Voorbeeld 2a

Actie 1

Compileer dit programma en geef de executable de naam : **`vb1a`**.
Je kunt dit in 1 commando doen !

Vraag 1 : waarvoor dienen deze push en pop acties?

> De methode moet de situatie (in de registers) achterlaten in dezelfde toestand als het heeft aangetroffen.
> Je weet immers niet of een voorgaande methode data in de registers heeft geplaatst.
> Hiervoor wordt vooraf een kopie v/d registers gemaakt op de stack, en na afloop weer teruggeplaatst.

Vraag 1b : waarom staan de pop acties (r13,r12,rbx) in omgekeerde volgorde van de push acties (rbx,r12,r13) ?

> Dit heeft te maken met de LIFO werking van de stack.

Actie 2

Run het gemaakte programma **`vb1a`** en bepaal de return waarde van dit programma.

Vraag 2 : geef bij elke regel de inhoud van alle relevante registers na de actie.
Gebruik hiervoor een tabel.

#	Opcode	EBX	R12	R13	EAX
6	mov ebx,0x0	0	?	?	?
7	mov r12d,0x67	0	103	?	?
8	mov r13d,0xdd	0	103	221	?
9	mov ebx,r13d	221	103	221	?
10	mov r13d,r12d	221	103	103	?
11	mov r12d,ebx	221	221	103	?
12	shl ebx,0x2	884	221	103	?
13	mov eax,ebx	884	221	103	884

In deze tabel staan de getallen in decimale notatie.

Actie 3

Vraag 3 : wat is de return waarde van dit programma?

> $ **`echo $?`** geeft : 116

Vraag 3b : Komt dit overeen met de waarde in EAX ?

De return waarde komt niet overeen met eax maar wel met al : de least signifcant byte van dit register.
Je kunt dit berekenen m.b.v. 884 % 256 = 116
of in hexnotatie :
884 = 0374
116 = 74

en inderdaad de return waarde van een ELF programma is altijd 1 byte groot