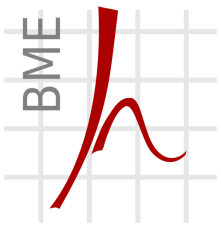
Biztonságos programozás – Puffer túlcsordulásos támadások



Híradástechnikai Tanszék

Izsó Tamás

2015. október 12.

Section 1

DEP támadás

ret2libc

A különböző DEP beállítások miatt több függvény is van, amellyel a DEP védelmet ki lehet kapcsolni[1].

```
ULONG ExecuteFlags = MEM_EXECUTE_OPTION_ENABLE; // 2

NtSetInformationProcess(
    NtCurrentProcess(),      // (HANDLE)-1
    ProcessExecuteFlags,     // 0x22
    &ExecuteFlags,           // ptr to 0x2
    sizeof(ExecuteFlags));   // 0x4
```

[1] Windows 8-ban ezekkel a függvényekkel csak bekacsolni lehet, kikapcsolni nem!

Védelem kikapcsolása

Terv:

- A visszatérési címet írjuk felül az NtSetInformationProcess függvény címével.
- A következő stack címre tegyük a (HANDLE)-1 értéket.
- A következő stack címre tegyünk 0x22 értéket.
- A következő stack címre egy 32 bites 2 értéket tartalmazó adatterület címét tegyük.
- Ezután következzen a 4-es.
- A következő adat egy cím, amely egy DLL-ben lévő **jmp esp***;* utasításra mutat.

Ha ez működik, akkor ki tudjuk kapcsolni a védelmet, és le tudjuk futtatni a programunkat.

ret2libc programozás I

A módszer lényege, hogy nem a kódot, hanem függvény címeket és a paramétereket tesszük a stack-re.

- A puffer túlcsordulást tartalmazó függvény visszatérésénél a **ret** utasítás hatására az oda készített címre kerül a vezérlés, miközben az **esp** értéke eggyel csökken.
- Ha a meghívott rutinnak paraméterekre van szüksége, akkor a puffer felülírás során gondoskodhatunk ezek megadásáról.
- A Windows API függvények a visszatérés előtt felszabadítják a stack-et (stdcall konvenció). Így az **esp** regiszter a gondosan előkészített, következő meghívandó függvényre fog mutatni. A **ret** hatására a következő könyvtári függvény hívódik meg. Vegyük észre, hogy sohasem fut le **call** utasítás, ami letenné a visszatérési címet.

ret2libc programozás II

- Sok esetben nem célszerű a függvény első utasítását meghívni, hanem beleugorhatunk a függvény közepébe. Nem szabad elfeledkezi arról, hogy ilyenkor is a függvény felszabadítja a stack-en használt lokális változókat. Ezért a következő meghívandó függvény címét ennek megfelelően a stacken távolabb kell elhelyezni.

ret2libc módszer korlátai

- Csak olyan standard C vagy operációs rendszer függvényeket lehet meghívni, amelyek:
  - statikusan hozzá vannak linkelve a programkódhoz, vagy
  - betöltött (használt) dll-ben lévő függvények.
- Nem minden cím vagy paraméter érték vihető be a pufferbe, például string puffer esetén 0 értékű byte nem szerepelhet a címekben vagy paraméterek értékében[2].

[2] Ezen még dolgozunk!

ret2libc programozás (esettanulmány) I

A NtSetInformationProcess függvény paraméterei 0-ás értékű byte-okat tartalmaznak. Az alábbi példa Windows XP SP3 rendszeren készült. A cím és a kód változhat. A módszer és nem az értékek fontosak.

A ntdll.dll-ben a LdrpCheckNXCompatibility hívja az NtSetInformationProcess rendszer függvényt.

```
0:000> u 7c936861
ntdll!LdrpCheckNXCompatibility+0x4d:
7c936861    push    4
7c936863    lea     eax,[ebp-4]
7c936866    push    eax
7c936867    push    22h
7c936869    push    0FFFFFFFFh
7c93686b    call    ntdll!ZwSetInformationProcess (7c90dc9e)
7c936870    jmp     ntdll!LdrpCheckNXCompatibility+0x5c (7c91cd8d)
7c936875    nop
```

ret2libc programozás (esettanulmány) II

Tehát az [**ebp**−4] címen van a 2-es érték. A kód, ahonnan a 0x7c936861 címre ugrottunk.

```
7c91cd4f   cmp    dword ptr [ebp-4],0
7c91cd53   jne    ntdll!LdrpCheckNXCompatibility+0x4d (7c936861)
```

Ha az [**ebp**−4] területre be tudjuk írni a 2-es értéket, akkor az ugrás meg fog történni. Meg kell nézni, hogy hogyan tudunk erre a pontra jutni.

```
7c94153e   mov    dword ptr [ebp-4],esi
7c941541   jmp    ntdll!LdrpCheckNXCompatibility+0x1d (7c91cd4f)
```

Ahhoz, hogy az [**ebp**−4] területen 2-es legyen, az **esi** regiszterbe 2-es értéket kell tölteni.

ret2libc programozás (esettanulmány) III

```
0:000> uf ntdll!LdrpCheckNXCompatibility
ntdll!LdrpCheckNXCompatibility:
7c91cd31   mov    edi,edi
7c91cd33   push   ebp
7c91cd34   mov    ebp,esp
7c91cd36   push   ecx
7c91cd37   and    dword ptr [ebp-4],0
7c91cd3b   push   esi
7c91cd3c   push   dword ptr [ebp+8]
7c91cd3f   call   ntdll!LdrpCheckSafeDiscDll (7c91cccb)
7c91cd44   cmp    al,1
7c91cd46   push   2
7c91cd48   pop    esi
7c91cd49   je     ntdll!LdrpCheckNXCompatibility+0x1a (7c94153e)
```

A 7c91cd46 címen lévő utasítás **push** 2; **pop esi** pont jól állítja be a leendő paraméter értékét.

Azt szeretnénk, ha a 7c91cd49 címen lévő

je ntdll !LdrpCheckNXCompatibility+0x1a ugrás teljesüljön, de

ret2libc programozás (esettanulmány) IV

ez akkor fog megtörténni, ha az **al** regiszter 1-et tartalmaz. Viszont azt a 7c91cd3f címen lévő **call** utasítás állítja be.

Teendők:

- Tegyünk 1-et az **al** regiszterbe.
- A függvényt ne az elejétől, hanem a 7c91cd44 címtől hívjuk meg.

Követett módszer

Ha a programanalízisben arra vagyunk kíváncsiak, hogy egy változó (regiszter) tartalma mely utasításoktól függ, akkor pont ezt a módszert követjük. A módszer neve visszafele haladó program szeletelés (Backward Static Program Slicing).

ret2libc programozás (esettanulmány) V

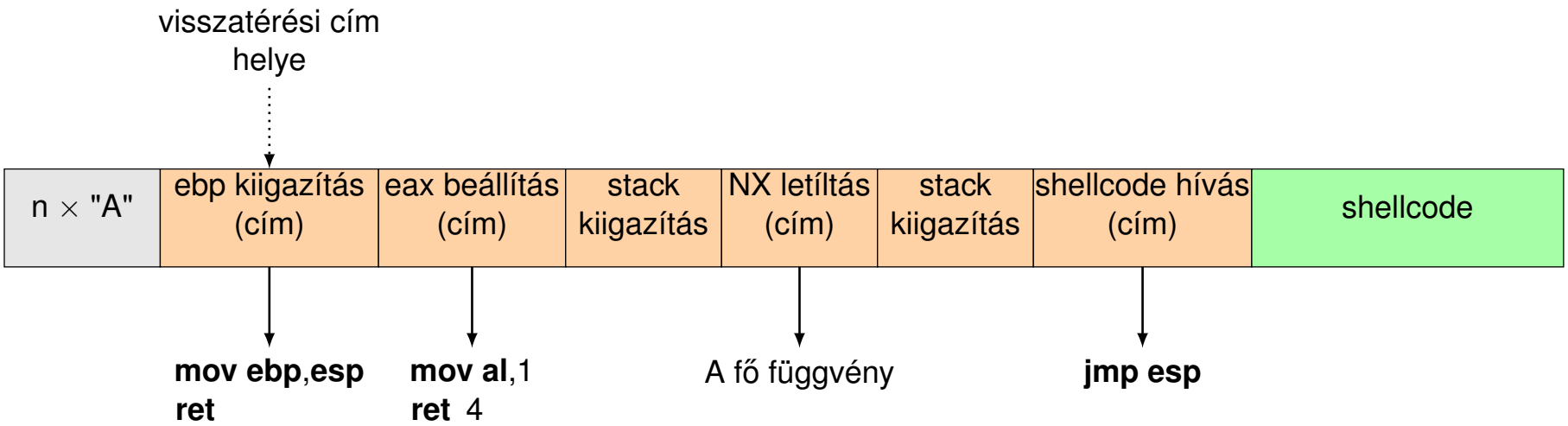
Már csak egy teendő van hátra, az al regiszterbe 0x01-et kell tenni. Hogyan lehet ezt megoldani?

Már kerestünk dll-ekben **pop esi**, **pop edi**, **ret** kódot. Akkor ehhez hasonlóan keressünk:

- **mov eax**,1; **ret** ; vagy
- **mov al**,1; **ret** ; vagy
- **xor eax**,**eax**; **inc eax**; **ret** ; vagy
- ehhez hasonlóan, a célnak megfelelő kódot.

ret2libc programozás (esettanulmány) VI

A stack felépítése:



Lehetőségek a DEP kiiktatására I

1. Kapcsoljuk ki a védelmet. Ígéretes Windows rendszer függvények (paraméterek nincsenek feltüntetve):
   - SetProcessDEPPolicy(),
   - NtSetInformationProcess().

   Ezek az Windows XP-nél működtek, újabb Windows rendszereknél már nem (csak nagyon speciális esetben) működnek. Szigorították a DEP stratégia védelmét.

Lehetőségek a DEP kiiktatására II

2. Hozzunk létre olyan memóriaterületet amely írható és futtatható memória védelemmel rendelkezik. Másoljuk ide a shellcode-unkat és adjuk rá a vezérlést. A cél eléréséhez a következő Windows rendszer függvényeket használhatjuk (paraméterek nincsenek feltüntetve):
   - VirtualAlloc() az operációs rendszer a processz számára memóriát biztosít.
   - HeapCreate() + a HeapAlloc(), hasonló mint a VirtualAlloc csak private heap-et hoz létre.
   - WriteProcessMemory() függvéy a shellcode-ot át tudja másolni olyan helyre, ahol azt le lehet futtatni.

DEP védelem kihívásai

Kérdések:

- Hogyan hívjunk meg egymás után függvényeket?
- Hogyan állítsuk elő a függvények paramétereit, ha az értékük futási időben derül ki, például függenek a memóriacímektől?
- Hogyan írjunk SEH exploitot, ahol a **jmp** $+8; **nop**; **nop** a stack-en helyezkedi el, de ott nem lehet programot futtatni?
- Hogyan vigyünk be nem megengedett értékeket (string puffer esetén 0-át).

A válasz, hogy saját kódot kell írni. De hogyan, ha a stack-en lévő kód nem futtatható.

Megoldás

ROP Return Oriented Programing

Bigyók

Szlovénia, ismeretlen alkotó műve

Return Oriented Programing

Definíció

Gadget olyan gépi utasítássorozat, amely **ret** utasítással végződik.

- Keressünk a processz címterében gadget-eket. Erre kész alkalmazások vannak.
- A stack tartalmát állítsuk össze úgy, hogy a gadget címeket, és a gadget által **pop**-pal felvett értékeket tartalmazza.

Return Oriented programozás gyakorlása I

Illusztráció!

Állítsuk elő a 0x00410A00 értéket (0 bytet-ot tartalmazó címet)

	Stack címe	stack értéke	stack tartalma
ESP ide mutat	00201A10	1003564A	cím, ahol a **pop eax**; **ret** utasítások vannak.
	00201A14	BEFFC8BF	adat, amit a **pop eax** utasítás olvas ki.
	00201A18	1A56201C	cím, ahol a **pop edi**; **ret** utasítások vannak.
	00201A1C	41414141	adat, amit a **pop edi** utasítás olvas ki.
	00201A20	2AE1231B	cím, ahol a **add eax,edi**; **pop edx**; **ret** utasítások vannak
	00201A24	DEADBEEF	adat, töltelék érték

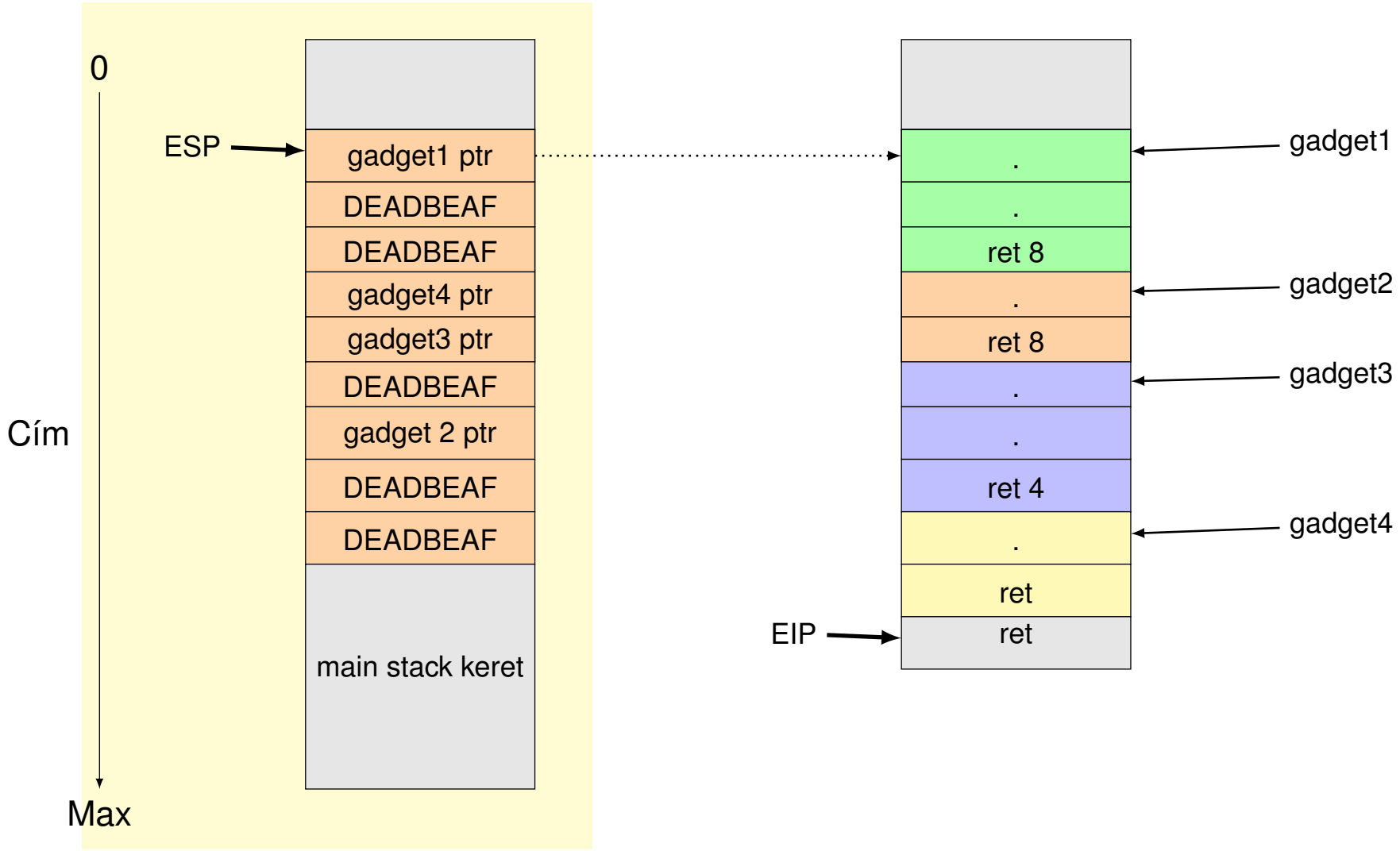
Return Oriented programozás gyakorlása II

A támadó nem ijed meg, ha ROP írása közben nem talál pont megfelelő *gadget*-et. Ha az értékes művelet és a visszatérés között van pár felesleges utasítás, ami nem csinál bajt, akkor azt is felhasználja.

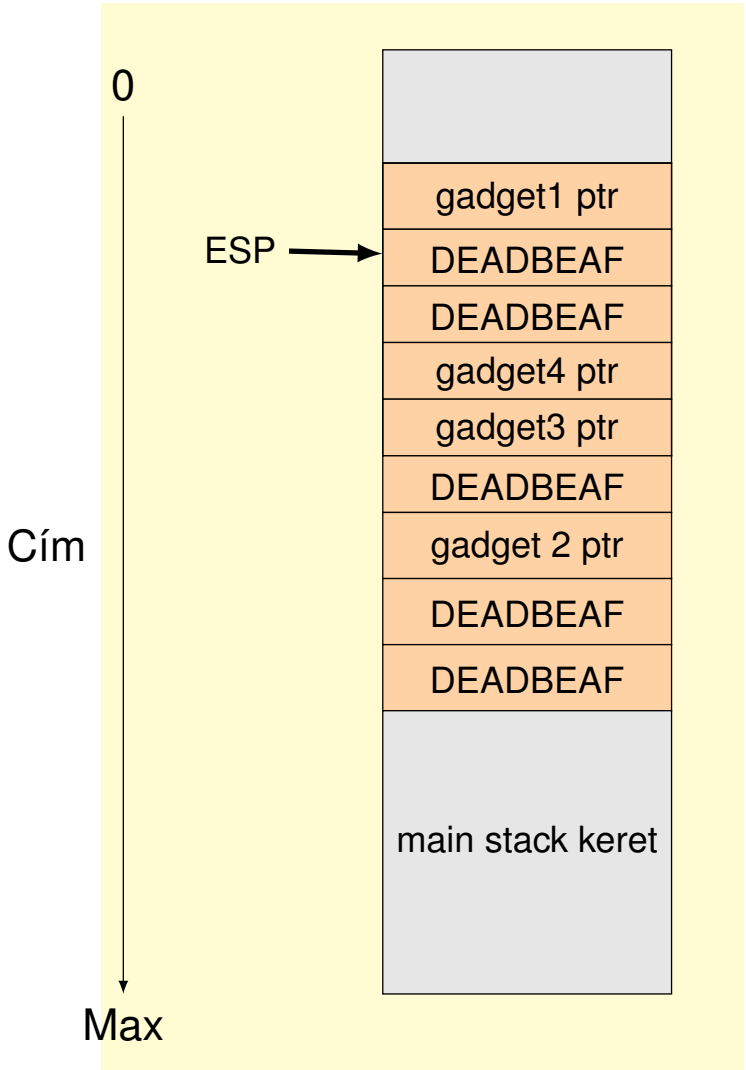
Megjegyzés

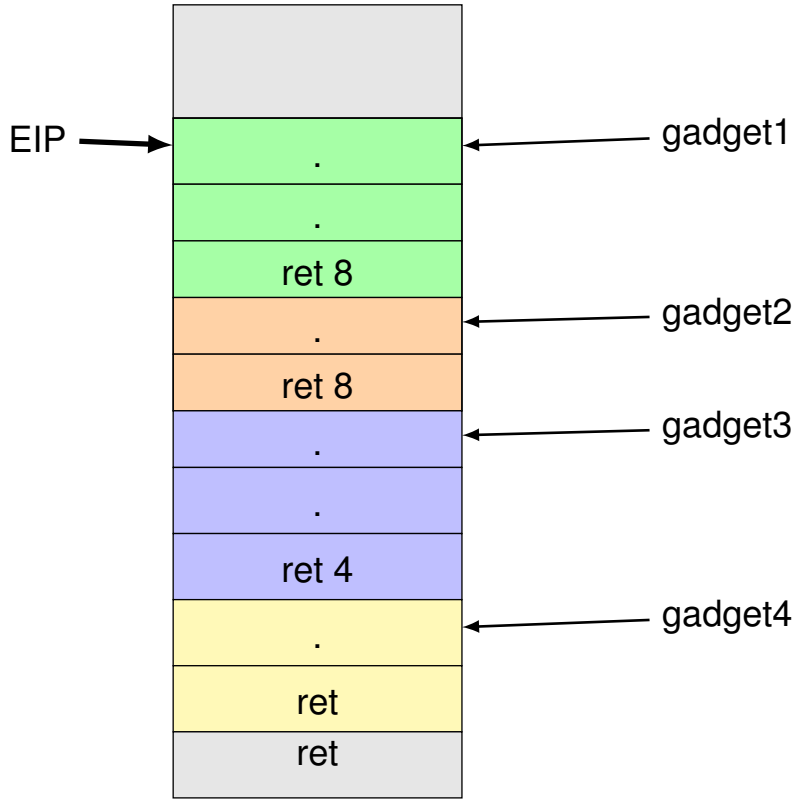
A program visszafejtésének megnehezítésére nagy számú felesleges utasításokat illesztenek a kódba. A felesleges utasítások a program szemantikáját nem változtatják meg. Ezek kiszűrésére fejlesztették ki a program szeletelés módszerét. ROP írásnál a kód *összezavarása* nem cél, de felesleges utasítások bevételével hamarabb célba érhetünk.

ROP kód működése

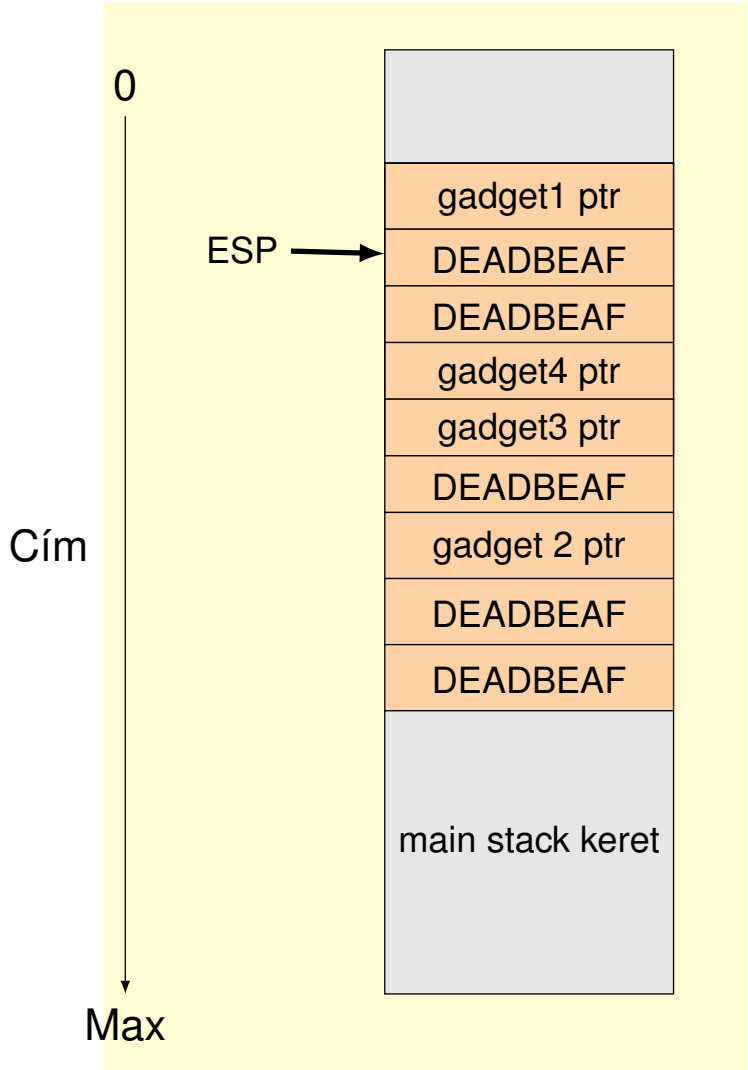


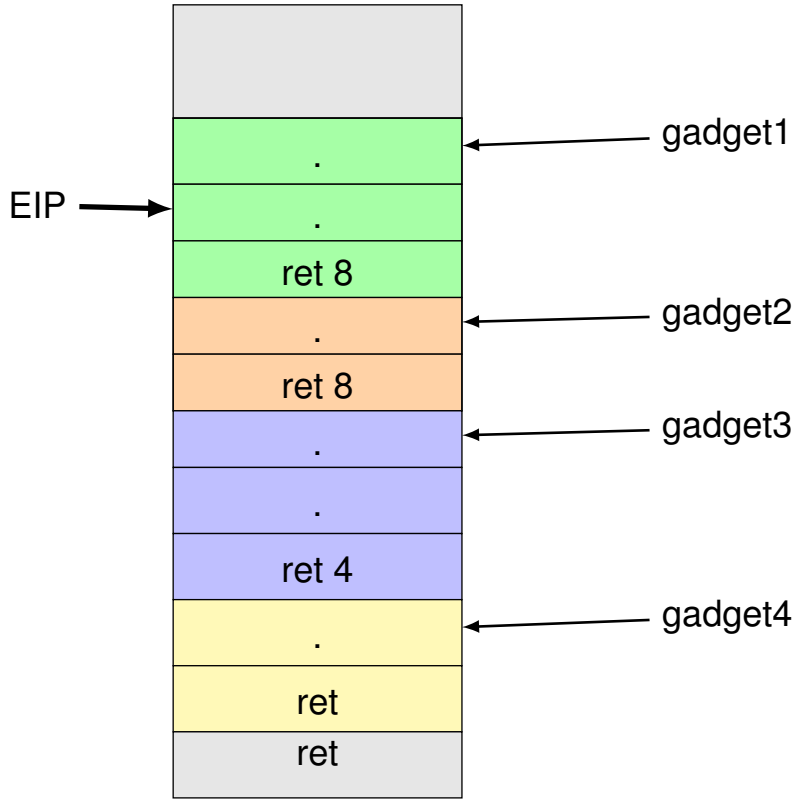
ROP kód működése



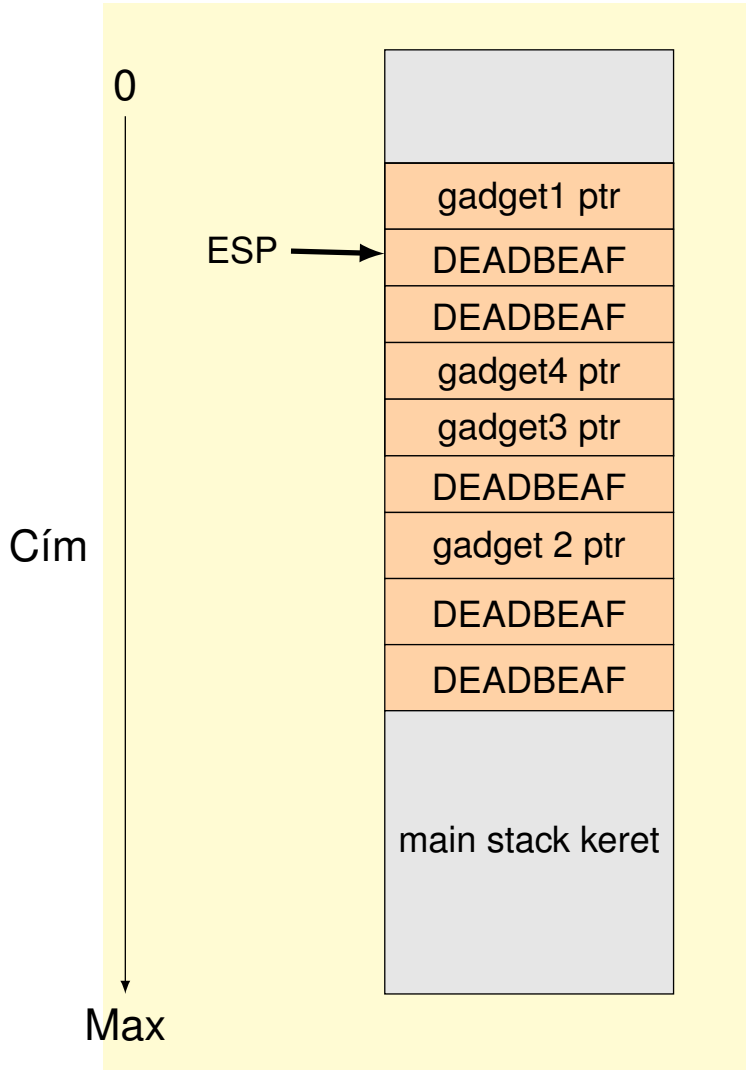


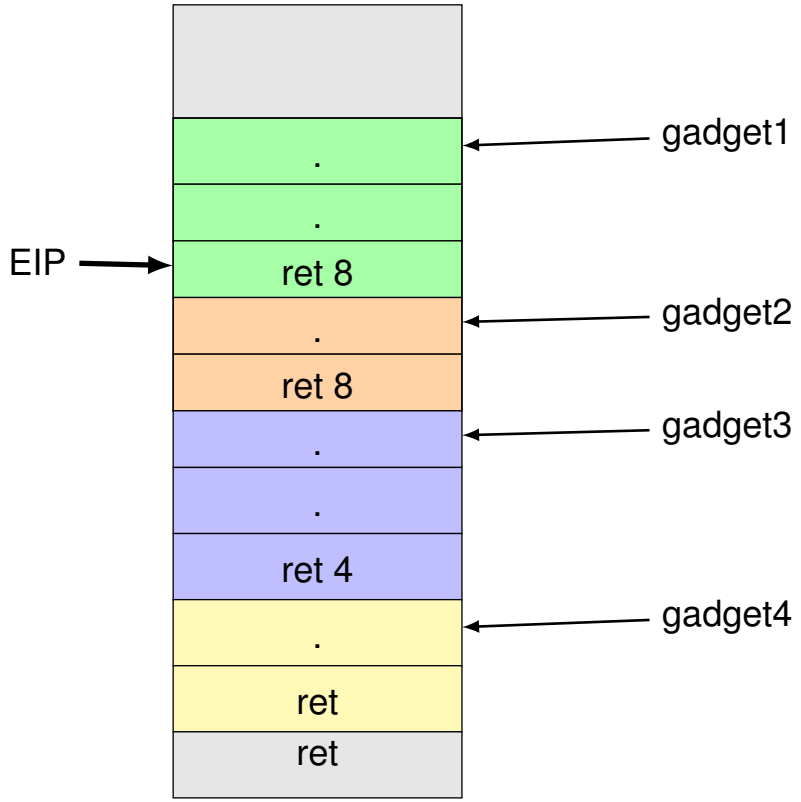
ROP kód működése



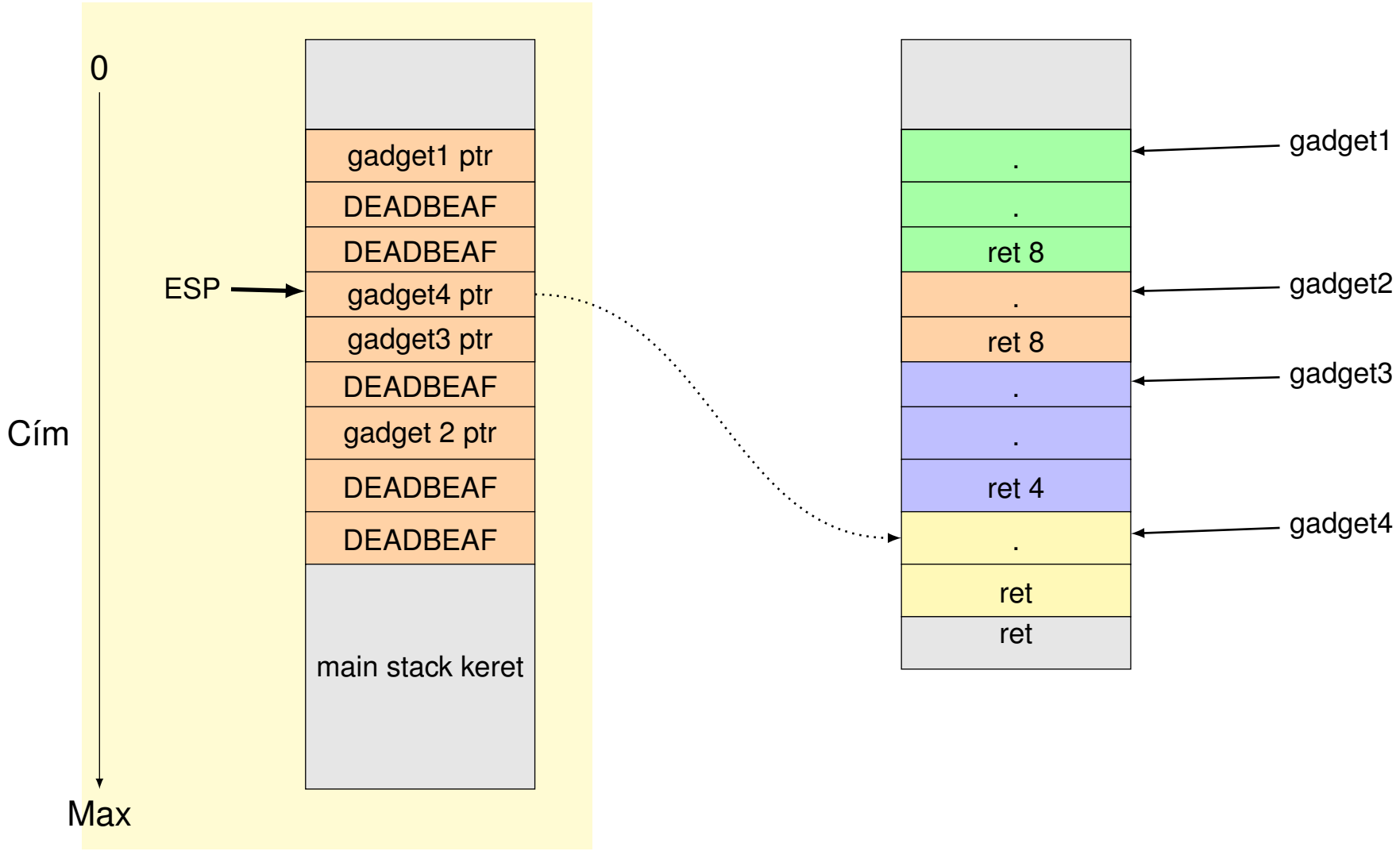


ROP kód működése

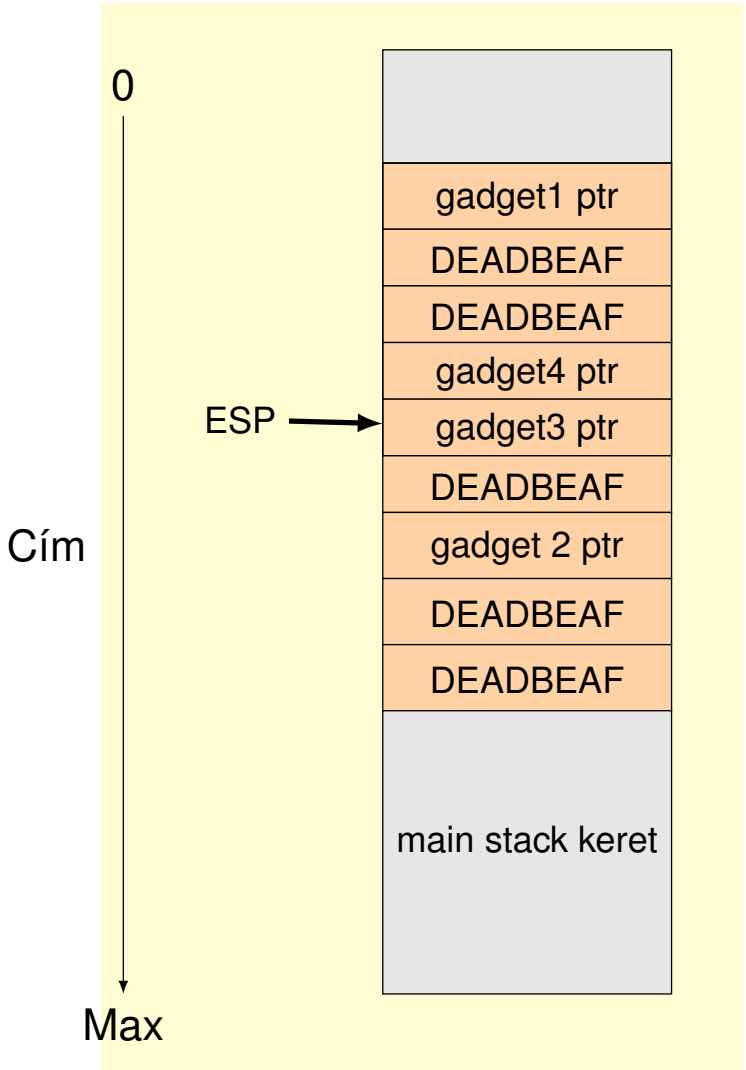


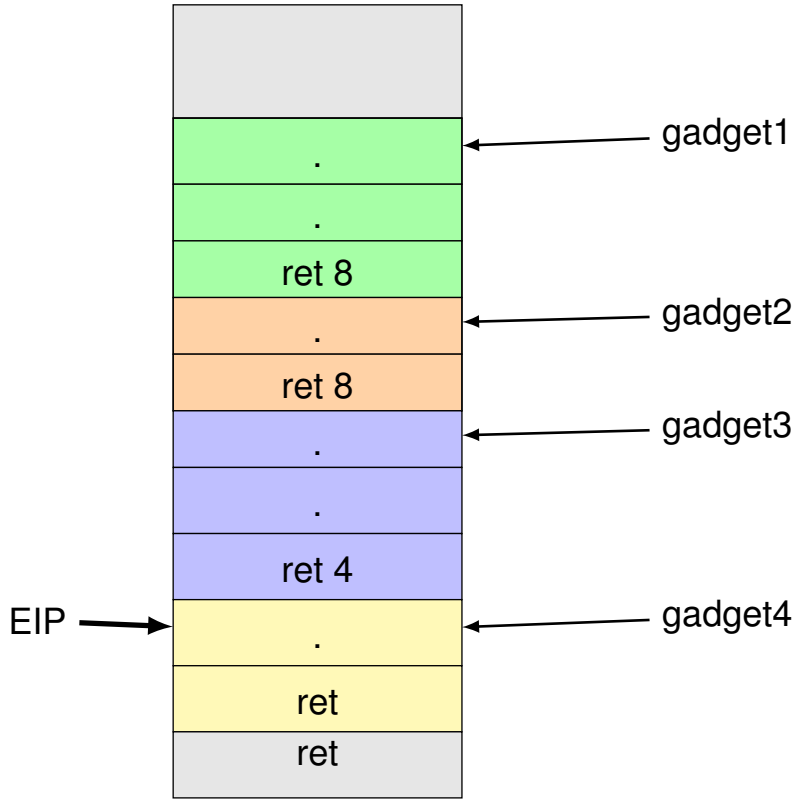


ROP kód működése

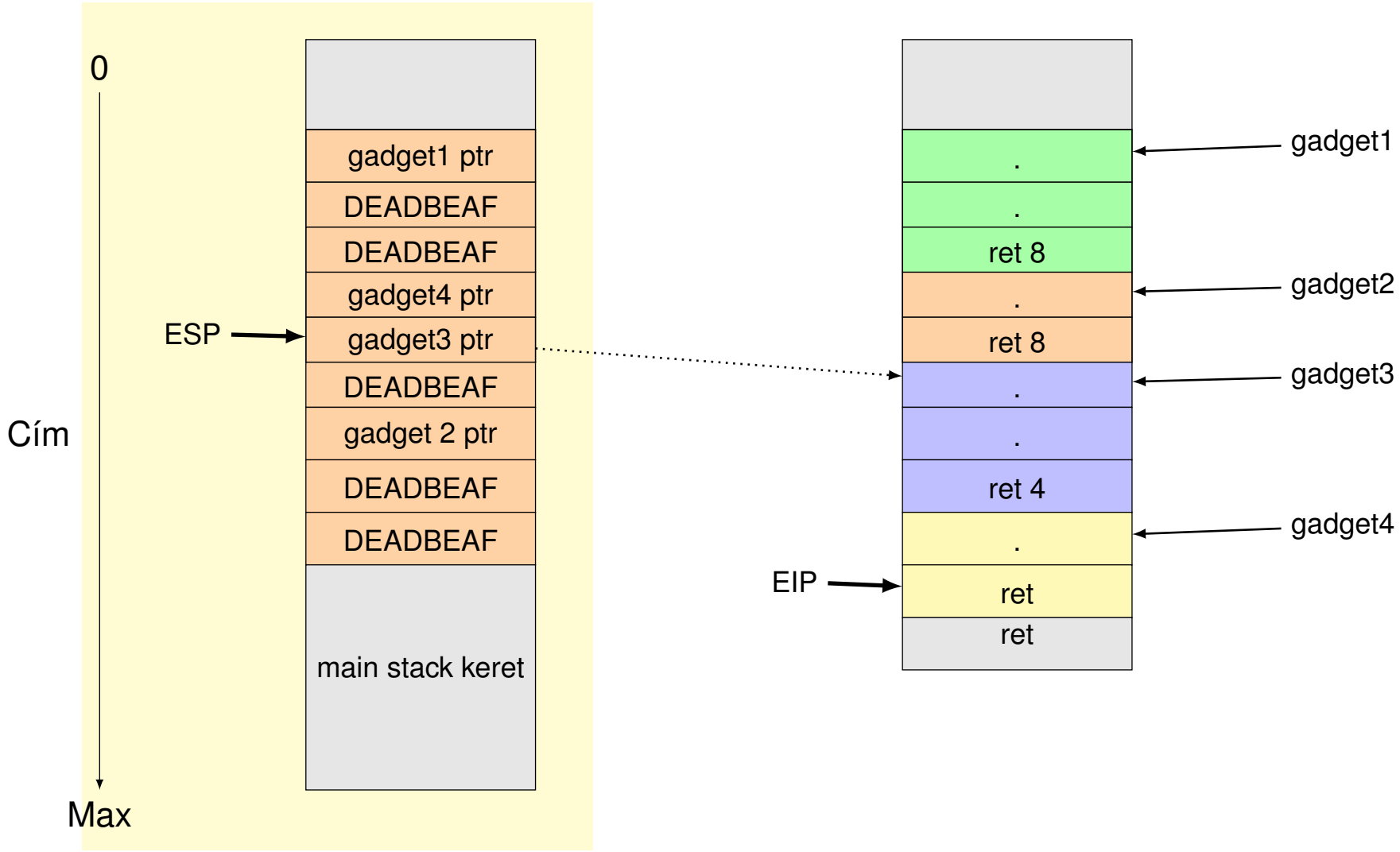


ROP kód működése

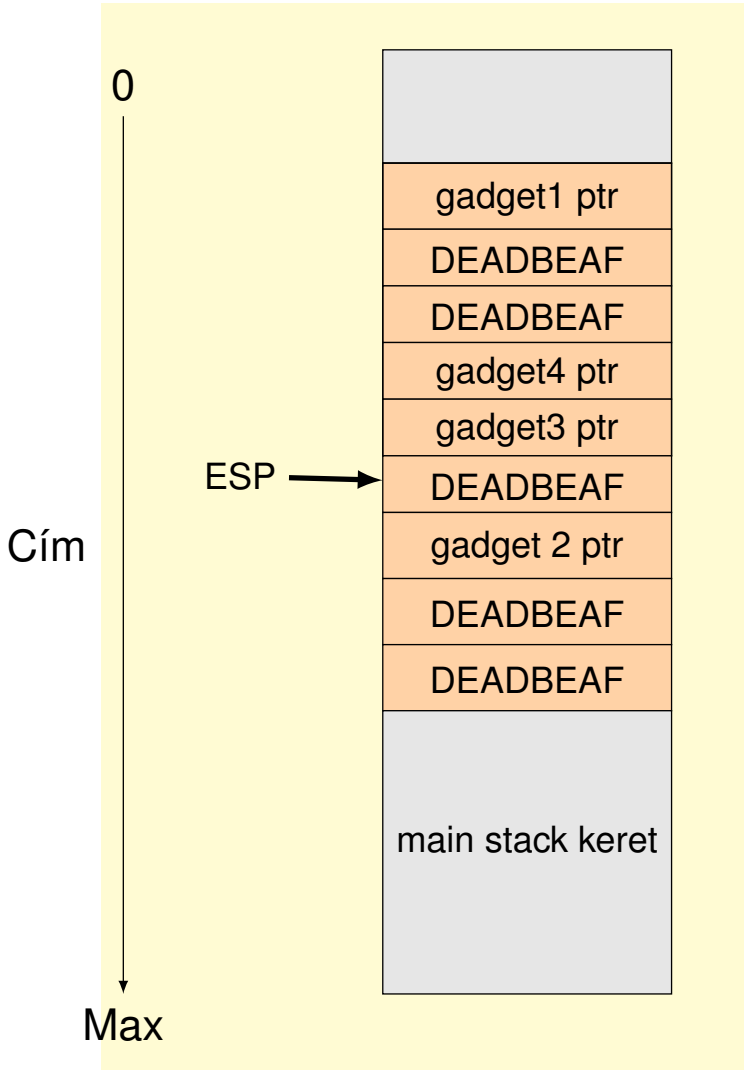


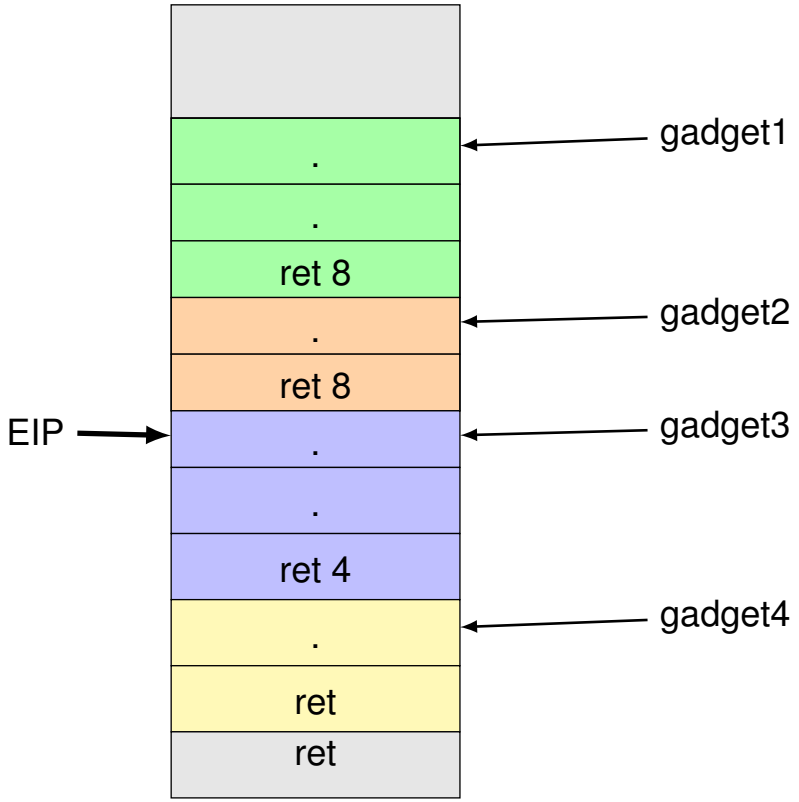


ROP kód működése

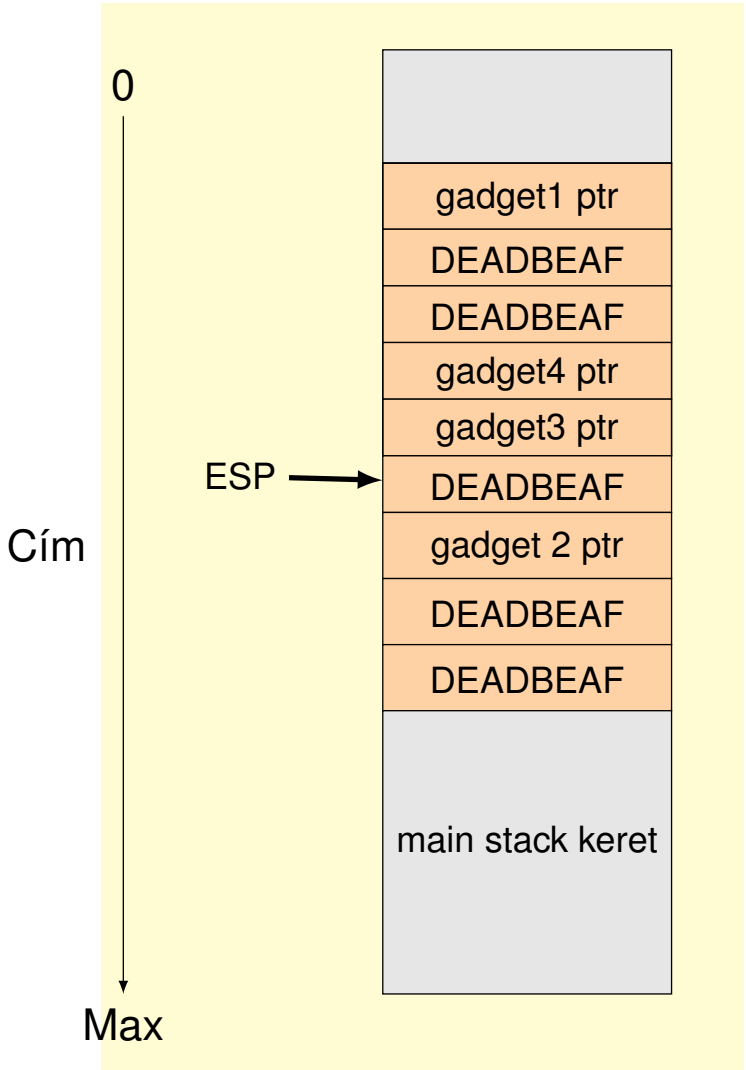


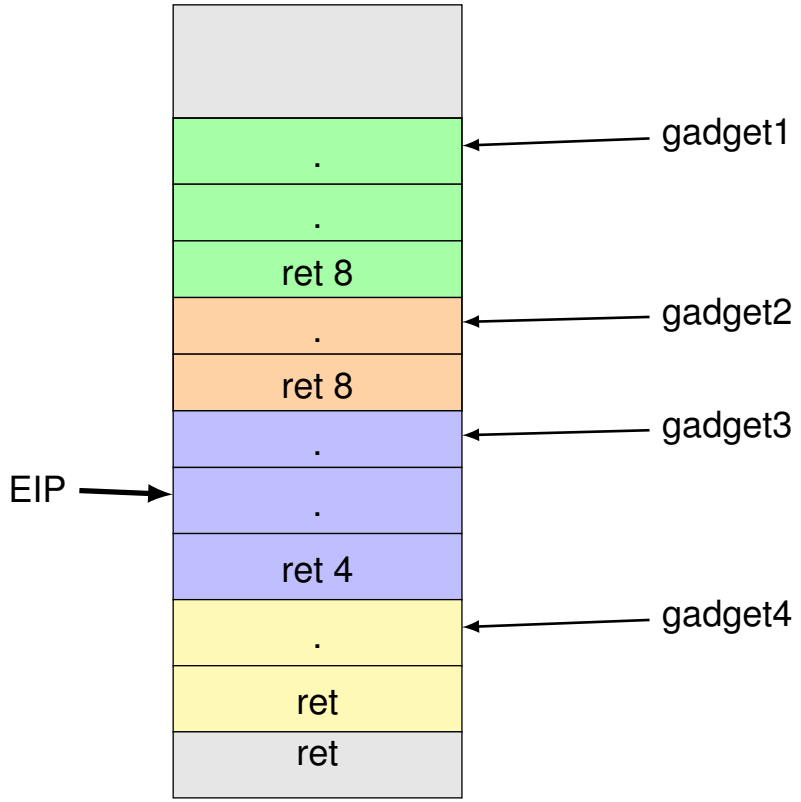
ROP kód működése



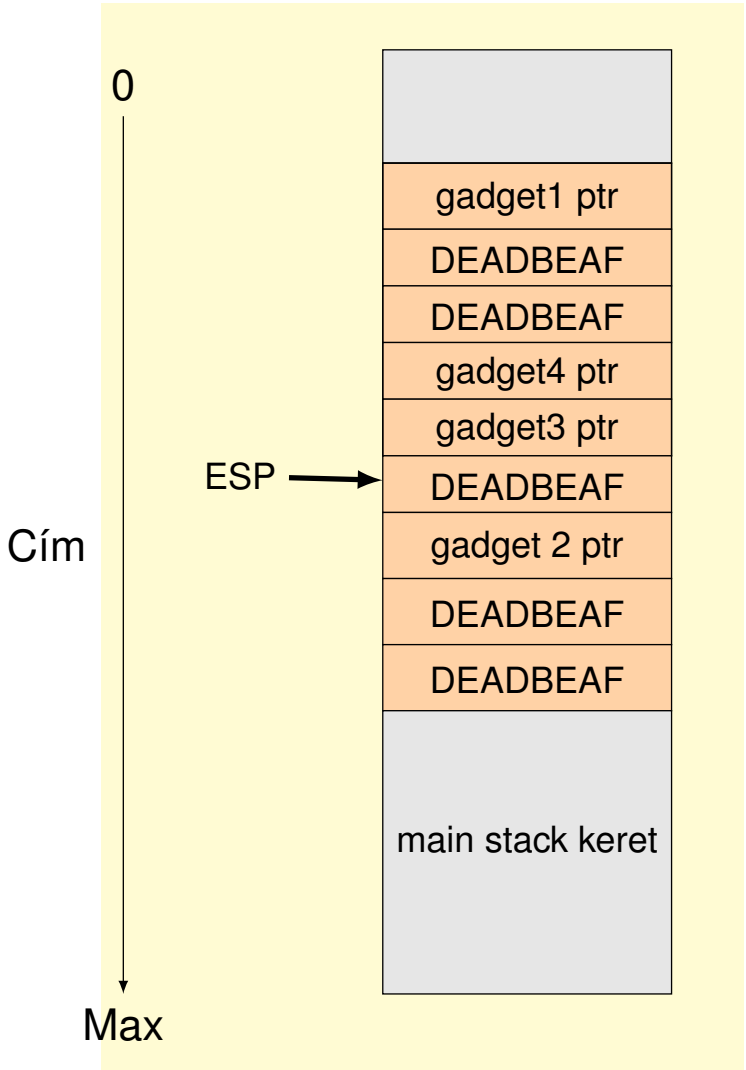


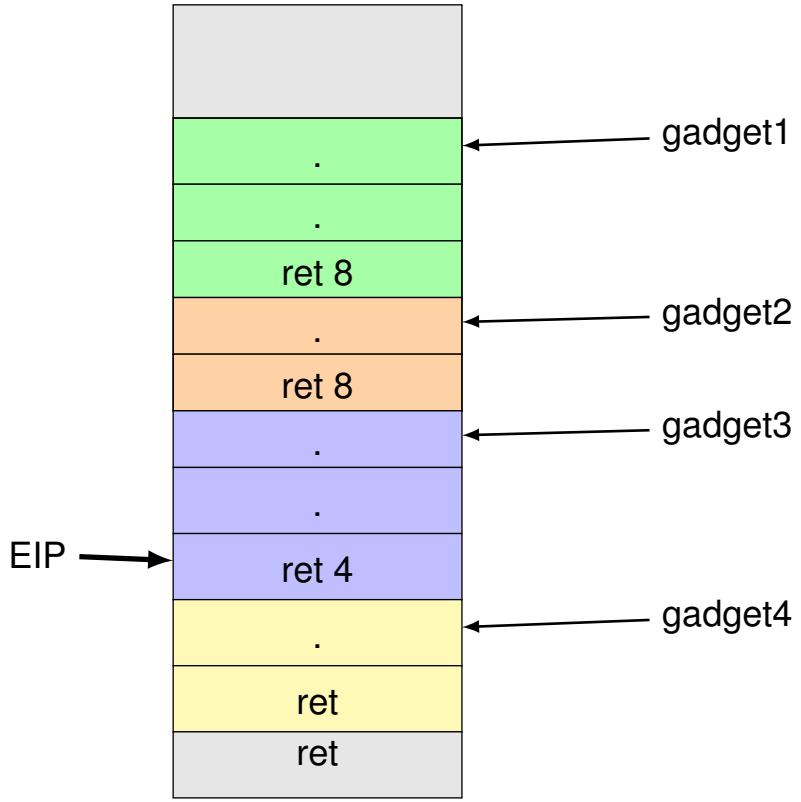
ROP kód működése



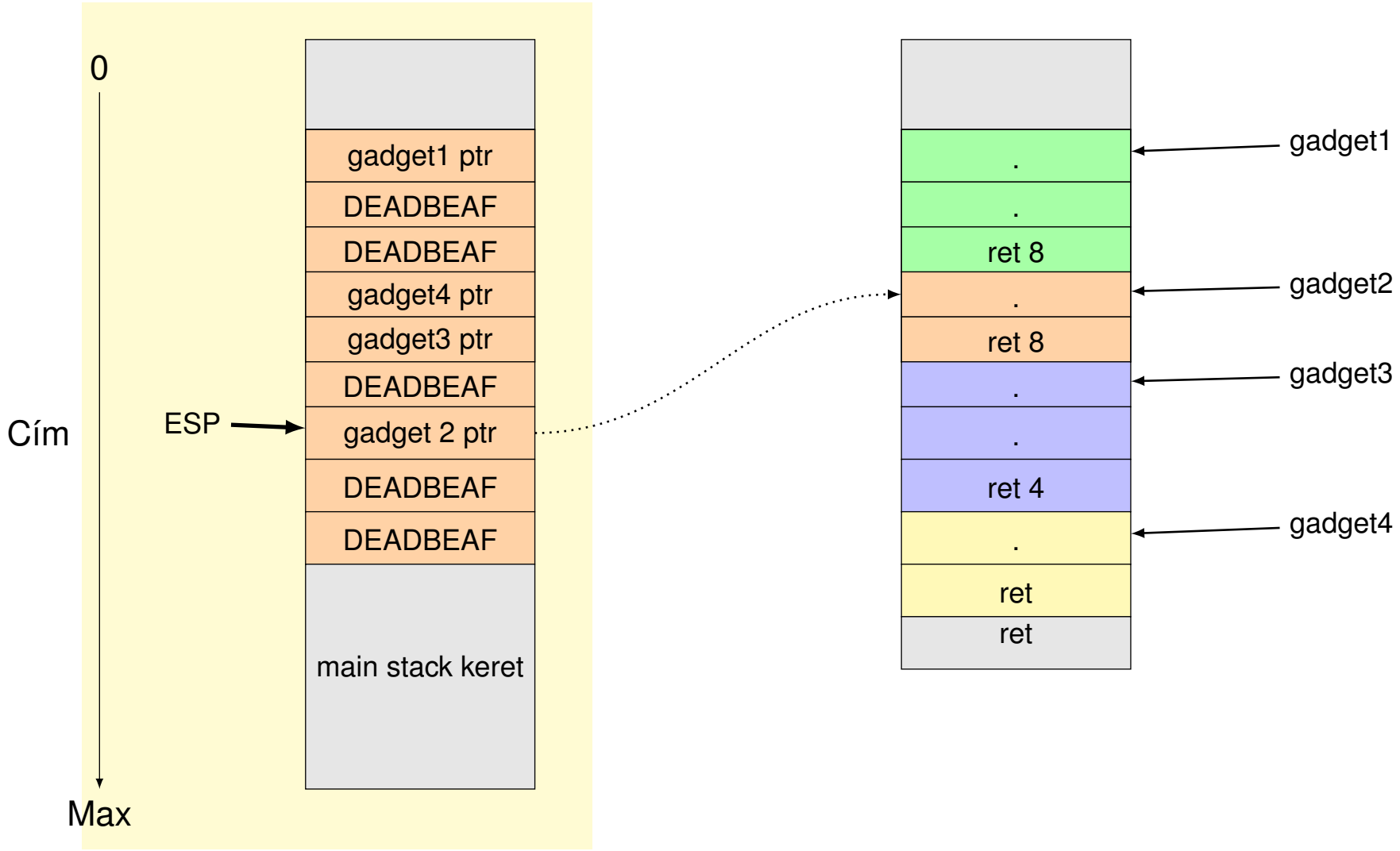


ROP kód működése

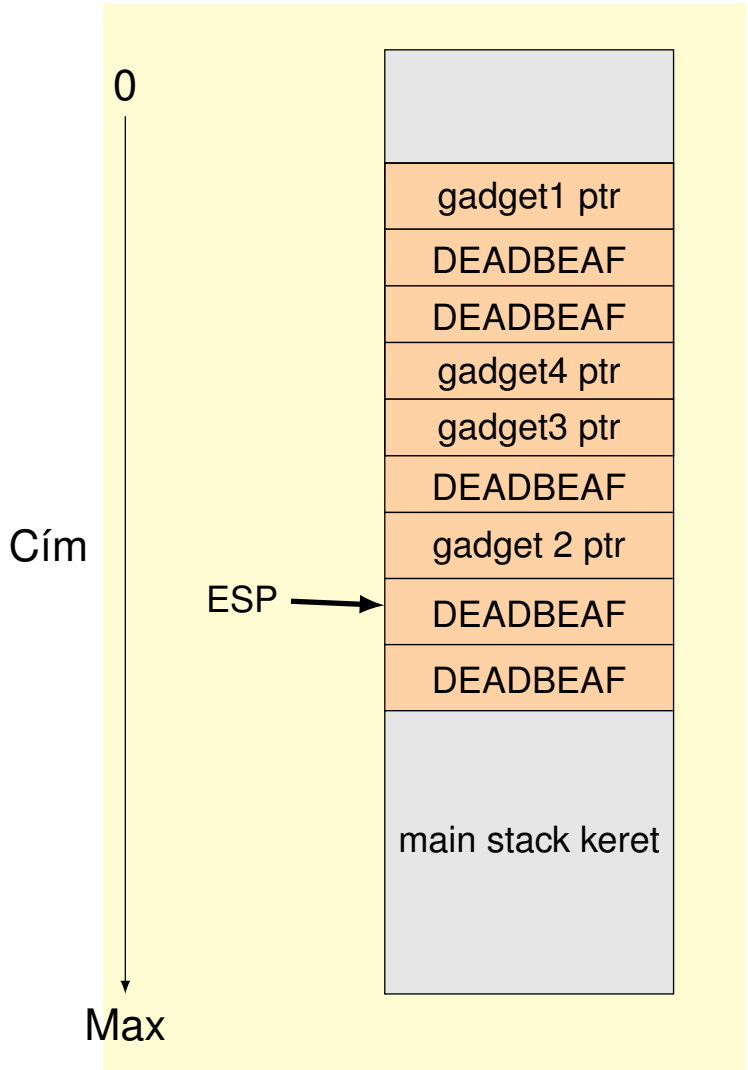


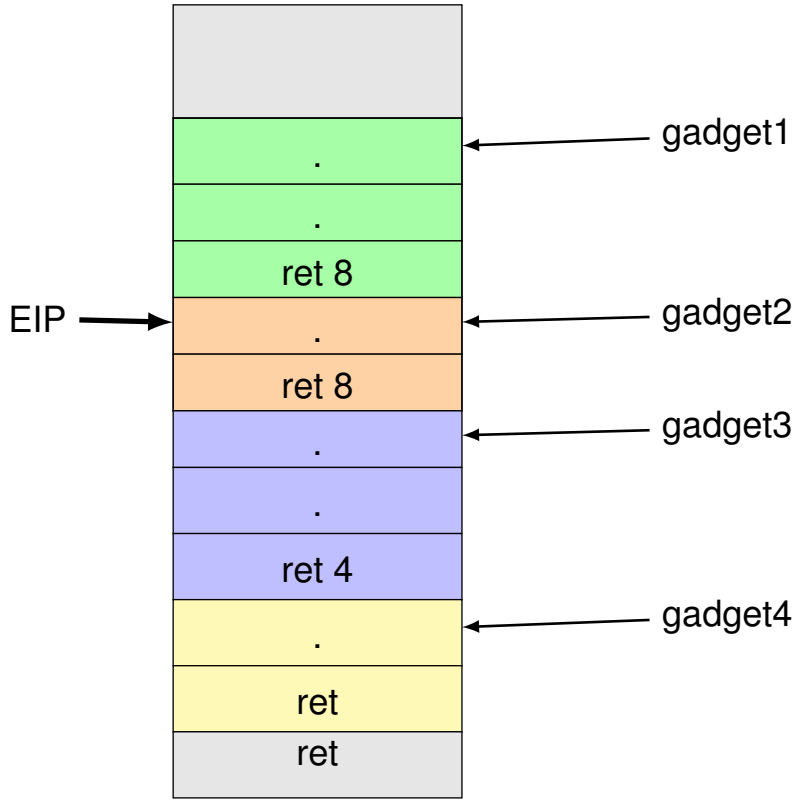


ROP kód működése

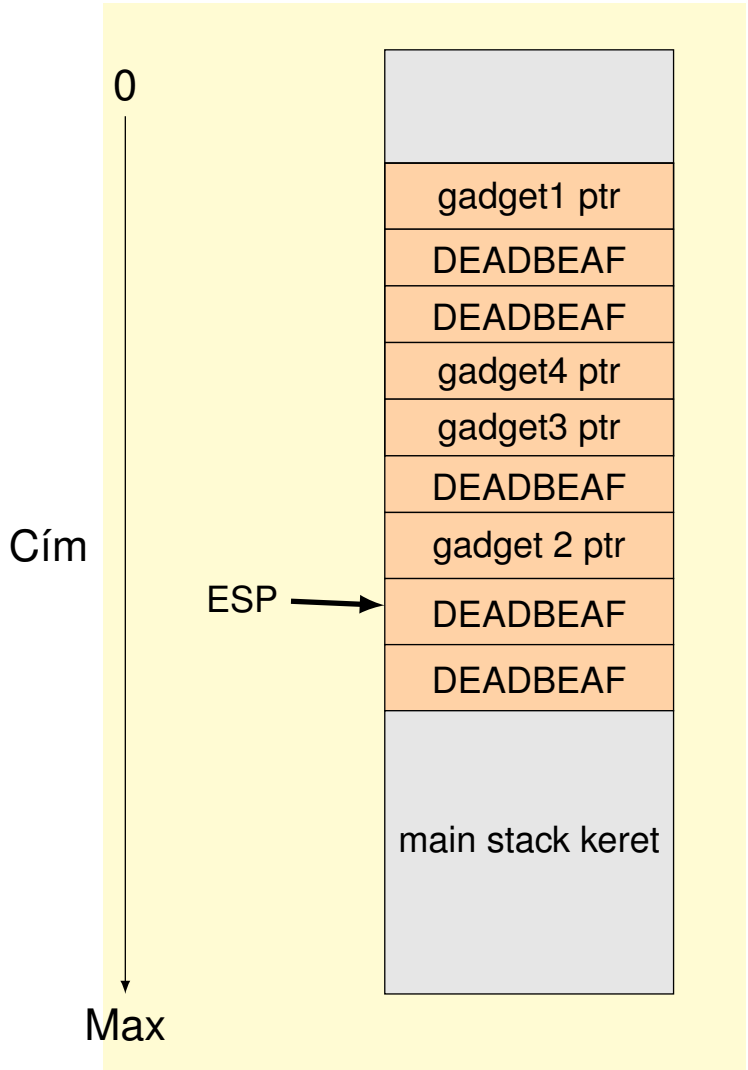


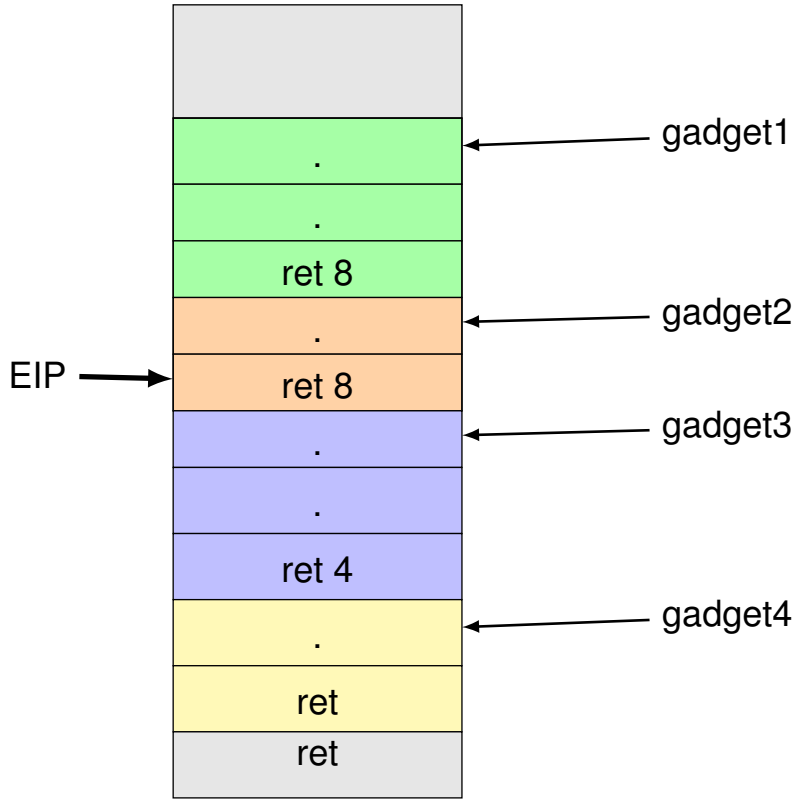
ROP kód működése





ROP kód működése





Section 2

Védekezés a ROP (Ret2CLib) programok ellen

ROP program megakadályozása

ROP program esetén *gadget*-eket (**ret**-tel végződő utasításokat) hívunk meg. Ha a kikeresett *gadget* címe nem fix, akkor ezt nem tehetjük meg.

Megoldás

Address Space Layout Randomization

Address Space Layout Randomization (ASLR)

ASLR esetén minden rendszerindítás után változik a processzek, modulok

- betöltési címe (image base address),
- stack kezdetének a címe,
- heap-en lefoglalt adatok címe.

Visual Studio esetén a /DYNAMICBASE linker opcióval adhatjuk meg.

A program áthelyezhetősége miatt tudnunk kell, hogy hol vannak olyan adatok, amelyek függenek a betöltés helyétől. Ezt a relokációs tábla írja le. Sok program fix címre lett linkelve, így ezek áthelyezése lehetetlen.

ASLR működésének korlátai 32 bites cím esetén

A program betöltés során a szegmensek laphatáron 4Kbyte (2^{12}) kezdődnek. Ezért az alsó 12 bit nem fog változni, ha a szegmens más lapra kerül. 32 bites operációs rendszerben ASLR esetén ráadásul csak a felső 16 bit változik véletlenszerűen, az alsó nem. A felső címbitekből is csak 8 vesz fel véletlen érték, ami 256 lehetséges esetet jelent.

ASLR átverése I

1. Adott a 0x12345678 visszatérési cím.[3] 0x1234 érték a következő boot esetén változhat, de a 0x5678 nem. Ne írjuk át puffer túlcímzéssel a felső cím értékeket. Ez megtehetjük, mivel Little-Endian ábrázolásban a cím a memóriában 0x78, 0x56, 0x34, 0x12 alakban követik egymást. String puffer esetén a felülírt adatterület a 0x56 byte címen végződjön, ahova a stringet lezáró 0 fog kerülni. Felülírt terület a memóriában 0x??, 0x00, 0x34, 0x12 értékeket tartalmazza, ami az Little-Endia ábrázolás miatt a 0x123400?? címet jelenti. A kérdőjel helyén akármilyen érték szerepelhet, így 256 különböző címet definiálhatunk. Ha ezeken a helyeken van olyan utasítás, amivel a shellcode-unkra ugorhatunk, akkor az ASRL-t átvertük.

ASLR átverése II

2. Ha létezik olyan modul (DLL) ami nem /DYNAMICBASE opcióval lett készítve, akkor a már megismert, a modul címterében lévő **jmp esp**-s trükkel meghívhatjuk a shellcode-unkat.
3. ASLR és DEP kétségtelenül a legnehezebb, és jelenleg csak szerencsével törhető fel. Első ASLR feltörés az animated cursor gyengeségét használta ki (ma már a biztonsági rést befoltozták). Mivel 256 *slot* lehet az ASLR-rel fordított ntdll.dll module címe, ezért nyers erő módszerrel max 256 kurzor létlehozásával törhető volt a rendszer.

[3] Ez csak 32 bites címre igaz, 64 bitesre nem!

Heap spray

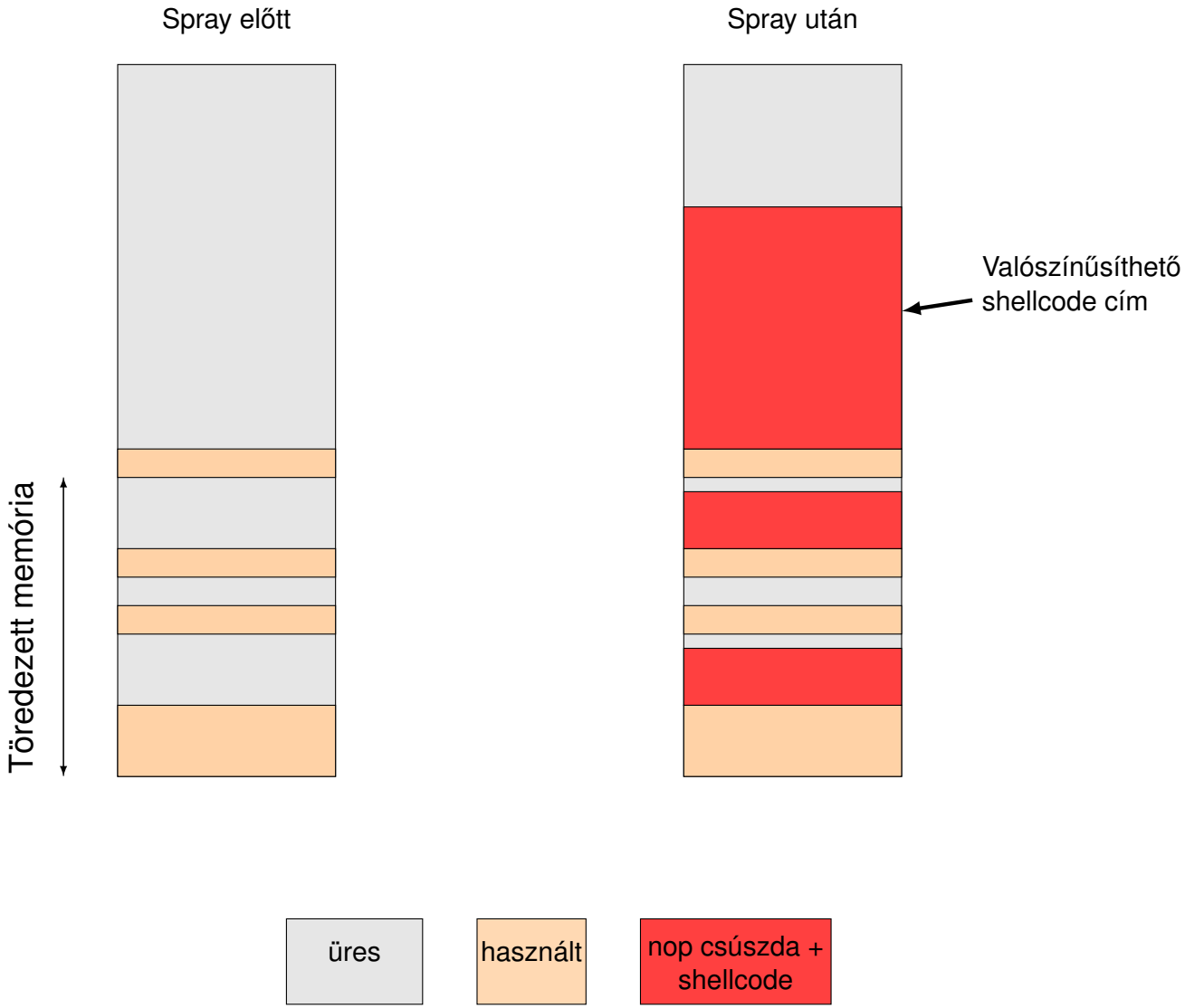
Memóriakezelő függvények:

- VirtualAlloc(), VirtualAllocEx() memória lefoglalás,
- VirtualFree(), VirtualFreeEx() memória felszabadítás.

Ezekre a függvényekre épülnek a C standard könyvtári malloc, calloc, realloc, free, függvények is.

A lefoglalás és a lefoglalt területek egy részének a felszabadítása után a memóriában lyukak keletkeznek. A következő memóriafoglalások előbb a lyukakat töltik ki, és csak azután a *szűz* területeket. Megfelelő mennyiségű memóriafoglalás után a *bepermetezett* terület címe nagy valószínűséggel megmondható.

Heap képe permetezés előtt és után



Heap permetezésének megvalósítása

Azoknál a programoknál lehetséges, ahol script programot lehet futtatni. Például:

- Internet Explorer, FireFox, és a többi böngésző. (linuxon is működik). Programozási nyelvük javascript.
- Word for Windows, Excel, PowerPoint. Programozási nyelvük Visual Basic, C♯.
- Acrobat Reader. Programozási nyelve javascript.
- Adobe Flash Actionscript stb.

Heap permetezésének előnye

A heap permetezés a shellcode egyik bevitelének hatékony módja. A kód meghívását nem oldja meg, ahhoz valamilyen biztonsági rés megléte is szükséges.

Előnyök:

- Safe SEH esetén a heap-en lévő kód futtatható.
- A heap DEP-pel alapból nem védett. Az újabb IE esetén már védett, de ROP programhoz továbbra is használható.
- Sok esetben a stack-en nem fér el a shellcode. Heap esetén ez a korlát értelemszerűen megszűnik.

Érdekes cím 0x0c0c0c0c I

Ez már a heap magasabb címtartományába fekszik, mégis van egy kis előnye oda helyezni a shellcode-ot.

Nézzük meg hogyan működik objektum orientált kódnál a virtuális függvény meghívása.

1. Objektum címének a felvétele.
2. Objektum elejéről a virtuális függvénytábla címének a kiolvasása.
3. A virtuális függvénytáblában lévő valamelyik függvény címének a kiolvasása. (Az egyszerűség kedvéért a virtuális függvénytáblát tekinthetjük egy függvényre mutató pointereket tartalmazó tömbnek.)
4. Virtuális függvény meghívása.

Érdekes cím 0x0c0c0c0c II

- Bepermetezzük a 0x0c0c0c0c címen lévő heap területet is.
- **nop** csúszda 0x90 helyett is 0x0c-t írunk. 0x0c0c az **or al**,0c utasítás kódja.
- Felülírjuk az objektum címét 0x0c0c0c0c-vel, akkor azon a helyen keresi a virtuális függvénytáblára mutató pointert $\mapsto$ ahonnan felveszi a virtuális függvénytábla címét, ami persze 0x0c0c0c0c $\mapsto$ amely helyről kiolvassa a virtuális függvényre mutató pointert, ami szintén 0x0c0c0c0c. A végén meghívja a 0x0c0c0c0c címen lévő kódot, aminek a tartalma 0x0c0c ami **nop** csúszdának is megfelel.

Mi van, ha az objektumban lévő virtuális függvénytáblára mutató pointert írjuk felül? Ugyanez történik eggyel kevesebb indirekcióval.