memoria
- Eventuali reference non risolte, sono risolte dal kernel attraverso la chiamata di sistema:

get_kernel_syms

- dopo questo la systemcall init_module viene utilizzata per l`inizializzazione

```
#define MODULE
#include <linux/module.h>

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

void cleanup_module(void) {
	printk("<1>Goodbye cruel world\n");
}
```

Prima DIFFERENZA.....aaaaaahaaaaaaah ->

- non ho utilizzato printf

la programmazione a livello kernel, e` totalmente diversa dalla programmazione a livello utente!

- a livello kernel, abbiamo soltanto un ristretto set di comandi

non si puo` proprio fare tutto
(a meno di non essere dei virtuosi dell`asm)

come aiutare i non virtuosi, ad utilizzare funzioni a livello utente nel livello kernel ??

Giusto per essere chiari, dobbiamo dare al gcc alcuni parametri per far compilare adeguatamente la nostra creatura, questo perke` e` un modulo, non dimentichiamolo (=

```
# gcc -c- O3 -Wall -fomit-frame-pointer helloworld.c
           ^^  ^^^          ^^^^
            \___\________\_____
                    vedremo dopo a ke servono, eh!
```

 # insmod helloworld.o

A questo punto il nostro modulo e` in memoria, e mostra il nostro testo.

per verificare che il nostro modulo sta` davvero in kernel space, e gira....

 # lsmod

lsmod, legge alcune informazioni da /proc/modules, per mostrarci quali moduli sono caricati in quel momento.

+ 'Pages' sono informazioni riguardanti la memoria (quante pagine questo modulo riempie).

+ 'Used by' ci dice quanto spesso il modulo e` usato dal sistema (contatore di referenze).

Il modulo puo` essere rimosso, solo quando questo contatore e` a zero.

 # rmmod helloworld

Benissimo, abbiamo appena **ABUSATO** (???) del nostro kernel....

la famiglia si windows nt/2000/xp, possiede un sistema piu` o meno analogo a questo
per la gestione di particolari porzioni di codice, essi vengono presentato sotto il nome di files **.VXD**.

La parte davvero interessante di questi pezzi di programma, e` quella di
rimanere praticamente residenti nel sistema

poter **hookare/wrappare** le chiamate del sistema *(in linux: systemcalls)....*

SystemCalls queste sconosciute

Sono il livello piu` basso di funzioni disponibili in un sistema unix

sono implementate all`interno del kernel

linux le chiama: syscall

rappresentano un **transiente** tra lo spazio utente e quello kernel

L'apertura di un file a livello utente, e' rappresentato dall'uso di una syscall: **__NR_OPEN** a livello kernel.

Per una lista completa delle funzioni disponibili nel proprio sistema basta dare un occhio al file:

/usr/include/asm/unistd.h

```
#ifndef _ASM_I386_UNISTD_H_
#define _ASM_I386_UNISTD_H_

/*
* This file contains the system call numbers.
*/

#define __NR_exit                1
#define __NR_fork                2
#define __NR_read                3
#define __NR_write               4
#define __NR_open                5
#define __NR_close               6
#define __NR_waitpid             7
#define __NR_creat               8
#define __NR_link                9
#define __NR_unlink              10
#define __NR_execve              11
#define __NR_chdir               12
#define __NR_time                13

[SNIP]
```

- **Ogni systemcall ha un numero di definizione, che viene usato per effettuare la systemcall.**

Il kernel utilizza l'interrupt **0x80** per gestirle

numero della systemcall
ed ogni argomento vengono mossi su un registro
(eax per il numero della systemcall, ad esempio).

Il numero di systemcall e' un indice in un array di una struttura kernel chiamata **sys_call_table[].**

Questa struttura mappa i numeri di systemcall per le funzioni che ne richiedono un servizio.

Ovviamente questo non e' tutto

Ma…

questo e' un workshop, non un corso di studi

chiedete ai vostri professori universitari
(se non ci siete, aspettate di arrivarci)
per avere i dovuti approfondimenti

proveremo a vedere come funzionano e come possono essere sfruttate a nostro vantaggio
in termini pratici

La KERNEL-Symbol-Table

c'e' un'altro importante punto da sottolineare:

La tabella dei simboli.

Proviamo a dare un okkio alla struttura **/proc/ksyms.**

c01f05dc nf_unregister_hook_R8358a47a
c01f0608 nf_register_sockopt_R64785b9d

Ogni entry in questo file rappresenta un simbolo di kernel pubblico **ESPORTATO**, che puo' essere utilizzato dal nostro modulo.

c'e' un problema…

Ogni simbolo usato nel nostro modulo (come funzione) e' altresi` esportato al pubblico, ed e' quindi listato in questo file!!

un admin esperto puo' scoprire cose "particolari" e killarle (=.

ci sono tanti metodi per nascondersi ed occultarsi dalla vista degli amministratori...

un modo comune, che spesso viene utilizzato dai programmatori di moduli kernel, per evitare che i propri simboli siano esportati in /proc/ksysm.

Qualkosa del genere, e' adatta al nostro scopo:

static struct symbol_table module_syms={ … }

/* definiamo la nostra personale tabella dei simboli */

register_symtab(&module_syms);

non vogliamo esportare nessun simbolo al pubblico

- ***register_symtab(NULL);***

- ***EXPORT_NO_SYMBOLS;***

nei kernel moderni si usa una delle quattro macro messe a disposizione:

EXPORT_SYMTAB;
da specificare prima di linux/module.h, per esportare simboli

EXPORT_SYMBOL(name);
esportiamo il simbolo chiamato 'name'

EXPORT_SYMBOL_NOVERS (name);
esportiamo senza informazioni sulla versione il simbolo 'name'

EXPORT_NO_SYMBOLS;
credo sia autoesplicativo

Kernel in Spazio Memoria-Utente

svantaggi ?

in effetti, ci stanno anche quelli...

- Le systemcalls, prendono i loro argomenti dallo user-space ed il nostro modulo lavora in kernelspace.

- *come possiamo accedere ad un argomento allocato in userspace dal nostro kernelspace ?*

Dobbiamo fare una **transizione**

prendiamo in nota, la seguente systemcall:

int sys_chdir (const char *path)

Immaginiamo che il sistema ne faccia in qualke modo uso (come, dove e quando, non e' importante)

Quello ke vorremmo fare e` controllare il percorso che l'utente vuole utilizzare;

quello ke dobbiamo fare, quindi, e` accedere a:

const char *path

se provassimo a fare qualkosa del tipo:

printk("<1>%s\n", path);

il minimo che dovremmo aspettarci e` un coredump galattico...

in realta' non succede…questo tipo di bugs sono MOLTO piu' insidiosi…

siamo in **kernelspace**, e non e` possibile mappare, leggere e scrivere la memoria a
livello utente, come faremmo dall'altro lato.

Una prima soluzione, a questo scempio e` la seguente:

get_user(pointer);

dando a questa funzione un puntatore alla nostra locazione di *path, otterremo i bytes voluti dal nostro userspace in kernelspace.

Diamo un okkiata ad una implementazione tipica di quanto detto:

```
char *strncpy_fromfs(char *dest, const char *src, int n)
{
	char *tmp = src;
	int compt = 0;
	do {
		dest[compt++] = __get_user(tmp++, 1);
	} while ((dest[compt - 1] != '\0') && (compt != n));
	return dest;
}
```

Se si vuole convertire la nostra variabile *path, possiamo usare qualkosa del tipo:

```
char *kernel_space_path;

/* allochiamo memoria in kernelspace*/
kernel_space_path = (char *) kmalloc(100, GFP_KERNEL);

/* chiamiamo la funzione di cui sopra */
 (void) strncpy_fromfs(test, path, 20);

/* ora possiamo farci quello ke vogliamo */
printk("<1>%s\n", kernel_space_path);

kfree(test);
```

In realta` quello appena descritto e` un percorso da seguire solo in caso di **reale bisogno.**

Quello ke si fa` piu` spesso, a meno di costrizioni particolari (restrizioni del sistema, hacking pesanti, etc)
e` quello di usare la funzione:

void memcpy_fromfs(void *to, const void *from, ulong count);
void memcpy_tofs(void *to, const void *from, ulong count);

entrambe le funzioni sono ovviamente basata sullo stesso tipo di comandi.

La prima serve per copiare
una porzione di memoria dall'userspace in kernelspace,

la seconda serve per fare esattamente il contrario.

Questo e` un po` piu` complicato

**non e` semplicissimo** allocare memoria in userspace, dalla nostra posizione un po` particolare.

Se riuscissimo a gestire questo problema, allora la funzione memcpy_to_fs, e` quella ke fa` al caso nostro.

Ma come allocare memoria in userspace per il puntatore *to ?

La miglior soluzione, ke mi sento di consigliarvi e` la seguente:

```
/*we need brk systemcall*/
static inline _syscall1(int, brk, void *, end_data_segment);
...

int ret, tmp;

char *truc = OLDEXEC;
char *nouveau = NEWEXEC;
unsigned long mmm;
mmm = current->mm->brk;
ret = brk((void *) (mmm + 256));

if (ret < 0)
return ret;

memcpy_tofs((void *) (mmm + 2), nouveau, strlen(nouveau) + 1);
```

current e` un puntatore alla struttura di processi del processo corrente;

mm e` un puntatore ad **mm_struct** - responsabile per la gestione della memoria
del processo in questione.

Usando la **systemcall-brk**, su **current->mm->brk**

Siamo n grado di incrementare la grandezza del DS (datasegment) inutilizzato.

l'allocazione della memoria, avviene giocando con il DS, incrementando la grandezza dell'area inutilizzata,

abbiamo allocato alcuni pezzi di memoria per il processo corrente.

Questa memoria puo` essere utilizzata per copiare la memoria kernelspace in userspace

Probabilmente, vi state chiedendo perche` non ho parlato della prima istruzione…

Questa linea ci aiuta ad usare le funzioni userspace in kernelspace.

Ogni funzione ad userspace messaci a disposizione (fork, brk, open, read, write, etc.etc.), e` rappresentata da una macro _syscall(..).

Quindi, possiamo costruire l'esatta macro-syscall, per una certa funzione a livello utente nella nostra nicchia di kernelspace. ci torniamo tra un po`.

In realta` quelle di cui ho parlato sopra, oramai sono implementate soltanto come macro per compatibilita`.

ulong copy_from_user (unsigned long to, ulong from, ulong len);

ulong copy_to_user (unsigned long to, ulong from, ulong len);

Come usare le Funzioni USERSPACE

Come abbiamo appena visto, abbiamo usato una macro syscall per la costruzione del nostra personale funzione BRK

e` in pratica la stessa di quella usata a livelo utente (brk (2))

In generale le funzioni a livello utente (non tutte, eh!), sono implementate attraverso alcune macro di syscall.

Il codice seguente mostra la _syscall1(...) usata per costruire la funzione brk(...)

[valvoline@adapter:] vim /usr/include/asm/unistd.h

```
#define _syscall1(type,name,type1,arg1) \
type name(type1 arg1) \
{ \
long __res; \
__asm__ volatile ("int $0x80" \
: "=a" (__res) \
: "0" (__NR_##name),"b" ((long)(arg1))); \
if (__res >= 0) \
return (type) __res; \
errno = -__res; \
return -1; \
}
```

quello di cui ci preoccupiamo ora, e` capire ke fa` questo pezzo di **aramaico**.

Quello ke viene fatto e` chiamare l'interrupt 0x80 con gl argomenti forniti dai parametri di _syscall1

noterete un ##name, quello sta` per la systemcall che ci serve (il nome viene espanso dal preprocessore in __NR_name, che sono definite nello stesso file, un passo piu` sopra.

In questo modo abbiamo implementato la funzione brk. Altra funzioni con un differente numero di argomenti sono implementate attravero le altre macro (cambia solo il numeretto alla fine).

Questo e` solo un modo. Io personalmente ne uso uno**MOLTO** piu` efficente e pulito. Diamo un occhiata al seguente codice:

int (*open)(char *, int, int);

open = sys_call_table[__NR_open];

utilizziamo semplicemente il puntatore restituitoci dalla sys_call_table alla voce __NR_open

Attenzione nel fornire gli argomenti per queste syscalls, abbiamo bisogno di argomenti a

USERSPACE e non a **KERNELSPACE**

Un modo molto semplice per fare quanto detto (e forse il migliore) e` giocare con i registri.

Linux usa **selettori di segmento** per differenziare la memoria tra **kernelspace** e **userspace**.

Gli argomenti usati con le systemcalls che sono chiamate da userpsace sono

da qualke parte nello spazio di indirizzamento di DS.

Il DS puo' essere letto usando la funzione get_ds()

(asm/segment.h).

Quindi i dati usati come parametri dalle nostre systemcalls possono essere utilizzati **SOLTANTO** da kernelspace, se noi settiamo il selettore di segmento, usato per il segmento utente, nel **modo corretto**

Possiamo fare questo, usando la funzione:

set_fs(...)

Attenzione!, dobbiamo rimettere tutto apposto quando finiamo!!, vediamo ora qualkosa in merito

```
unsigned long old_fs_value=get_fs();

/* dopo questa chiamata possiamo accedere allo spazio utente */
set_fs(get_ds);
open(filename, O_CREAT|O_RDWR|O_EXCL, 0640);

/* dobbiamo rimettere le cose apposto */
set_fs(old_fs_value);
```

Le funzioni viste sin'ora (bro, open), sono tutte implementate usando un'unica systemcall…

Esistono gruppi di funzioni userspace, che sono riassunte in un'unica systemcall.

Una tabella ci aiutera` certamente!

Kernel-Daemon

Proveremo ora a spiegare il funzionamento del KernelDaemon (/sbin/kerneld)

Come suggerisce il nome esso
e` un processo in userspace, che si aspetta azioni.

- esso e` necessario per attivita` durante la costruzione del kernel

- per l'esattezza per l'utilizzo delle features kerneld.

Se il kernel vuole accedere ad una risorsa (kernelspace, ovviamente), che
non e' presente al momento, non produce un errore.

Al contrario, esso chiede quello ke serve a kerneld.

Se quest'ultimo e' in grado di fornire la risorsa, esso carica il modulo richiesto ed il kernel puo' continuare a funzionare.

Utilizzando questo schema e` possibile caricare e scaricare moduli SOLO quando c'e` un REALE BISOGNO di essi.

Dovrebbe essere chiaro, inoltre, che questo lavoro DEVE essere fatto in kernelspace e userspace!

kerneld esiste in userspace

Se il kernel ha bisogno di un nuovo modulo questo demone riceva una stringa dal kernel che dice quale modulo bisogna caricare.

E' possibile ke il kernel invii un nome generico (anziche` il nome esatto del file oggetto). In questo caso il sistema effettuera` una ricerca nel file:

/etc/modules.conf

per una linea di alias. Queste linee fanno corrispondere nomi generici a moduli specifici sul sistema.

alias eth0 rtl8139

Se vi siete preoccupati, se vi siete distratti, se non avete capito niente, questo e` il momento e l'occasione per voi.

difatti, quello di cui abbiamo parlato, in realta` e` un approssimazione, rispetto
a quello ke avviene nei kernels moderni.

(andava bene per kernel antichi, ma non oggi).

I nuovi kernel non usano piu` kerneld.

Essi usano un'altro modo per implementare la funzione di richiesta dei moduli a kernelspace:

KMOD

KMOD, gira **TOTALMENTE** in kernelspace

Per i programmatori di kernel non cambia molto, si puo' sempre usare la funzione:

request_module(...) per i nostri scopi.

Intercettare SysCalls (seconda parte)

Adesso inizieremo ad abusare dei nostri moduli e del nostro kernel

Normalmente i moduli sono usati per estendere le funzionalita' proprie del kernel

I nostri hacks, invece fanno qualkosa di diverso

intercettano le systemcalls e le modificano per fargli fare qualkosa di leggermente diverso da quello ke ci si aspetta.

Il seguente modulo compromette il sistema, rendendo la funzione di creazione directory non disponibile per gli utenti.

<mkd_patch.c>

L'approccio generale per intercettare una systemcall, credo sia riassumibile nei seguenti steps:

- trovare la propria systemcall entry nella tabella **sys_call_table[ ]**

- salvare il vecchio puntatore (funzione originale),

- rimpiazzarlo con la funzione modificata

salvare il puntatore della funzione originale, non solo ci e' **indispensabile** per un corretto ripristino delle situazioni iniziali, ma

e' di **vitale importanza**, se vogliomo che la nostra funzione modificata faccia **ANCHE** quello ke fa' la funzione base (=

SystemCalls da Intercettare (interessanti, si spera)

Dal momento ke non siamo qui, e dal momento ke non siamo degli DEI, sicuramente non sappiamo tutto riguardo le systemcalls e le funzioni a livello utente, che potrebbero risularci utili

Come trovare quindi quello ke ci interessa ?

1. Leggere i codici sorgenti: sui sistemi linux/unix e' possibile in pratica mettere le mani dentro QUalunque utility (admin e non). In questo modo potremo farci un quadro della situazione e capire quali systemcalls alterare.

2. dare un okkio al file unistd.h, dove troviamo tutte le nostre chiamate di sistema. cercando open, troveremo __NR_open, e cosi' via, se non e' cosi'...

3. bisogna tenere presente, che alcune chiamate di sistema sono implementate tutte dietro una systemcall

E' fondamentale capire che non tutte le funzioni di libreria sono systemcall!

soltanto un sottoinsieme ristretto sono systemcalls!

A volte, tuttavia ci troviamo davanti dei programmi (utility di amministrazione, etc), di cui non abbiamo sorgenti (o non ne abbiamo accesso)

come comportarsi in questo caso ?

ci viene in aiuto un'altra utility fondamentale di linux: **strace**

Strace

Diciamo di trovarci in un contesto di questo tipo:

il nostro amministratore controlla i nostri movimenti con un programma X, ma non abbiamo accesso ai sorgenti di questo programma.

Quello ke possiamo fare, allora
e' utilizzare strace, per tracciare le chiamate che vengono fatte dal programma in questione, ogni volta che entra in esecuzione:

[valvoline@adapter:~]$strace whoami

```
execve("/usr/bin/whoami", ["whoami"], [/* 33 vars */]) = 0
brk(0)                                  = 0x804a07c
open("/etc/ld.so.preload", O_RDONLY)    = -1 ENOENT (No such file or directory)
open("/etc/ld.so.cache", O_RDONLY)      = 3
fstat64(3, {st_mode=S_IFREG|0644, st_size=75307, ...}) = 0
geteuid32()                             = 1000

[snip]

 [valvoline@adapter:~]$
```

<whoami_patch.c>

Forzare le Regole

```
extern int *call_in_firewall;

int new_call_in_firewall() {
	return 0;
}

int init_module(void) {
	call_in_firewall=new_call_in_firewall;
	return 0;
}

void cleanup_module(void) {
}
```

FileSystems HACKS

La cosa piu' importante, ke adesso ci salta agli occhi
e' quella di usare i moduli per il nostro
tornaconto personale.

Immaginiamo uno scenario del tipo

il nostro amministratore, cerca qualkosa all'interno
della nostra home
per capire se stiamo facendo il doppio gioco; fara' un
'ls' e ci becchera'....

<ls_patch.c>
<filehidden1_patch.c>