Tema 1 - Interceptare apeluri de sistem

Termen de predare: **joi, 17 Martie 2011, ora 23:00**

Scopul temei

Monitorizarea apelurilor de sistem venite din partea proceselor, prin crearea unui interceptor de apeluri de sistem în kernel.

Enunţ

Să se scrie un modul de kernel **sci** care să intercepteze, la cerere, apeluri de sistem. Pentru apelurile de sistem interceptate, modulul trebuie să poată monitoriza anumite procese ce fac apeluri şi să le jurnalizeze în sysklog.

La inserare, modulul va rezerva un apel de sistem nefolosit (`MY_SYSCALL_NO`) şi va aştepta următoarele instrucţiuni (comenzi) din userspace prin respectivul apel:

- `REQUEST_SYSCALL_INTERCEPT` pentru a intercepta un apel de sistem
- `REQUEST_SYSCALL_RELEASE` pentru a renunţa la interceptarea unui apel de sistem
- `REQUEST_START_MONITOR` pentru a porni monitorizarea unui apel de sistem (ce a fost în prealabil interceptat) pentru un anumit proces
- `REQUEST_STOP_MONITOR` pentru a opri monitorizarea unui apel de sistem (ce a fost în prealabil interceptat) pentru un anumit proces

Rutina de tratare a apelului rezervat va avea urmatoarea semnătură:

```
long my_syscall(int cmd, int syscall, int pid);
```

unde:

- **cmd** este comanda aşteptată din userspace
- **syscall** este numărul apelului de sistem la care se referă operaţia
- **pid** este PID-ul procesului pentru care se doreşte pornirea sau oprirea monitorizării. PID-ul este necesar doar pentru operaţiile de monitorizare şi va fi ignorat pentru operaţiile de interceptare.

Interceptarea unui apel de sistem se referă la faptul că modulul va înlocui intrarea din tabela de apeluri de sistem, astfel încât toate apelurile ulterioare să "treacă prin modul" (şi să ajungă în cele din urmă la apelul de sistem original).

Monitorizarea se refera la faptul că modulul va loga în userspace informaţii despre proces şi apel de sistem: numărul apelului de sistem, parametrii apelului de sistem, codul de eroare întors de apelul de sistem (cel original), PID-ul procesului.

Monitorizarea se poate face pentru un proces (**pid** primeşte un PID) sau pentru toate procesele (**pid** primeşte 0).

Înainte de a efectua operaţiile cerute din userspace, modulul va trebui să facă verificări de consistenţă, drepturi, stare şi să semnaleze, daca este cazul, condiţiile de eroare:

- verificări asupra drepturilor [eroare: permisiune/access refuzat]:
  - doar procesele ce aparţin utilizatorului privilegiat vor putea intercepta sau deintercepta un apel de sistem
  - doar procesele ce aparţin utilizatorului privilegiat pot porni sau opri monitorizarea unui apel de sistem pentru toate PID-urile
  - procesele ce aparţin de un utilizator neprivilegiat vor putea opri sau porni monitorizarea unui apel de sistem doar pentru procesele sale
- verificări de consistenţă [eroare: parametru nevalid]:
  - numărul apelului de sistem (se consideră nevalid apelul `MY_SYSCALL_NO` sau `__NR_exit_group`)
  - PID-ul procesului
  - un apel de sistem ce nu a fost interceptat nu poate fi deinterceptat
  - nu se poate monitoriza un apel de sistem care nu a fost interceptat
  - monitorizarea pentru un proces/apel de sistem nu poate fi oprită dacă, în prealabil, nu a fost pornită
- verificări de stare [eroare: ocupat]:
  - un apel de sistem deja interceptat nu poate fi interceptat din nou
  - monitorizarea nu poate fi pornită de două ori pentru acelaşi apel de sistem şi pid

- alte verificări:
  - modulul trebuie să verifice codul de eroare al funcţiilor apelate şi, în cazul în care nu poate continua, să transmită utilizatorului cauza (exemplu: nu se mai poate aloca memorie)
- cazuri speciale:
  - modulul trebuie să oprească monitorizarea pentru procesele ce s-au terminat

Se impune folosirea unei **liste pentru menţinerea informaţiilor despre procesele monitorizate**. Folosirea unui vector de dimensiune fixă se consideră neadecvată pentru că timpul de căutare va fi similar, complexitatea implementării va fi similară, vectorul va avea un număr limitat de intrări. Se recomandă folosirea structurilor existente în kernel.

Pentru că numărul de apeluri de sistem este în general mic (250-300) apeluri de sistem, dar şi pentru o latenţă cât mai mica introdusă de interceptare, se recomandă ca **informaţiile despre apelurile de sistem interceptate să se ţină într-un vector**.

Pentru interceptarea de apeluri de sistem revedeţi informaţiile prezentate în Cursul 2.

Precizări Linux

- tabela cu apelurile de sistem este exportată pentru module doar în 2.4; pentru 2.6 va trebui sa exportaţi manual această tabelă, adăugând în fişierul `arch/x86/kernel/i386_ksyms_32.c` următoarele linii:

```
extern void* sys_call_table[];
EXPORT_SYMBOL(sys_call_table);
```

De asemenea, va trebuie să modificaţi `arch/x86/kernel/entry_32.S` pentru a muta tabela din segmentul `.rodata` în segmentul `.data`:

```
.section .data,"a"
#include "syscall_table_32.S"
```

În plus, pentru că macro-ul `NR_syscalls` nu mai este definit în nucleu, va trebui să definiţi o variabilă specială care să conţină numărul de apeluri de sistem în `arch/x86/kernel/entry_32.S`. Nucleul maşinii virtuale defineşte variabila `my_nr_syscalls` în acest sens.

```
ENTRY(my_nr_syscalls)
.long    nr_syscalls
```

De asemenea, trebuie exportată variabila de mai sus în `arch/x86/kernel/i386_ksyms_32.c`:

```
extern long my_nr_syscalls;
EXPORT_SYMBOL(my_nr_syscalls);
```

Imaginea de pe site are făcute aceste modificări;

- pentru a folosi în modulul vostru cele două variabile exportate (`sys_call_table` şi `my_nr_syscalls`) va trebui să le marcaţi cu extern:

```
extern void *sys_call_table[];
extern long my_nr_syscalls;
```

- cum kernel-ul 2.6 este preemptiv, trebuie să protejaţi accesul la datele comune; este indicat să folosiţi spinlock-uri
- se impune să se folosească apelul de sistem 0 pentru `my_syscall`, care în prezent nu mai este folosit
- utilizatorul privilegiat va fi `root` (uid 0)
- pentru determinarea uid-ului unui proces folosiţi câmpul **cred** din structura ce identifică procesul (`struct task_struct`)
- `task_struct`-ul procesului curent este dat de macroul **current**
- pentru `task_struct`-ul unui proces oarecare folosiţi funcţiile **pid_task()** şi **find_vpid()**
- logarea apelului de sistem se va face cu macro-ul **log_syscall** definit în `sci_lin.h` (fişierul se găseşte în arhiva de teste)
- pentru eliminarea proceselor ce s-au terminat va trebui să interceptaţi apelul de sistem **exit_group**

- codurile de eroare ce trebuie întoarse sunt:
  - `-EINVAL` pentru parametru invalid
  - `-EBUSY` pentru ocupat
  - `-EPERM` pentru access refuzat
  - `-ENOMEM` pentru memorie insuficientă
  - `0` pentru succes

Precizări Windows

- datorită faptului că accesul la tabelele de apeluri de sistem este dependent de versiunea/build-ul sistemului de operare, tema trebuie rezolvată pentru maşina virtuală de pe site; este foarte probabil ca funcţiile helper puse la dispoziţie în `sci_win.h` (fişierul se găseşte în arhiva de teste) să nu funcţioneze pe alte versiuni

- tabelele de apeluri de sistem pot fi accesate din descriptorii asociaţi, definiţi de

```
struct std KeServiceDescriptorTable[2];
struct std *KeServiceDescriptorTableShadow;
```

structura `std` este definită în `sci_win.h`

- la iniţializarea modulului, va trebui să chemaţi funcţia **get_shadow()** care va iniţializa descriptorul pentru tabela shadow; asta pentru că tabela shadow nu este exportată de către kernel

- pentru că nu există apeluri de sistem nefolosite în Windows, va trebui să înlocuiţi tabela de apeluri de sistem cu una nouă, în care să adaugaţi apelul de sistem propriu; atenţie, trebuie să înlocuiţi tabelele de apeluri de sistem atât în descriptorul principal cât şi în shadow

- numărul apelului de sistem **my_syscall** a fost definit în `sci_win.h` la **MY_SYSCALL_NO**; puteţi presupune că această valoare este întotdeauna mai mare decât numărul de apeluri de sistem prezente

- trebuie tratate doar apelurile de sistem din tabela 0

- pentru determinarea utilizatorului unui process folosiţi funcţiile **GetUserOf(pid)** şi **GetCurrentUser**, funcţii definite în `sci_win.h`; pentru a verifica dacă doi utilizatori sunt identici folosiţi funcţia **CheckUsers**

- utilizatorul privilegiat se consideră utilizatorul cu dreptul de încărcat / scos module din kernel; puteţi verifica dacă procesul curent aparţine de un utilizator privilegiat cu funcţia **UserAdmin**, definită în `sci_win.h`

- logarea apelului de sistem se va face prin logarea de pachete `struct packet_log` definite în `sci_win.h`, pachete ce vor fi încapsulate în câmpul **DumpData** al unei intrări de log (vezi **IoWriteErrorLogEntry**) cu codul de eroare 0

- pentru că W2K, WXP, W2k3 sunt preemtive în kernel, trebui să folosiţi spinlock-uri pentru a vă proteja de race-uri; folosiţi pe cât posibil funcţii **Interlocked***

- de la XP se marchează tabela de apeluri de sistem ca read-only; folosiţi macrourile **WPOFF** şi **WPON** (definite în `sci_win.h`) pentru a opri, respectiv porni protecţia la scriere

- pentru eliminarea proceselor ce s-au terminat din structurile folosite vedeţi **PsSetCreateProcessNotifyRoutine**

- codurile de eroare ce trebuie întoarse sunt:
  - `STATUS_INVALID_PARAMETER` pentru parametru invalid
  - `STATUS_DEVICE_BUSY` pentru ocupat
  - `STATUS_ACCESS_DENIED` pentru access refuzat
  - `STATUS_NO_MEMORY` pentru memorie insuficientă
  - `STATUS_SUCCESS` pentru succes

Exemplu de cod pentru interceptarea apelurilor de sistem în Windows

```
void (*f)();
void intercept(int syscall)
{
```

```
    int syscall_table, syscall_index;
    syscall_table=syscall>>12;
    syscall_index=syscall&0x0000FFF;
    f=KeServiceDescriptorTable[syscall_table]->st[syscall_index];
    KeServiceDescriptorTableShadow[syscall_table]->st[syscall_index]=interceptor;
}

NTSTATUS interceptor()
{
    int syscall, params, syscall_table,
    syscall_index, r;
    void *old_stack, *new_stack;
    asm mov syscall, eax
    syscall_table=syscall>>12;
    syscall_index=syscall&0x0000FFF;
    params=KeServiceDescriptorTable[syscall_table]->spt[syscall_index];
    mov old_stack, ebp
    add old_stack, 8
    sub esp, params
    mov new_stack, esp
    memcpy(new_stack, old_stack, params);
    r = f();
    DbgPrint(?%d: %d\n", syscall, r);
    return r;
}
```

Testare

Pentru simplificarea procesului de corectare al temelor, dar şi pentru a reduce greşelile temelor trimise, corectarea temelor se va face automat cu ajutorul unor teste publice (linux md5:adc363c555b81d063fa61bdd7949d62c, windows md5:dd79fd6529513fa46020087ba50c2295). Testele presupun că numele modulului de kernel este `sci` atât pe Linux cât şi pe Windows.

Linux

Testul se foloseşte de utilizatorul `nobody` care ar trebui să existe pe toate sistemele Linux. Din această cauză testul trebuie să fie plasat intr-un director public.

Windows

Utilizatorul cu care veţi rula testul trebuie să aibă drepturile "Act as part of the operating system", "Create a token object" şi "Replace a process level token". Pentru a configura aceste drepturi: Start Administrative Tools Local Security Policy Local Policies User Rights Assignments.

Depunctări

Depunctările generale pentru teme se găsesc pe pagina de Indicaţii generale.

Depunctări pentru probleme constatate în implementarea temei prezente:

- -0.5 implementare incorecta a cerintelor temei
- -0.2 nu se implementeaza eliminarea proceselor care s-au terminat
- -1.0 nu se folosesc liste in implementare
- -0.2 lipsa sincronizari sau sincronizare incompleta / incorecta
  - - 0.1 lipsa sincronizare accces lista procese
  - - 0.1 lipsa sincronizare vector de apeluri de sistem
- -0.1 utilizare functii la nivele IRQL necorespunzatoare
- -0.0 observatii
- -0.1 pentru fiecare test picat

În cazuri excepţionale (tema trece testele prin nerespectarea cerinţelor) şi în cazul în care tema nu trece toate testele se poate scădea mai mult decat este menţionat mai sus.

Întrebări

Pentru întrebări legate de temă puteţi consulta arhivele listei de discuţii sau puteţi trimite un e-mail (trebuie să fiţi înregistraţi).

FAQ

Q: Ce ar trebui să se întample în următoarele situaţii:

1. s-a cerut monitorizarea tuturor proceselor, şi apoi se cere demonitorizarea unui proces specific.
2. s-a cerut monitorizarea tuturor proceselor, şi apoi se cere monitorizarea unui proces specific.
3. s-a cerut monitorizarea unui proces specific, şi apoi se cere demonitorizarea tuturor.

A: Nu se va testa asemenea condiţii. Este recomandată o abordare simplă, în care pid=0 este considerat la fel ca PID -ul celorlalte procese.

(http://cursuri.cs.pub.ro/pipermail/pso/2009-March/003030.html)

Q: La tema 1 pe windows, structura std este un pic ambigua. Ce inseamna fiecare camp ?

```
struct std {\
  void **st;    /* service table */\
  int *ct;        /* counter table */\
  int ls;            /* last service no */\
  unsigned char *spt;    /* service parameter table */\
};
```

A:

```
 void **st;    /* service table */
```

Un pointer la service call table [un vector de adrese ale handlerelor pentru fiecare system call, indexat dupa id-ul system call-ului]

```
 int *ct;        /* counter table */
```

Pentru profiling, folosit doar in debug builds. Un pointer la un vector de countere, indexat tot dupa id-ul system call-ului, care numara de cate ori a fost apelat fiecare system call.

```
 int ls;            /* last service no */
```

Numarul de intrari din service table / counter table / service parameter table.

```
 unsigned char *spt;    /* service parameter table */
```

Numarul de bytes ocupati de argumentele fiecarui system call.

Pentru dispatcherul vostru nu veti folosi decat service table si last service no, dar cand creati noua tabela este nevoie sa duplicati si service parameter table.

Pentru mai multe informatii vezi [1], descrierea KeAddSystemServiceTable [2] si trap.asm, sursa dispatcherului de system calls pentru windows [3].

[1] http://www.honeynet.org/node/438

[2] http://koala.cs.pub.ro/lxr/wrk/ntos/ke/miscc.c#L729

[3] http://koala.cs.pub.ro/lxr/wrk/ntos/ke/i386/trap.asm