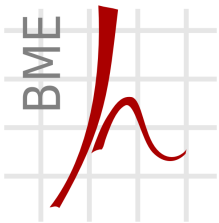
Biztonságos programozás – Puffer túlcsordulásos támadások

Intel x68



Híradástechnikai Tanszék

Izsó Tamás

2015. október 5.

Section 1

Puffer túlcsordulás sebezhetőség elleni védelem

Hogyan tovább?

Mit tegyünk?

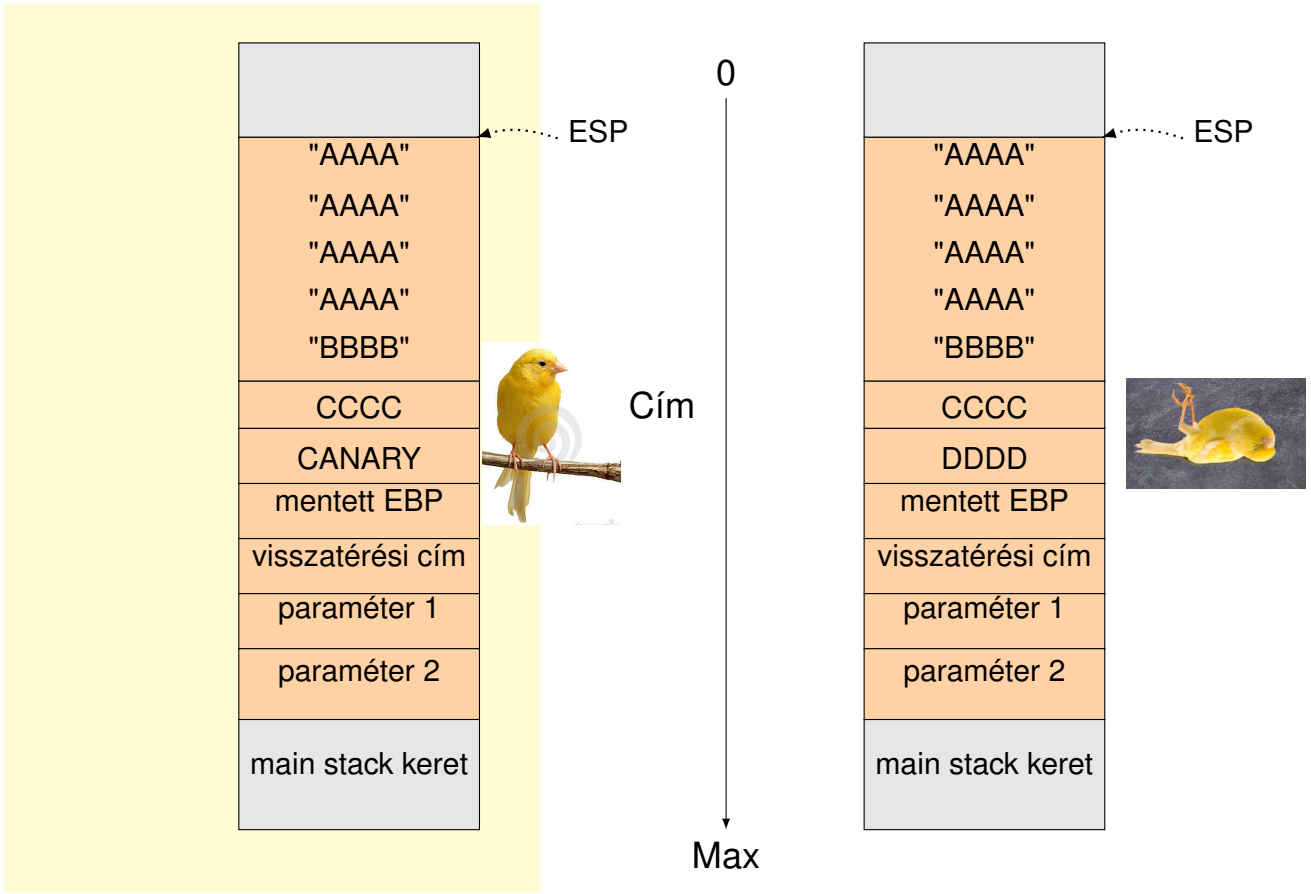
Kanári (más néven stack süti)

Tegyünk a stackre egy értéket, és visszatérés előtt ellenőrizzünk, hogy nem írta-e valaki felül. Ezt az értéket nevezzük kanárinak, vagy stack sütinek.

Kanári típusa:

- Konstans. Nem sokat ér.
- Terminátor. Konstans, de olyan értéket tartalmaz, amit nem lehet bevinni. Például string puffer esetén 0-át.
- Véletlen. Egy mestersüti van, de minden program indításkor más az értéke. A függvényenkénti azonos értékek gyengítheti a védelem erősségét.
- Véletlen + xor. Minden hívásnál más az értéke, mivel a véletlen értékkel és a stack címével kizáró vagy műveletet végzünk.

Kanárit tartalmazó stack képe



Visual Studio /GS+

Visual Studio a /GS+ opció esetén használ kanárit (stack sütit).
Ez az alapértelmezés, letíltása /GS- .
Példaprogram a jól ismert authenticate függvény:

```
21 int authenticate(char *username, char *password) {
22   int authenticated;
23   char buffer[1024];
24
25   authenticated = verify_password(username, password);
26   if(authenticated == 0)   {
27     sprintf(buffer,
28       "password is incorrect for user %s\n",
29       username);
30     log("%s", buffer);
31   }
32
33   return authenticated;
34 }
```

Visual Studio kanárija

```
int authenticate(char *username, char *password) {
004114A0 55                   push        ebp
004114A1 8B EC                mov         ebp,esp
004114A3 81 EC 48 04 00 00    sub          esp,448h
004114A9 A1 00 70 41 00       mov         eax,dword ptr [___security_cookie (417000h)]
004114AE 33 C5                xor         eax,ebp
004114B0 89 45 FC             mov         dword ptr [ebp-4],eax
004114B3 53                   push        ebx
004114B4 56                   push        esi
004114B5 57                   push        edi
        int authenticated;
        char buffer[1024];

    ----- kihagyott sorok ----

004114FF 5F                   pop         edi
00411500 5E                   pop         esi
00411501 5B                   pop         ebx
00411502 8B 4D FC             mov         ecx,dword ptr [ebp-4]
00411505 33 CD                xor         ecx,ebp
00411507 E8 1C FB FF FF       call        @ILT+35(@__security_check_cookie@4) (411028h)
0041150C 8B E5                mov         esp,ebp
0041150E 5D                   pop         ebp
0041150F C3                   ret
```

Hogyan keletkezik a mester süti érték

Illusztráció!

Windows 2003 esetén:

```
#include <stdio.h>
#include <windows.h>
int main()
{
    FILETIME ft;
    unsigned int Cookie=0;
    unsigned int tmp=0;
    unsigned int *ptr=0;
    LARGE_INTEGER perfcount;
    GetSystemTimeAsFileTime(&ft);
    Cookie = ft.dwHighDateTime ^ ft.dwLowDateTime;
    Cookie = Cookie ^ GetCurrentProcessId();
    Cookie = Cookie ^ GetCurrentThreadId();
    Cookie = Cookie ^ GetTickCount();
    QueryPerformanceCounter(&perfcount);
    ptr = (unsigned int)&perfcount;
    tmp = *(ptr+1) ^ *ptr;
    Cookie = Cookie ^ tmp;
    printf("Cookie: %.8X\n",Cookie);
    return 0;
}
```

Stack süti megkerülése

- Ha a stack süti konstans értékű, akkor triviális.
- Ha a puffer után van értékes adat, amit felül lehet írni, anélkül hogy sérülne a stack süti, akkor sem ér semmit ez a védelem. A mi példánkban a **int** authenticated változó továbbra is támadható 993 karakteres jelszó megadással. Ha a fordító átrendezte volna a stack-en a két változót, akkor ezt nem tehettük volna meg.
- Kiszámítjuk a processzhez *tartozó mester sütit*, bár nem valószínű, hogy minden adat a kezünkbe van.
- Újabb módszer alkalmazása.

Section 2

SEH

Interrupt és kivételkezelés

- Interrupt és kivételdobás események megszakítják a processzor utasítás végrehajtásnak normális folyamatát.
- Az interrupt lehet szinkron, és aszinkron. Aszinkron interrupt esetén a processzor képes az utasítás végrehajtás ismétlésére.
- A megszakítást mindig az operációs rendszer kapja el és továbbítja.

Hibakezelés C-ben

```
#include <stdio.h>
#include <Windows.h>

void TestExceptions() {
    int a, b=10;
    printf("Triggering␣SEH␣exception\n");
    a = b / 0;
}

int main() {
     __try
     {
          TestExceptions();
     }
     __except( GetExceptionCode() == EXCEPTION_INT_DIVIDE_BY_ZERO ?
               EXCEPTION_EXECUTE_HANDLER : EXCEPTION_CONTINUE_SEARCH)
     {
         printf("Hiba␣lekezelve\n");
     }
     printf("Nincs␣hibakezelés\n");
     TestExceptions();
     return 0;
}
```

__try és __except nem szabványos C nyelvi elem

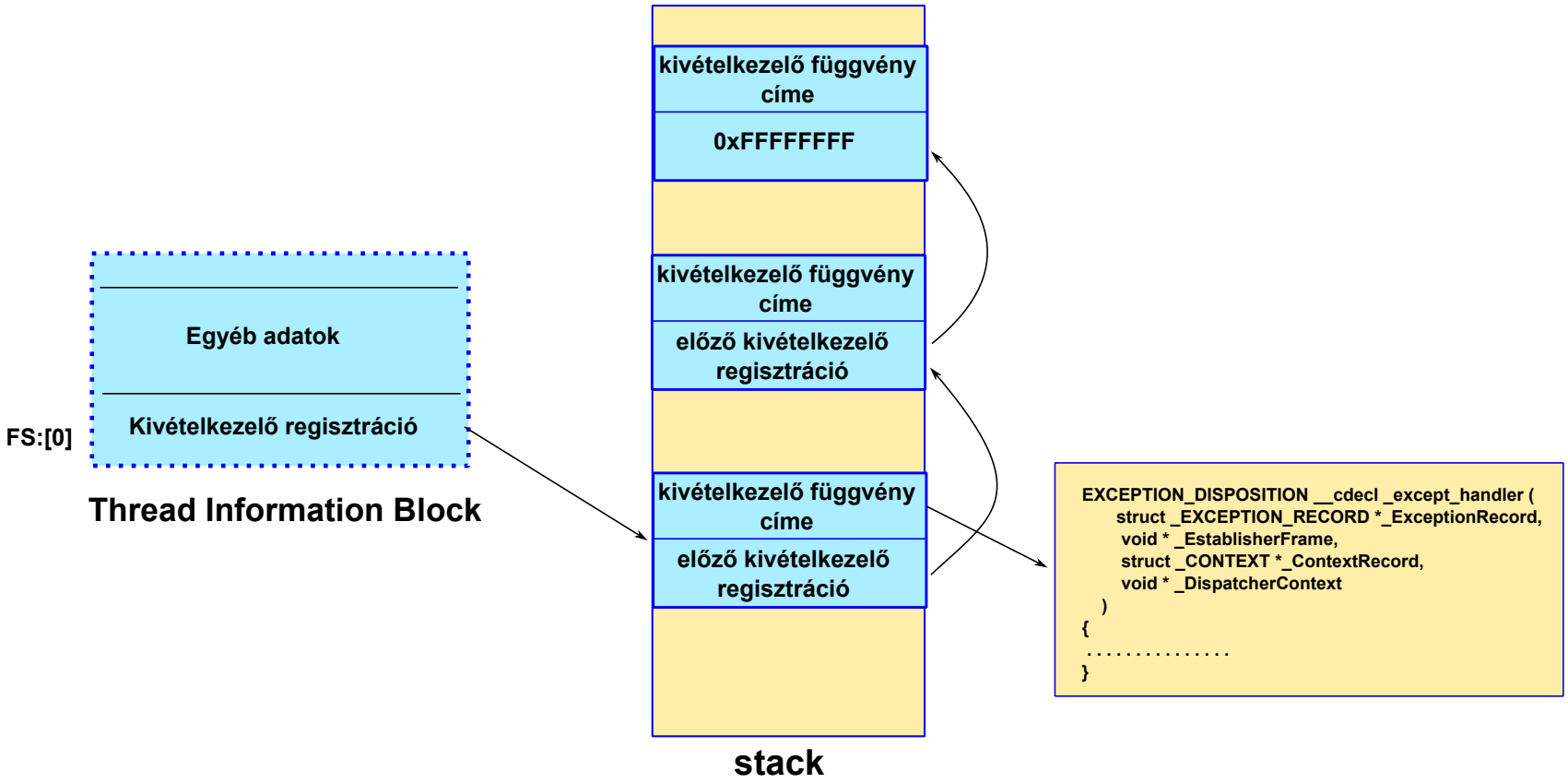
- EXCEPTION_EXECUTE_HANDLER itt kezeljük le a megszakítást.
- EXCEPTION_CONTINUE_SEARCH következő regisztrált megszakításkezelő keresése.

_EXCEPTION_RECORD-ban az Exception

Illusztráció!

EXCEPTION_ACCESS_VIOLATION
EXCEPTION_DATATYPE_MISALIGNMENT
EXCEPTION_BREAKPOINT
EXCEPTION_SINGLE_STEP
EXCEPTION_ARRAY_BOUNDS_EXCEEDED
EXCEPTION_FLT_DENORMAL_OPERAND
EXCEPTION_FLT_DIVIDE_BY_ZERO
EXCEPTION_FLT_INEXACT_RESULT
EXCEPTION_FLT_INVALID_OPERATION
EXCEPTION_FLT_OVERFLOW
EXCEPTION_FLT_STACK_CHECK
EXCEPTION_FLT_UNDERFLOW
EXCEPTION_INT_DIVIDE_BY_ZERO
EXCEPTION_INT_OVERFLOW
EXCEPTION_PRIV_INSTRUCTION
EXCEPTION_IN_PAGE_ERROR
EXCEPTION_ILLEGAL_INSTRUCTION
EXCEPTION_NONCONTINUABLE_EXCEPTION
EXCEPTION_STACK_OVERFLOW
EXCEPTION_INVALID_DISPOSITION
EXCEPTION_GUARD_PAGE
EXCEPTION_INVALID_HANDLE
EXCEPTION_POSSIBLE_DEADLOCK

Strukturált exception handler — SEH — regisztráció



Exception rekord

```
11 struct EXCEPTION_REGISTRATION_FIELD
12 {
13     struct EXCEPTION_REGISTRATION_FIELD* prev;
14     pexcept_handler  handler;
15 };

19 struct EXCEPTION_REGISTRATION_FIELD* pRegister=0;
20 struct EXCEPTION_REGISTRATION_FIELD* prev=0;
```

Exception handler függvény

```
22 EXCEPTION_DISPOSITION __cdecl my_except_handler(
23     struct _EXCEPTION_RECORD *ExceptionRecord,
24     void * EstablisherFrame,
25     struct _CONTEXT *ContextRecord,
26     void * DispatcherContext )
27 {
28     unsigned i;
29
30     printf( "Megszakitaskezelo fut...\n" );
31
32     // Hiba kijavítása
33     ContextRecord->Eax = (int)&adat;
34
35     // Megszakítást okozó utasítás megismétlése
36     return ExceptionContinueExecution;
37 }
```

Exception handler regisztrálása

Illusztráció!

```
42      // Stack-be felépítjük a láncolást
43      __asm
44      {
45          sub     ESP, 8
46          mov     dword ptr pRegister,ESP
47          mov     EAX,FS:[0]
48          mov     dword ptr prev,EAX
49      }
50
51      pRegister->handler = my_except_handler;
52      pRegister->prev = prev;
53
54      // TIB-be regisztráljuk a hibakezelőnket
55      __asm
56      {
57          mov     EAX, dword ptr pRegister
58          mov     FS:[0],EAX
59      }
```

Exception kikényszerítése

Illusztráció!

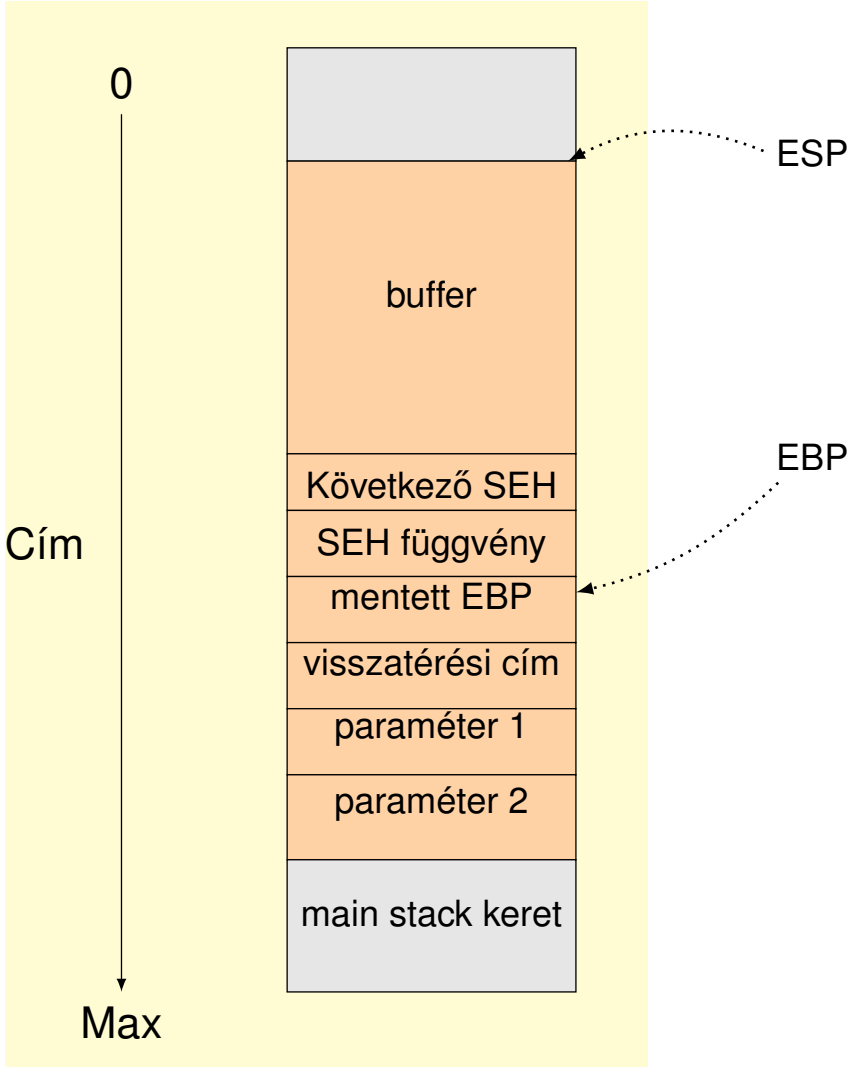
```
61      // megpróbálunk a 0-ás cimen lévő adatot
62      // felülírni
63      __asm
64      {
65          mov     EAX,0
66          mov     [EAX], 2
67      }
68
69      printf( "Hiba utan %d\n", adat );
```

Exception handler megszüntetése

Illusztráció!

```
71      __asm
72      {
73          mov     eax,[ESP]
74          mov     FS:[0], EAX
75          add     esp, 8
76      }
```

Stack keret SEH-vel



Exception handler működése és átverése

```
0013fb44: ntdll!RtlConvertUlongToLargeInteger+a0 (77d1ef10)
0013ff30: kiserlet1!ILT+5(_my_except_handler)+0 (0040100a)
0013ff70: kiserlet1!_except_handler4+0 (00402720)
```

} SEH lánc

```
@ILT+5(_my_except_handler):
0040100A: E9 11000000  jmp  my_except_handler (401020h)
0040100F CC           int  3

EXCEPTION_DISPOSITION __cdecl my_except_handler(
    struct _EXCEPTION_RECORD *ExceptionRecord,
    void * EstablisherFrame,
    struct _CONTEXT *ContextRecord,
    void * DispatcherContext )
{
00401020: 55      push   ebp   <<< Itt tartunk
00401021: 8B EC   mov    ebp,esp
00401023: 51      push   ecx
    printf( "Megszakitaskezelo fut...\n" );

0:000> d esp
0013fb30:   f2ee9977 30fc1300 30ff1300 4cfc1300  (exception
   handler paraméterek)

0:000> d 13ff30
0013ff30:  70ff1300 0a104000 80ff1300 b1134000 (SEH lánc)
```

Exception handler működése és átverése

```
0013fb44: ntdll!RtlConvertUlongToLargeInteger+a0 (77d1ef10)
0013ff30: kiserlet1!ILT+5(_my_except_handler)+0 (0040100a)
0013ff70: kiserlet1!_except_handler4+0 (00402720)

@ILT+5(_my_except_handler):
0040100A: E9 11000000  jmp  my_except_handler (401020h)
0040100F CC           int  3

EXCEPTION_DISPOSITION __cdecl my_except_handler(
    struct _EXCEPTION_RECORD *ExceptionRecord,
    void * EstablisherFrame,
    struct _CONTEXT *ContextRecord,
    void * DispatcherContext )
{
00401020: 55      push   ebp   <<< Itt tartunk
00401021: 8B EC   mov    ebp,esp
00401023: 51      push   ecx
    printf( "Megszakitaskezelo fut...\n" );

0:000> d esp
0013fb30:   f2ee9977 30fc1300 30ff1300 4cfc1300  (exception
   handler paraméterek)

0:000> d 13ff30
0013ff30:  70ff1300 0a104000 80ff1300 b1134000 (SEH lánc)
```

SEH lánc

Exception handler működése és átverése

```
0013fb44: ntdll!RtlConvertUlongToLargeInteger+a0 (77d1ef10)
0013ff30: kiserlet1!ILT+5(_my_except_handler)+0 (0040100a)
0013ff70: kiserlet1!_except_handler4+0 (00402720)

@ILT+5(_my_except_handler):
0040100A: E9 11000000  jmp  my_except_handler (401020h)
0040100F CC           int  3

EXCEPTION_DISPOSITION __cdecl my_except_handler(
    struct _EXCEPTION_RECORD *ExceptionRecord,
    void * EstablisherFrame,
    struct _CONTEXT *ContextRecord,
    void * DispatcherContext )
{
00401020: 55      push   ebp   <<< Itt tartunk
00401021: 8B EC   mov    ebp,esp
00401023: 51      push   ecx
    printf( "Megszakitaskezelo fut...\n" );

0:000> d esp
0013fb30:   f2ee9977 30fc1300 30ff1300 4cfc1300  (exception
   handler paraméterek)

0:000> d 13ff30
0013ff30:  70ff1300 0a104000 80ff1300 b1134000 (SEH lánc)
```

} SEH lánc

Exception handler működése és átverése

```
0013fb44: ntdll!RtlConvertUlongToLargeInteger+a0 (77d1ef10)
0013ff30: kiserlet1!ILT+5(_my_except_handler)+0 (0040100a)
0013ff70: kiserlet1!_except_handler4+0 (00402720)

@ILT+5(_my_except_handler):
0040100A: E9 11000000  jmp  my_except_handler (401020h)
0040100F CC           int  3

EXCEPTION_DISPOSITION __cdecl my_except_handler(
    struct _EXCEPTION_RECORD *ExceptionRecord,
    void * EstablisherFrame,
    struct _CONTEXT *ContextRecord,
    void * DispatcherContext )
{
00401020: 55      push   ebp   <<< Itt tartunk
00401021: 8B EC   mov    ebp,esp
00401023: 51      push   ecx
    printf( "Megszakitaskezelo fut...\n" );

0:000> d esp
0013fb30:   f2ee9977 30fc1300 30ff1300 4cfc1300  (exception
   handler paraméterek)

0:000> d 13ff30
0013ff30:  70ff1300 0a104000 80ff1300 b1134000 (SEH lánc)
```

SEH lánc

Exception handler működése és átverése

```
0013fb44: ntdll!RtlConvertUlongToLargeInteger+a0 (77d1ef10)
0013ff30: kiserlet1!ILT+5(_my_except_handler)+0 (0040100a)
0013ff70: kiserlet1!_except_handler4+0 (00402720)

@ILT+5(_my_except_handler):
0040100A: E9 11000000  jmp  my_except_handler (401020h)
0040100F CC            int  3

EXCEPTION_DISPOSITION __cdecl my_except_handler(
    struct _EXCEPTION_RECORD *ExceptionRecord,
    void * EstablisherFrame,
    struct _CONTEXT *ContextRecord,
    void * DispatcherContext )
{
00401020: 55      push   ebp   <<< Itt tartunk
00401021: 8B EC   mov    ebp,esp
00401023: 51      push   ecx
    printf( "Megszakitaskezelo fut...\n" );

0:000> d esp
0013fb30:   f2ee9977 30fc1300 30ff1300 4cfc1300  (exception
   handler paraméterek)

0:000> d 13ff30
0013ff30:  70ff1300 0a104000 80ff1300 b1134000 (SEH lánc)
```

SEH lánc

Exception handler működése és átverése

```
0013fb44: ntdll!RtlConvertUlongToLargeInteger+a0 (77d1ef10)
0013ff30: kiserlet1!ILT+5(_my_except_handler)+0 (0040100a)
0013ff70: kiserlet1!_except_handler4+0 (00402720)

@ILT+5(_my_except_handler):
0040100A: E9 11000000  jmp  my_except_handler (401020h)
0040100F CC           int  3

EXCEPTION_DISPOSITION __cdecl my_except_handler(
    struct _EXCEPTION_RECORD *ExceptionRecord,
    void * EstablisherFrame,
    struct _CONTEXT *ContextRecord,
    void * DispatcherContext )
{
00401020: 55      push   ebp   <<< Itt tartunk
00401021: 8B EC   mov    ebp,esp
00401023: 51      push   ecx
    printf( "Megszakitaskezelo fut...\n" );

0:000> d esp
0013fb30:   f2ee9977 30fc1300 30ff1300 4cfc1300  (exception
   handler paraméterek)

0:000> d 13ff30
0013ff30:  EB069090 10011339 80ff1300 b1134000 (SEH lánc)
```

SEH lánc

```
pop edi
pop esi
ret
```

jmp $+8; **nop**; **nop**;

SEH exploit

1. Előző kivételt adminisztráló blokkra mutató pointer struct EXCEPTION_REGISTRATION_FIELD* prev*;* átírása ugró utasítással **jmp** $+8*;* **nop***;* **nop***;* !
2. Kivételkezelő függvénypointer átírása egy olyan címmel, ahol **pop** reg3*;* **pop** reg3*;* **ret** *;* vagy ezzel ekvivalens utasítás van. (Például **add esp**,8*;ret*.)
3. Kivételdobás kikényszerítése. Általában ez automatikusan megtörténik, ha elrontjuk a stack tartalmát.

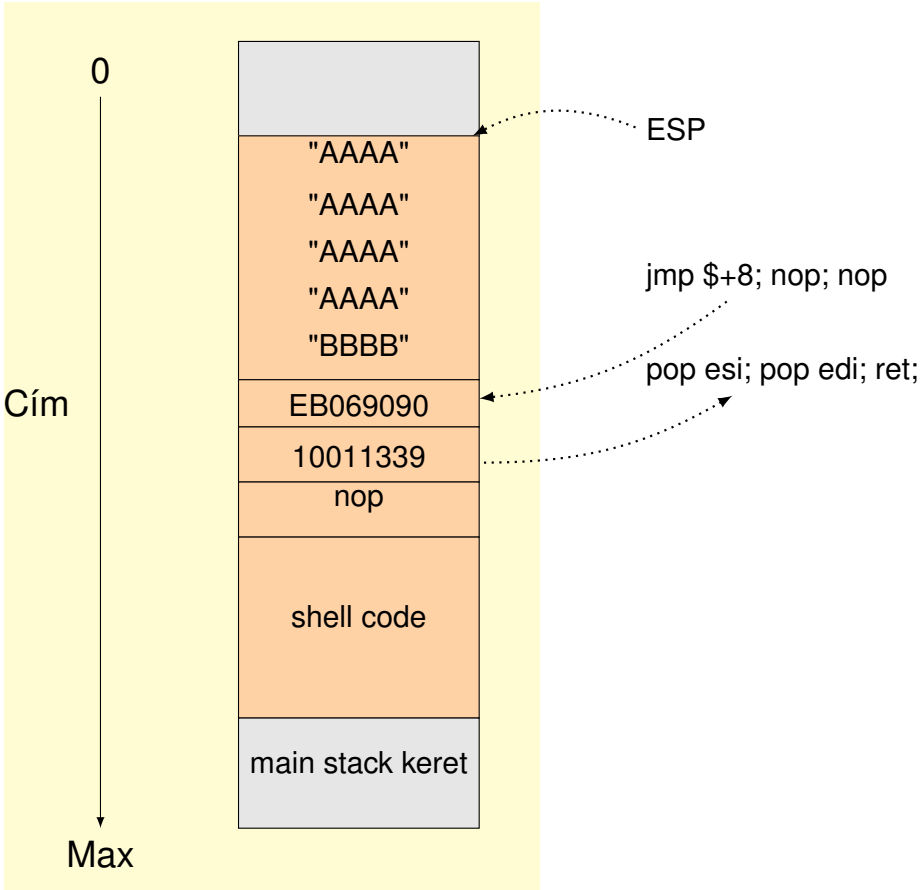
Exception kikényszerítése

```
21 int authenticate(char *username, char *password) {
22   int authenticated;
23   char buffer[1024];
24   __try
25   {
26
27     authenticated = verify_password(username, password);
28     if(authenticated == 0)    {
29       sprintf(buffer,
30         "password is incorrect for user %s\n",
31         username);
32       log("%s", buffer);
33     }
34   }
```

A puffer túlcsordulás miatt az authenticate függvény mielőtt visszatérne az sprintf -be kivételt dob!

SEH shellcode

```
a.exe 1001 A 4 B
    -a 909006eb
    -a 10011339
    1 nop
    -f shellcode.bin
    10 X
    |
    sehauthenticate.exe
    ↪
```



C++ kivételkezelés sebezhetősége

Figyelem

Windows alatt a C++ kivételkezelése nem szükségszerűen, de a Visual Studio esetén biztosan a SEH-re épül.
Blokk strukturált nyelveknél a kivételkezelés az objektumok destruktorainak a meghívása miatt bonyolult.
Ugyanakkor a fent leírt elvek alapján puffer-túlcsordulás esetén a C++ program támadható.

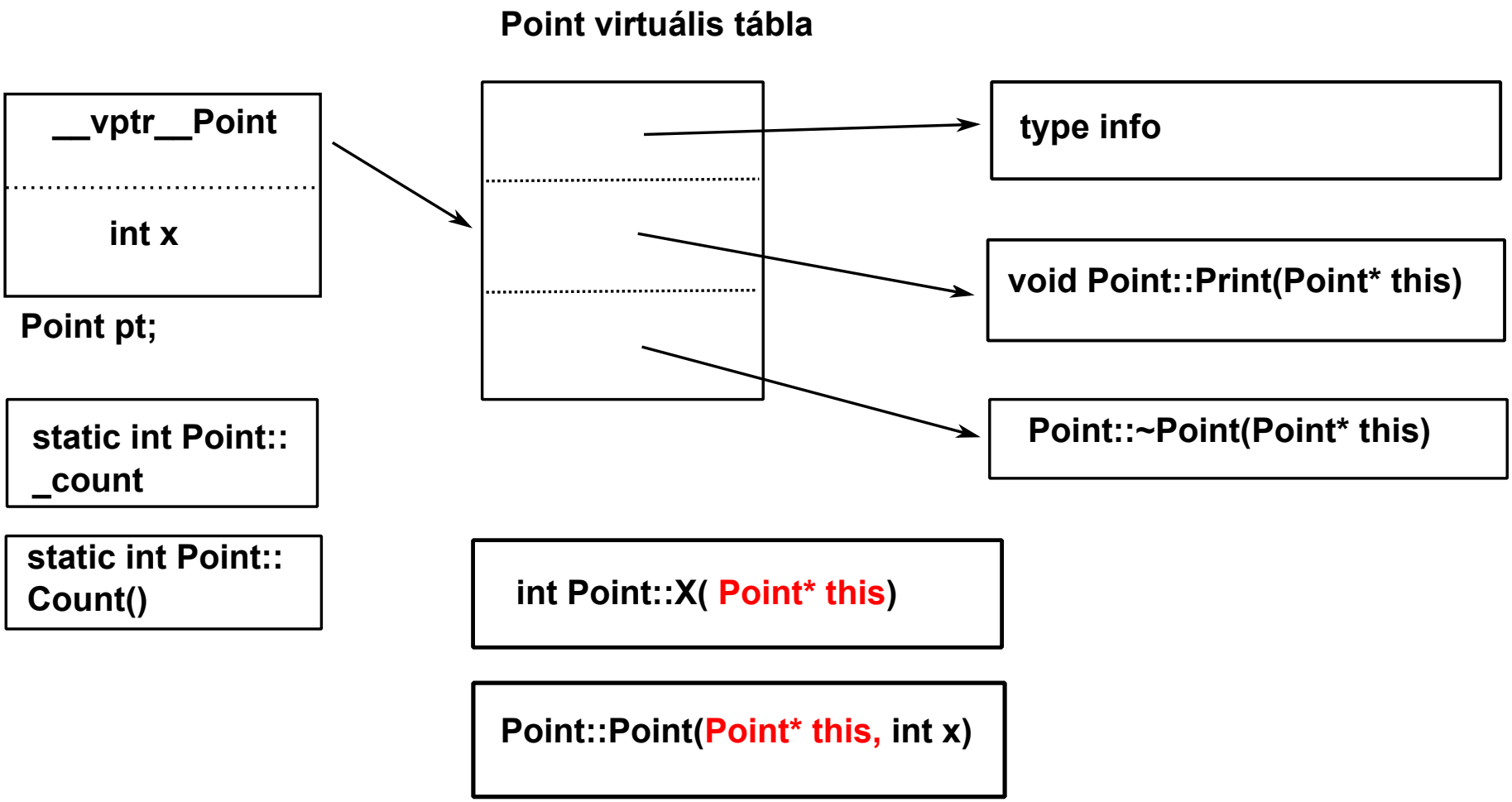
C++ objektum modell

Illusztráció!

```
class Point {
public:
    Point( int x );
    virtual ~Point();
    int X() const;
    static int Count() ;
    virtual void Print() const;

protected:
    int x;
    static int _count;
};

Point::Point( int x ) : x(x)  {  _count++; }
Point::~Point() { _count--; }
int Point::X() const { return x; }
int Point::Count() { return _count; }
void Point::Print() const { printf("%d ", x); }

int Point::_count = 0;

int main() {
    int n;
    Point p1(4);
    n = Point::Count();   // stat fv. hívás demo
    return 0;
}
```

Lehetséges C++ objektum modell

Illusztráció!



Támadás megvalósítása I

Előfeltétel:

- A virtuális függvényt tartalmazó objektum a stack-en legyen.
- Stringgel tudjuk a memóriát felülírni (csak a string vége tartalmaz nullát).

Támadás megvalósítása:

- A shellcode-ot a puffer elejére helyezzük.
- Írjuk felül a virtuális függvénytáblára mutató pointert is. A stack címe Windows alatt 0-val végződik, ezért a stringet lezáró 0 pont a virtuális függvény táblára mutató pointer utolsó byte legyen (Little-Endian)[1].

Támadás megvalósítása II

- A pufferbe hozzunk létre egy ál virtuális függvénytáblát. Hogyan tudjuk a shellcode-ot megcímezni, ha 0-val kezdődő értéket nem tudunk az ál virtuális táblába bevinni?
- Nézzük meg mely regiszterek mutatnak a stack-re. A **jmp esp** megoldáshoz hasonlóan keressünk megfelelő utasítás sorozatot a betöltött dll-ekben, és ezzel a címmel töltsük fel a virtuális függvénytáblát. (Csak az a cím jó amelyben nincs 0 byte).
- Várjuk, amig a program meg nem hívja a virtuális függvényt. Származtatás esetén a destruktornak illik virtuálisnak lenni.

[1] Linux vagy bináris adatokat tartalmazó puffer esetén ezzel nem kell törődni.

Stack süti megkerülése

- Futtassuk le a shellcode-unkat mielőtt a rendszer ellenőrizné a kanári értéket. Ez SEH exploit estén teljesül. Ez ellen a kanári nem véd. Másik lehetőség a virtuális függvénytábla átírása.
- Mi van, ha a függvény nem használ kivételkezelő rutint, és nincs benne virtuális függvényt tartalmazó objektum sem. Nagy valószínűséggel a megelőző stack keretekben van kivételt regisztráló bejegyzés, amit át tudunk írni. Lásd David Litchfield: Defeating the Stack Based Buffer Overflow Prevention Mechanism of Microsoft Windows 2003 Server cikkét.

Safe SEH

A stack sütit legegyszerűbben SEH exploit segítségével kerülhetjük meg. Akadályozzuk meg, hogy kivételdobás esetén bármely címen lévő utasítást meg lehessen hívni.

A cél, hogy csak a szabályosan regisztrált kivételkezelőket lehessen meghívni.

Visual Studioban a linkernek a /SAFESEH:yes paramétert kell megadni. Ez az alapértelmezés. Letiltása /SAFESEH:no

Safe SEH átverése I

- Ha az alkalmazás és néhány dll /SAFESEH:no-val volt linkelve, akkor az előzőekben megismert **pop**, **pop**, **ret**-es technika működik.
- `https://www.google.com/patents/US7480919` Safe exceptions szabadalom tartalmazza a biztonságos strukturált kivételkezelés működését. A szabadalom gondoskodik olyan object kódok együtt működéséről is, amelyek egy részét még még nem a Safe SEH figyelembevételével fordították. A .sxdata szegmens tárolja a regisztrált kivételkezelő függvények címét. Ha a kód /SAFESEH:no-val készült, akkor nincs ilyen szegmens. Ezért futásnál az *exception dispatcher* annyit tud megnézni, hogy az kivételkezelő függvény címe ne legyen az aktuális stack keretben. Az érvényes stack intervallumot az TEB-ben az FS:[4] és

Safe SEH átverése II

FS[8] érték tartalmazzák. Ezen kívül eső címen lévő kód hívható.

- Hasonló okok miatt a nem module címterébe eső kód is hívható. Minden adat, ami gépi utasításként is értelmezhető az egyben kód is. A heap-en lévő mp3 fájl tartalmára is rá lehet adni a vezérlést. A heap-pel az a baj, hogy a megfelelő adat címét nem mindig tudjuk előre.
- A Load Configuration Directory-t nem tartalmazó kódszegmens utasítása is hívható.

Data Execution Prevention (DEP) I

Korlátozzuk le a stack-et úgy, hogy ott ne lehessen kódot futtatni. Ehhez hardware támogatásra van szükség. Szegmens védelemmel ez megoldható, de akkor sérül a *flat memória modell*. Lap védelemmel lehet megoldani, de ez csak a későbbi Intel processzorokban lett megvalósítva (manapság már alig lehet találni olyan PC-t amelyik nincs erre felkészítve).

Első DEP-es Windows rendszerek

- Windows XP Service Pack 2;
- Windows XP Tablet PC Edition 2005;
- Windows Server 2003 Service Pack 1;
- Újabbak értelemszerűen tudják.

Data Execution Prevention (DEP) II

Üzemmódok:

- **Optin** futtatni lehet a nem DEP kompatibilis programokat, de egy listán meg lehet adni a melyek DEP kompatibilis módba fussanak. Önkicsomagoló exe program például futtat kódot a stacken.
- **OptOut** Általában a windows szerverekben az az alapértelmezés, hogy minden program DEP kompatibilis. Amelyek nem, azokat kell kivétel listára tenni.

Boot beállítások:

- **AlwaysOn** minden processz DEP védett, nincs kivétel.
- **AlwaysOff** egyik processz sem DEP védett.