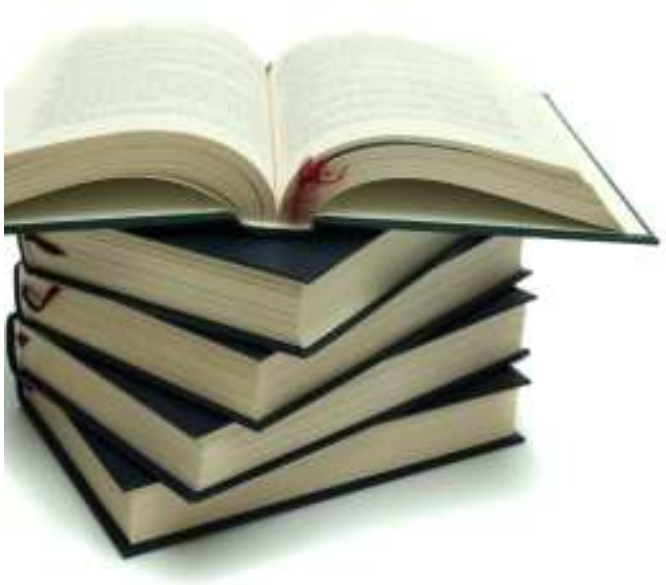
so2
kernel crunch

Cursul 3



4

Apeluri de sistem (II)

19 martie 2009

Test Curs 3

- Care este un dezavantaj al transmiterii argumentelor unui apel de sistem prin registre?
- Pe un sistem Linux se realizeaza apelul read(fd, buf, num). Explicati in pseudo-assembly cum se va realiza acest apel de sistem pe un sistem Linux care foloseste sysenter.
- Creați stack frame-ul pentru o funcție ce are un parametru şi două variabile locale.

Apeluri de sistem

- Apeluri de sistem
- GAS, stack frame
- Accesarea spațiului utilizator în Linux
- GCC extended ASM
- Implementarea apelurilor de sistem în Windows

Bibliografie

- UTLK: capitolul 10
- LKD: capitolul 5
- WI: capitolul 3 (System Service Dispatching)

VDSO

- Problema: folosirea int 0x80/sysenter depinde de versiunea procesorului, versiunea kernelului şi versiunea libc
- Soluția: Virtual Dynamic Shared Object:
  - kernelul decide de instrucțiune de trap să se folosească
  - Pentru genericitate, kernelul ca genera chiar el secvența de instrucțiuni ce trebuie folosită
  - Respectiva secvență va fi mapată în spațiul utilizator în zona VDSO

VDSO (2)

```
$ cat /proc/$$/maps
...
bfac5000-bfada000 rw-p bffeb000 00:00 0          [stack]
ffffe000-fffff000 r-xp 00000000 00:00 0          [vdso]

$ dd if=/proc/self/mem of=linux-gate.so bs=4096
skip=$[0xffffe] count=1
```

VDSO (3)

```
$ objdump -d linux-gate.so –no-show-raw-insn

ffffe400 <__kernel_vsyscall>:
ffffe400:       push   %ecx
ffffe401:       push   %edx
ffffe402:       push   %ebp
ffffe403:       mov    %esp,%ebp
ffffe405:       sysenter
ffffe407:       nop
```

Vsyscalls (virtual system calls)

- „Apeluri de sistem” care rulează direct din user-space
- În zone VSDO kernelul mapează cod ce rulează direct din user-space şi accesează direct date kernel, date mapate şi ele în VDSO (dar R/O)
- Vgettimeofday, Vgetcpu (x86_64)

Accesarea spațiului utilizator

- Primitive specializate: get_user, put_user, copy_from_user, copy_to_user
- Verificarea pointerului şi tratarea eventualului page-fault la acces invalid se face intern

```
/* daca s-a generat un page fault datorita unui access
invalid primitivele intorc o valoare diferita de zero */
if (copy_from_user(kernel_buffer, user_buffer, size))
      return -EFAULT;
```

get_user

```
#define get_user(x,ptr) \
({ \
    int __ret_gu; \
    unsigned long __val_gu; \
    switch(sizeof (*(ptr))) { \
    case 1: \
        __get_user_x(1,__ret_gu,__val_gu,ptr); \
        break; \
    case 2: \
        __get_user_x(2,__ret_gu,__val_gu,ptr); \
       break; \
...
    __ret_gu; \
})
```

__get_user_x

```
#define __get_user_x(size,ret,x,ptr) \
  __asm__ __volatile__( \
   "call __get_user_" #size \
   :"=a" (ret),"=d" (x) :"0" (ptr) \
)

ENTRY(__get_user_1)
        GET_THREAD_INFO(%edx)
        cmpl TI_addr_limit(%edx),%eax
        jae bad_get_user
1:      movzbl (%eax),%edx
        xorl %eax,%eax
        ret
ENDPROC(__get_user_1)
```

GCC extented asm

- Folosirea de cod ASM împreună cu cod C
- Se pastrează claritatea codului
- Se “optimizează de mâna”
- Se accesează resurse altfel neaccesibile din compilator (regiştri speciali, instrucțiuni speciale)

asm / __asm__

```
asm(intruction_template
    : constraints_1 output_operand_1,
      constraints_2 output_operand_2,
      ...,
      constraints_n output_operand_n
    : constraints_n+1 input_operand_n+1,
      constraints_n+2 input_operand_n+2,
      ...,
      constraints_n+m input_operand_n+m,
    : clobbered_regmem_1, clobbered_regmem_2,
      ...,
      clobbered_regmem_n);
```

Instruction template

```
instruction_mnemonic [ operand_asm | operand_no ], ...
```

- operand_asm = „%%registru”
- operand_no = „%x” unde x este numărul operandului

```
unsigned long address;
asm(“mov %%cr2, %0”: (address));
printk(“Adress: 0x%x”, address);
```

Constrângeri

- “m” = orice fel de operator memorie
- “r” = orice fel de operator registru
- “i” = un intreg imediat (intreg cu valoarea cunoscuta la momentul asamblarii)
- “0”, “1”, ..., “9” = impune sa se foloseasca acelasi operand ca cel indicat
- “=” = operandul este write-only; se foloseste pentru operanzii de output

Exemplu

```
void printk_cr2()
{
   unsigned long address;
   asm(“mov %%cr2, %0”: "=m" (address));
   printk(“0x%x”, address);
}
```

Exemplu (2)

```
00000000 <printk_cr2>:
   0:   push   %ebp
   1:   mov    %esp,%ebp
   3:   sub    $0x18,%esp
   6:   mov    %cr2,%eax
   9:   mov    %eax,-0x4(%ebp)
   c:   mov    -0x4(%ebp),%eax
   f:   mov    %eax,0x4(%esp)
  13:   movl   $0x0,(%esp)
          16: R_386_32    .rodata
  1a:   call   1b <x+0x1b>
          1b: R_386_PC32  printk
  1f:   leave
  20:   ret
```

Efecte laterale

- Unele instrucțiuni pot modifica şi alți regiştri / zone de memorie decât operanzii specificați
- Operanzii clobbered_regmem sunt folosiți pentru a indica compilatorului care regiştri / zone de memorie sunt afectate de blocul asm
- (blocul asm şi codul generat de compilator pot folosi acelaşi registru zonă de memorie)

__get_user_x

```
#define __get_user_x(size,ret,x,ptr) \
 __asm__ __volatile__( \
  "call __get_user_" #size \
  :"=a" (ret),"=d" (x) :"0" (ptr) \
)

ENTRY(__get_user_1)
        GET_THREAD_INFO(%edx)
        cmpl TI_addr_limit(%edx),%eax
        jae bad_get_user
1:      movzbl (%eax),%edx
        xorl %eax,%eax
        ret
ENDPROC(__get_user_1)
```

Se verifică ca pointerul să fie în user-space

Instrucțiunea de copiere şi adresa ei

__get_user_x (2)

```
bad_get_user:
        xorl %edx,%edx
        movl $-14,%eax
        ret
END(bad_get_user)

.section __ex_table,"a"
        .long 1b,bad_get_user
        .long 2b,bad_get_user
        .long 3b,bad_get_user
.previous
```

Adresa instrucțiunii ce face accesul în userspace din cadrul __get_user_1

Tabela de excepții

- Un vector de perechi (fault instruction address, fix-up code address)
- Generat la compilare în cadrul secțiunii __ex_table
- Rutina de tratare a page fault-ului va căuta adresa instrucțiunii ce a generat fault-ul în tabelă, şi dacă o găseşte va sări la adresa asociată

Accesul la datele din secțiunile speciale

- Scriptul de link editare (arch/*/kernel/*.lds.S) gardează secțiunea cu simboluri de geneul __start / __stop
- Exemplu:

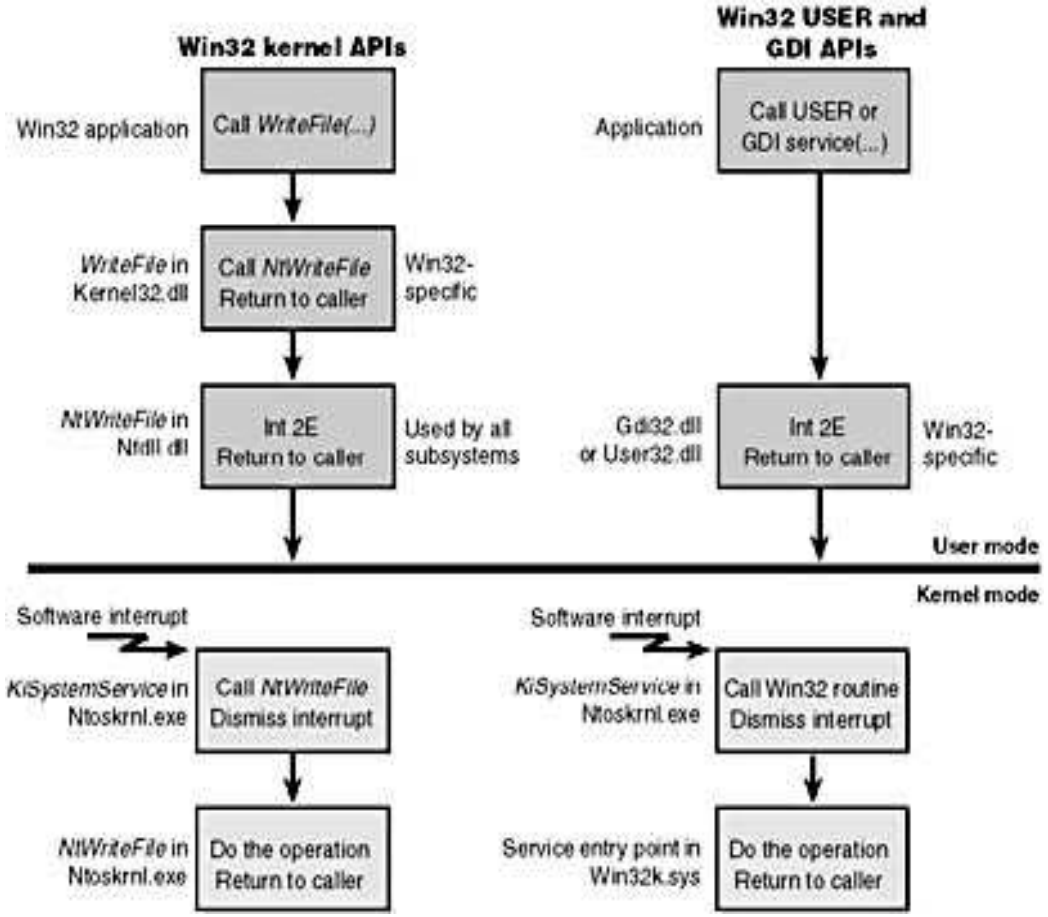
```
. = ALIGN(16);  /* Exception table */
__ex_table : AT(ADDR(__ex_table) - LOAD_OFFSET) {
    __start___ex_table = .;
    *(__ex_table)
    __stop___ex_table = .;
}
```

fixup_exception

```
int fixup_exception(struct pt_regs *regs)
{
  const struct exception_table_entry *fixup;

  fixup = search_exception_tables(regs->eip);
  if (fixup) {
    regs->eip = fixup->fixup;
    return 1;
  }

  return 0;
}
```

Setăm instrucțiunea de la care se continuă execuția după ce ieşim din handler-ul de tratare al page-fault-ului

Apeluri de sistem în Windows



Apeluri de sistem în Windows (2)

- Fluxul de execuție:
  - **WriteFile, ReadFile** (kernel32.dll) sunt transformate în apeluri mai complexe
  - **NtWriteFile, NtReadFile** (ntdll.dll) efectuează apelul de sistem
- Numărul apelului de sistem se pune în eax
- Se pun pe stiva user-space parametrii
- ebx <- pointer către parametri; int 0x2e
- edx <- pointer către parametri; sysenter

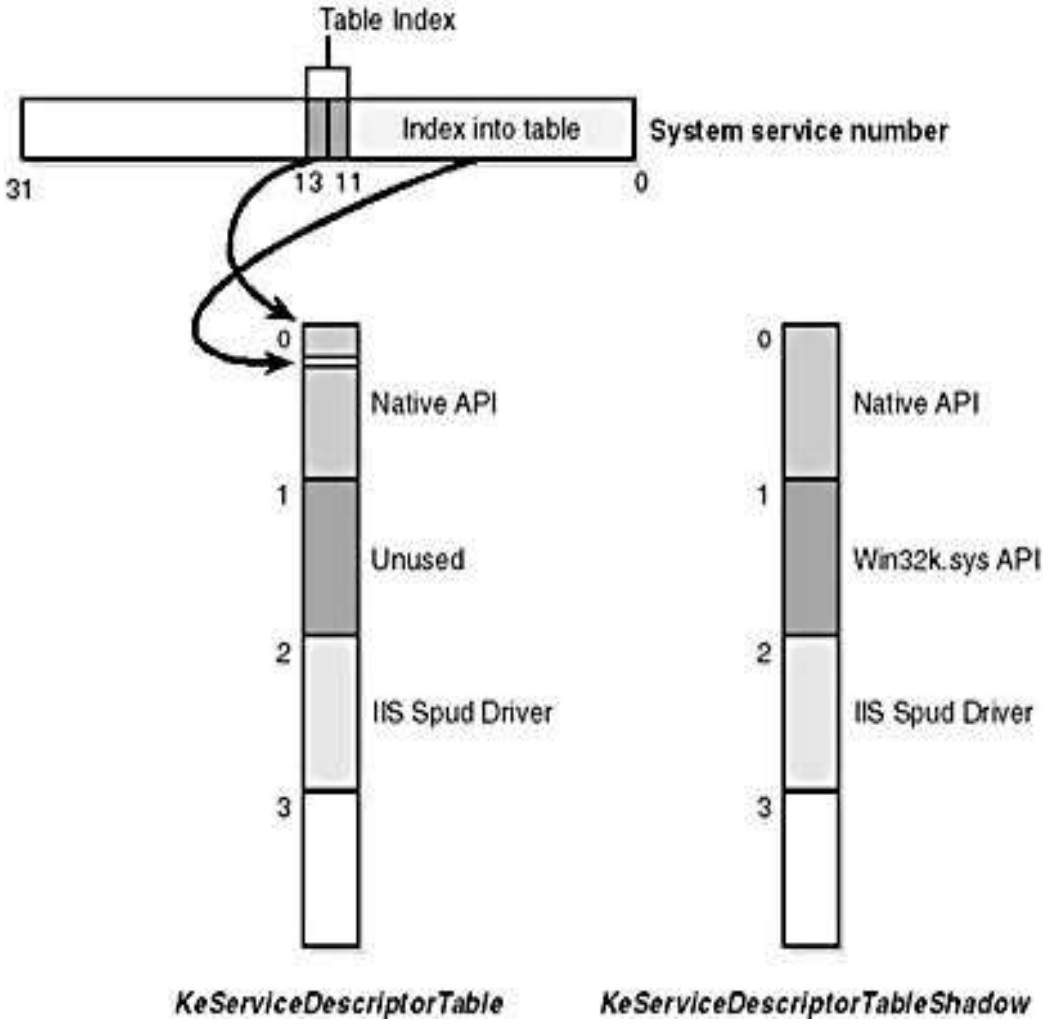
Exemplu de invocare

```
ntdll!NtReadFile:
77f5bfa8 mov   eax,0xb7
77f5bfad mov   edx,0x7ffe0300
77f5bfb2 call  edx
77f5bfb4 ret   0x24
...
SharedUserData!SystemCallStub:
7ffe0300 mov     edx,esp
7ffe0302 sysenter
7ffe0304 ret
```

KiSystemService

- Rutina de dispatch
  - identifică tabela şi indexul în tabela de servicii sistem
  - determină numărul de parametri ai serviciului sitem
  - copiază parametrii pe stiva kernel
  - identifică şi rulează rutina de tratare a serviciului
- Serviciile sunt grupate în tabele
  - numărul serviciului este compus din
    - 2 biți pentru tabele: 4 tabele
    - 12 biți pentru index: maxim 4096 intrări
  - fiecare proces poate avea teoretic tabele proproii

KeServiceDescriptorTable



KeServiceDescriptorTable (2)

```
struct service_descriptor_table {
        void **service_table;
        int *counter_table;
        int services_no;
        unsigned char *service_parameters_table;
} KeServiceDescriptorTable[4];
```

- Există două tabele de descriptori: cea principală şi cea shadow
- Cea shadow foloseşte tabela unu pentru apeluri de sistem GDI (şi zero pentru apelurile Win32)
- W2k3 SP1 nu mai permite înregistrarea de noi tabele de apeluri de sistem în afară de cele două

KeServiceDescriptorTable (3)

- **service_table** – un vector de pointeri către adresele funcțiilor de tratare ale apelurilor de sistem
- **counter_table** – un vector ce conține numărul de apeluri de sistem efectuate; este folosit doar atunci când se rulează în mod debug
- **services_no** – numărul de elemente din service_table şi service_parameters_table
- **service_parameters_table** – un vector ce conține dimensiunea pe stivă a parametrilor apelului de sistem

Exercițiu

Scrieți în C & ASM secvența de cod ce poate fi folosită dintr-un modul kernel pentru afişarea numărului apelului de sistem şi a codului de eroare la fiecare apel de sistem X.

Rezolvare

```
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;
}
```

Rezolvare (2)

```
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];
```

Rezolvare (3)

```
        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;

}
```

Rezolvare (4)

- Este doar un exercițiu didactic, în practică nu se recomandă interceptarea apelurilor de sistem
  - Probleme: race-uri, apelurile de sistem in Windows nu sunt backwards compatible
  - Folosiți filesystem filter drivers

Se verifică ca pointerul să fie în user-space

```
try {
    ProbeForRead(Buffer, Length, sizeof(UCHAR));
    tmp = ExAllocatePool (NonPagedPool, Length);
    RtlCopyMemory(tmp, Buffer, Length );
} except (EXCEPTION_EXECUTE_HANDLER) {
    IopExceptionCleanup(...);
    return GetExceptionCode();
}
```

Page fault la acces invalid

Structured Exception Handling

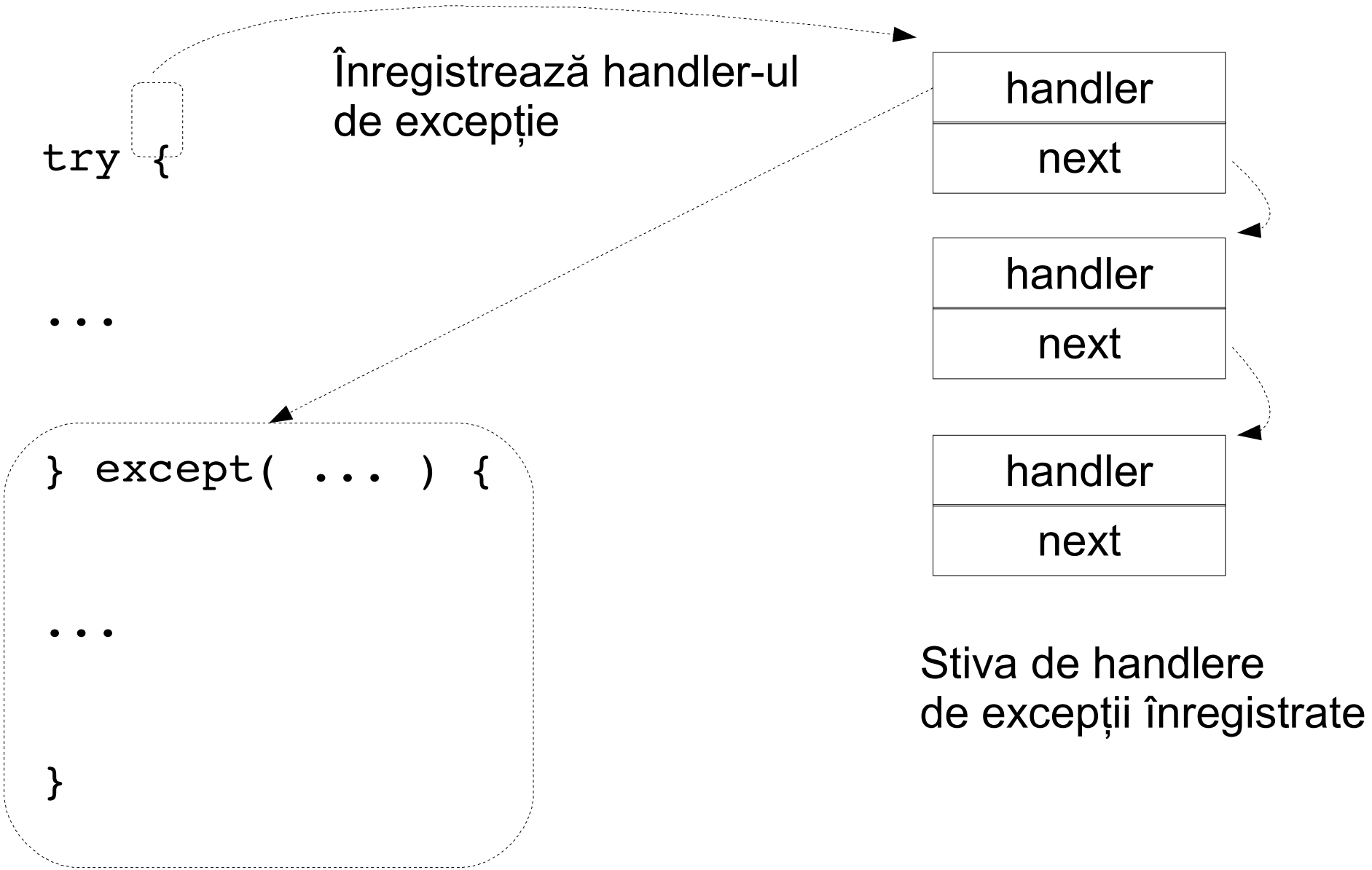
```
try {
    ...
} except( ... ) {
    ...
}
```

Dacă se generează page-fault în această secțiune ...

... şi expresia din această secțiune întoarce o valoare pozitivă

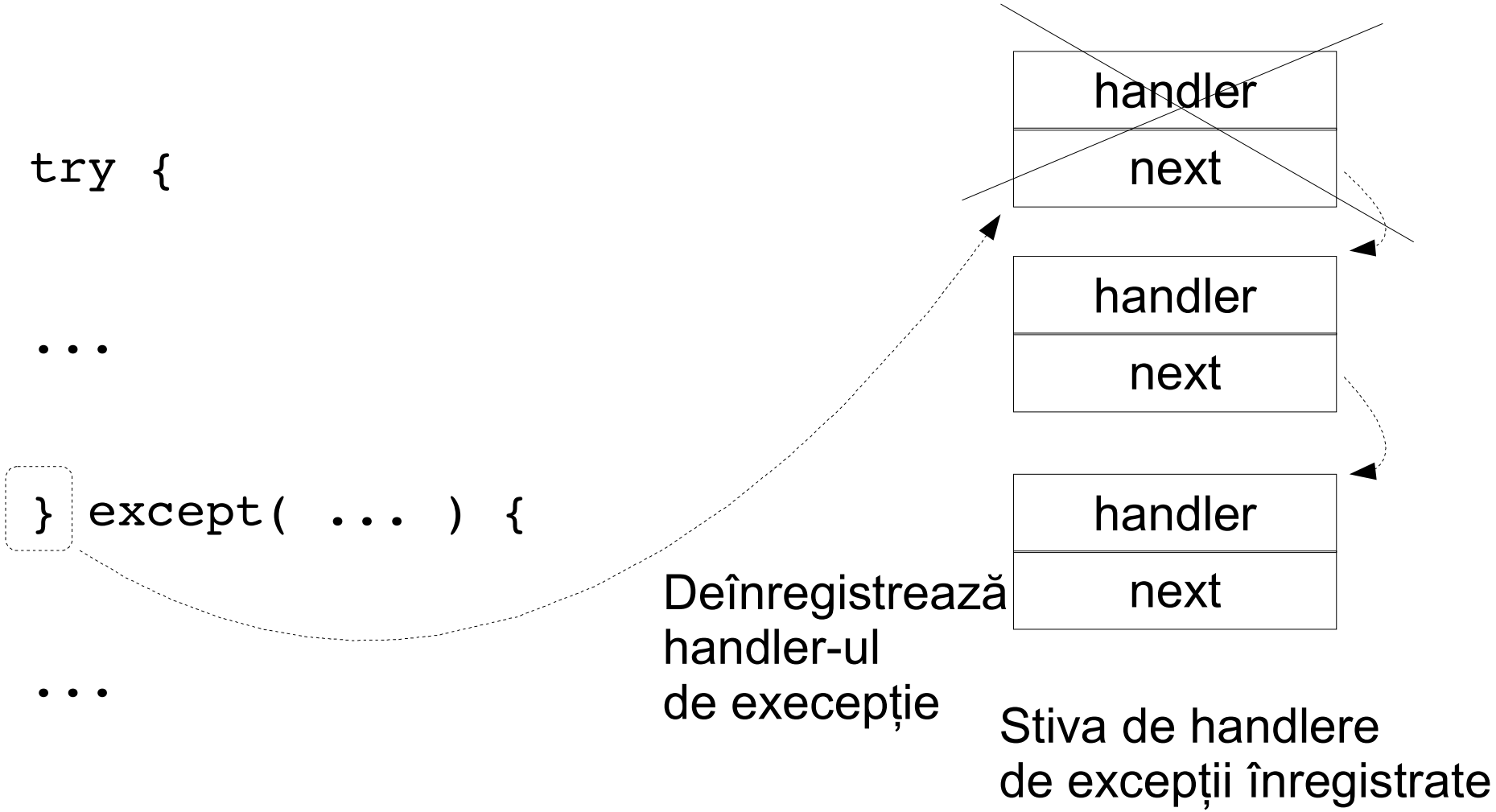
... page-fault-ul este tratat în această secțiune

Structured Exception Handling (2)

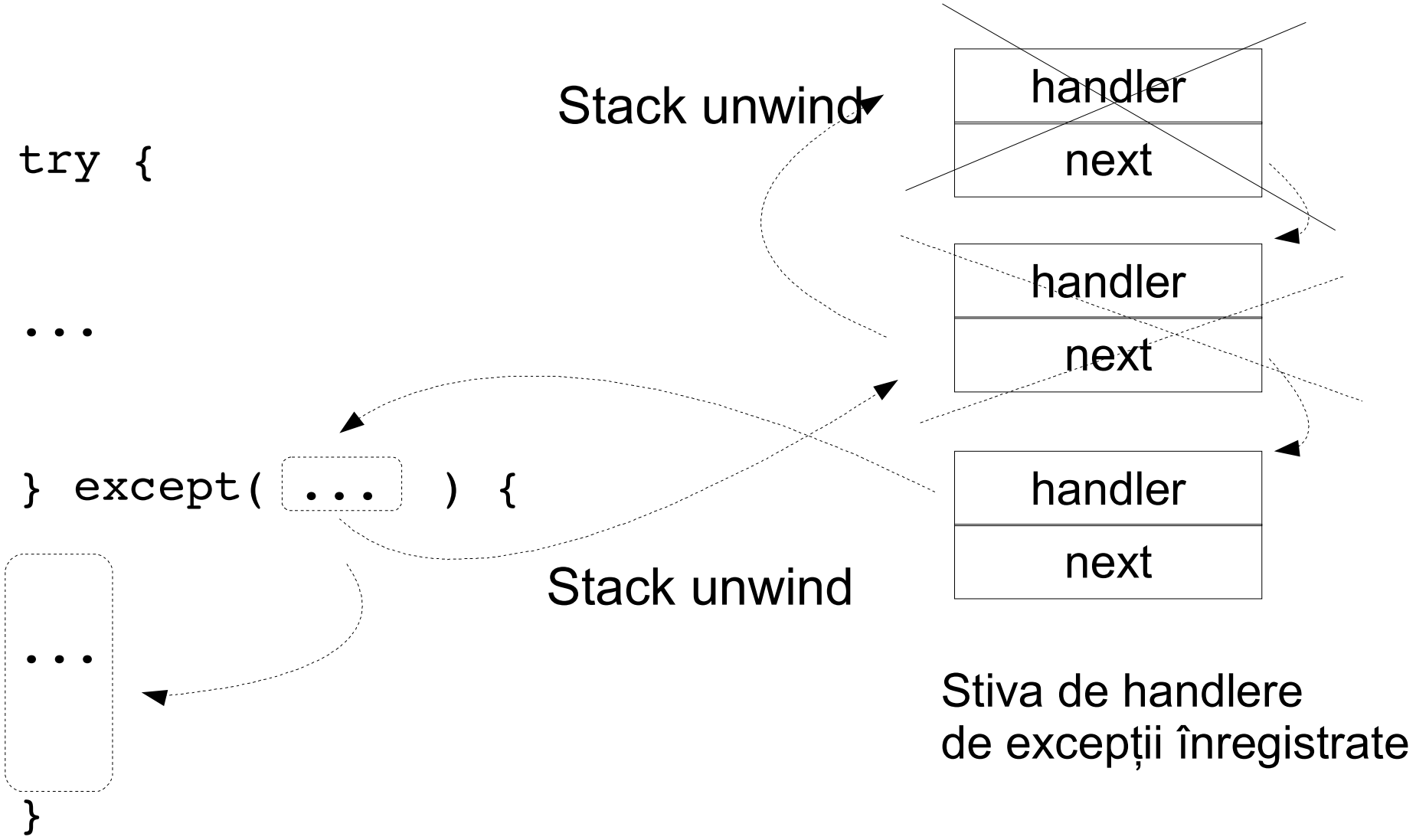


Structured Exception Handling (3)

```
try {

...

} except( ... ) {

...

}
```



Stack unwind



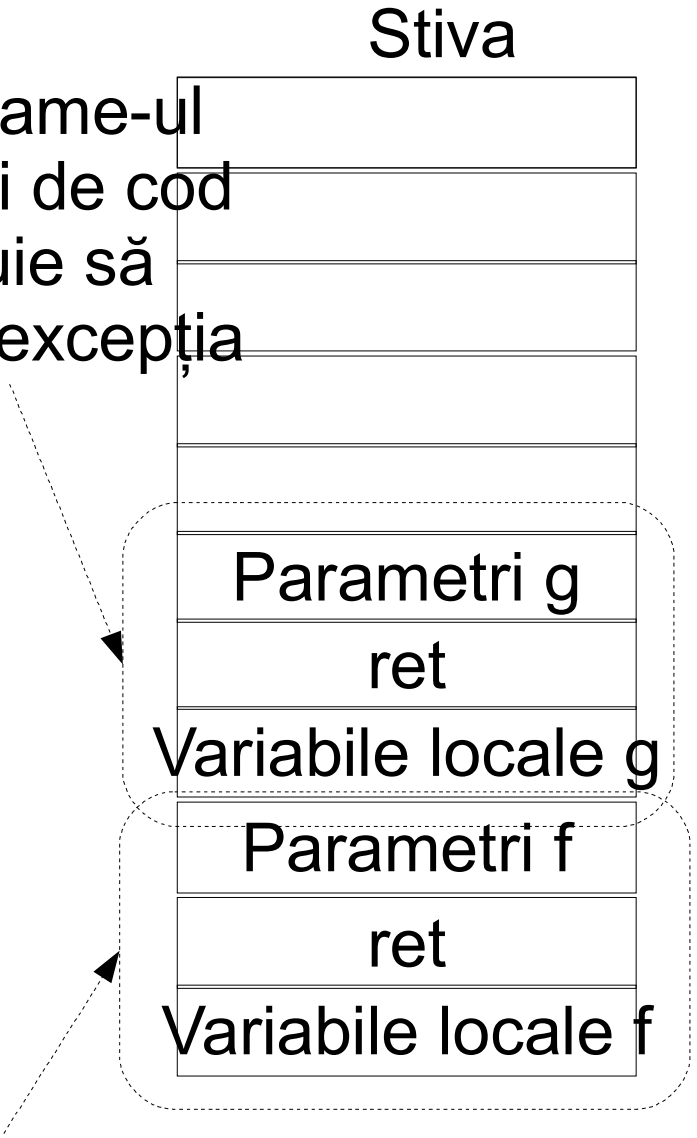
Stack unwind (2)

- Merge înapoi pe stiva de handlere de excepții înregistrate şi
  - Deînregistrează handler-ul de tratare a excepției
  - Curăță stack frame-ul asociat cu handler-ul de tratare al excepției (trebuie să ajungem sa executăm rutina de tratare în stack frame-ul original)

Exemplu

```
void g()
{
try {
   f();
} except(...) {
   ...
}
}
```

```
void f()
{
try {
   ...
} except(...) {
   ...
}
}
```



Intrebări

?