TD - Analyse de vulnérabilités logiciel

Exercice 1 (Instruction mov)

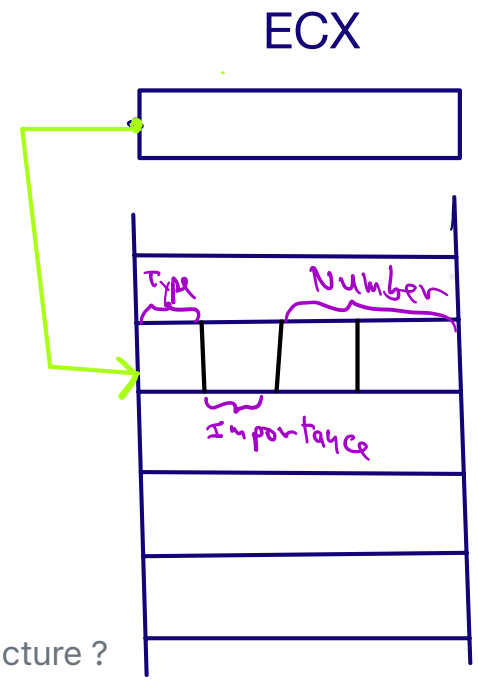
> Donnez le résultat de chaque instruction du code suivant :

```
mov dword ptr [eax], 1
mov ecx, [eax]
mov [eax], ebx
mov [esi+34h], eax
mov eax, [esi+34h]
mov edx, [ecx+eax]
```

Exercice 2 (Registre de base et offset)

Soit le registre `ECX` qui contient l'adresse de la structure système `KDPC` dont la forme est la suivante:

```
offset  name              type
-------------------------------------
+0x000  Type :            char
+0x001  Importance :          char
+0x002  Number :          short
+0x004  DpcListEntry :    _LIST_ENTRY
+0x00c  DeferredRoutine :   Ptr32 void
+0x010  DeferredContext :   Ptr32 Void
+0x014  SystemArgument1 :   Ptr32 Void
+0x018  SystemArgument2 :   Ptr32 Void
+0x01c  DpcData :         Ptr32 Void
```



> Quelle est la taille de chaque champ de cette structure ?

> Donnez le résultat de chaque instruction du code ci-dessous, en suposant que l'architecture est de type little endian.

```
01: mov eax, [ebp+0Ch]
```

```
02: and dword ptr [ecx+1Ch], 0
03: mov [ecx+0Ch], eax
04: mov eax, [ebp+10h]
05: mov dword ptr [ecx], 113h
06: mov [ecx+10h], eax
```

Mettre DpcData à 00000000
ARG1 → DeferredRoutine
ARG2 → EAX
ARG2 → Deferred Context
{ Type : 13
Importance : 01
Number : 0000

> Donnez la différence avec le code suivant et conclure sur la signification de `byte ptr` et `word ptr`.

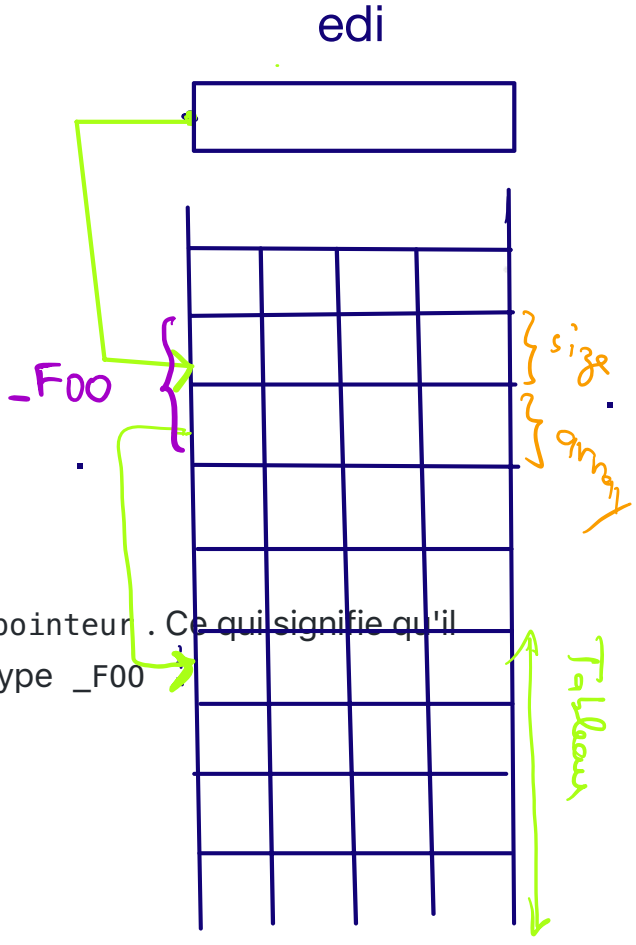
```
01: mov eax, [ebp+0Ch]
02: and dword ptr [ecx+1Ch], 0
03: mov [ecx+0Ch], eax
04: mov eax, [ebp+10h]
05: mov byte ptr [ecx],13h
06: mov byte ptr [ecx+1],1
07: mov word ptr [ecx+2],0
08: mov [ecx+10h], eax
```

Cela donne le même Résultat, comme on est en Little Endian
13 → Type
01 → Importance
0000 → Number

Exercice 3 ([Base + Index * scale])

- Soit la structure `_FOO` ci-dessous. Les variables de type `DWORD` ont une taille de `4 bytes`.

```
typedef struct _FOO
{
DWORD size;
DWORD array[...];
} *PFOO;
```



- Soit la variable `pointeur` de type `_FOO`.

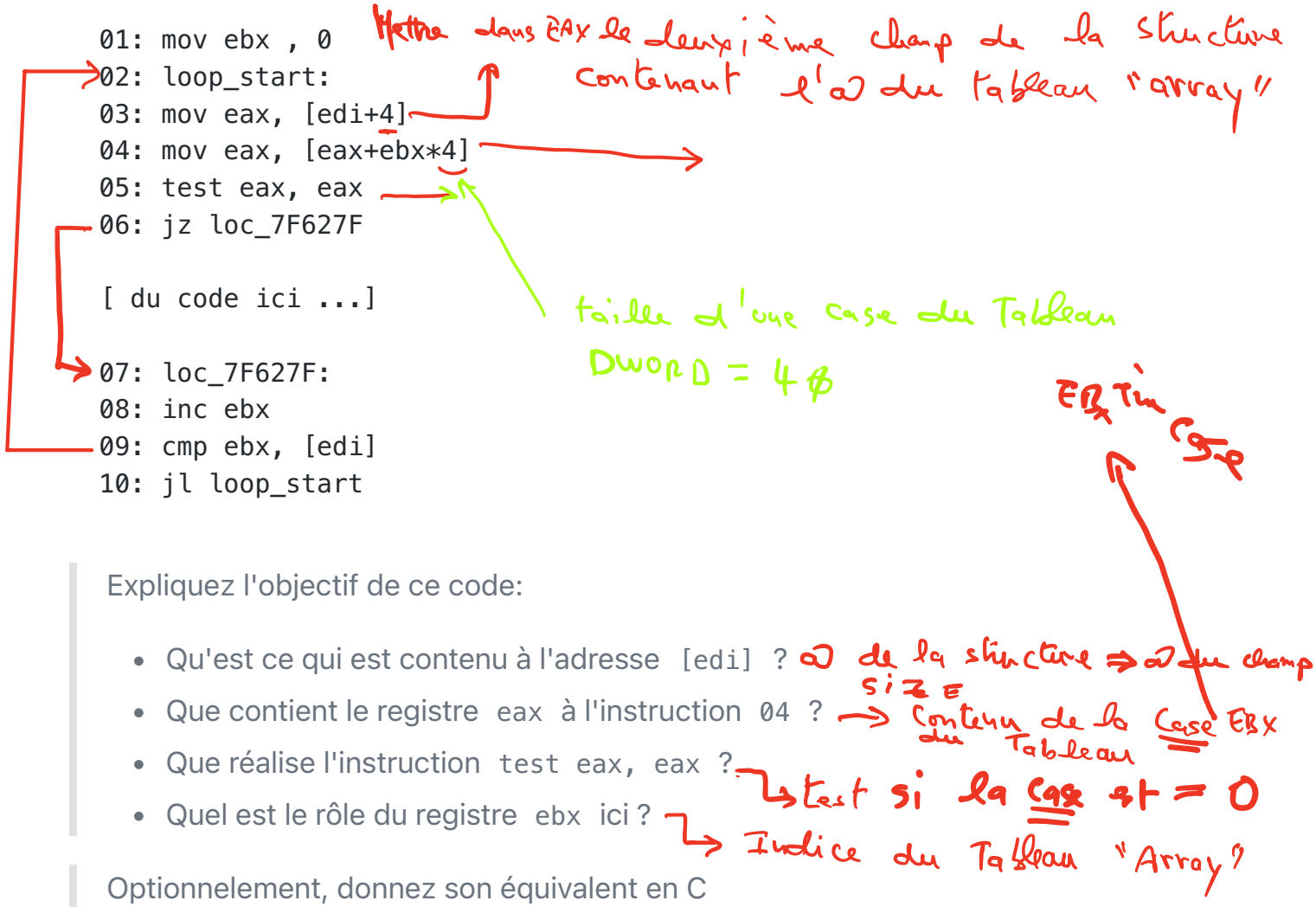
```
PFOO pointeur = ...;
```

- Le registre `edi` représente ici le pointeur `pointeur`. Ce qui signifie qu'il contient l'adresse d'une zone mémoire de type `_FOO`
- Soit le code assembleur suivant.

```
01: mov ebx , 0
02: loop_start:
03: mov eax, [edi+4]
04: mov eax, [eax+ebx*4]
05: test eax, eax
06: jz loc_7F627F

[ du code ici ...]

07: loc_7F627F:
08: inc ebx
09: cmp ebx, [edi]
10: jl loop_start
```



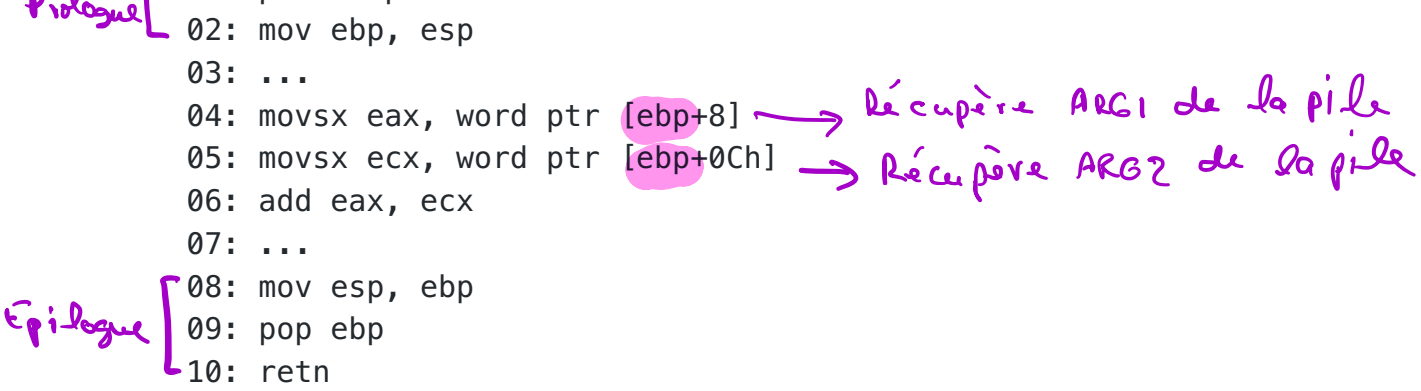
> Expliquez l'objectif de ce code:
>
> - Qu'est ce qui est contenu à l'adresse `[edi]` ?
> - Que contient le registre `eax` à l'instruction `04` ?
> - Que réalise l'instruction `test eax, eax` ?
> - Quel est le rôle du registre `ebx` ici ?
>
> Optionnelement, donnez son équivalent en C

Exercice 4:

- Avant de faire cet exercice, je vous invite à regarder la section `fonctionnement de la pile` du cours et de regarder la vidéo réalisée à cet effet.
- Soit le code assembleur suivant

```
addme:

01: push ebp
02: mov ebp, esp
03: ...
04: movsx eax, word ptr [ebp+8]
05: movsx ecx, word ptr [ebp+0Ch]
06: add eax, ecx
07: ...
08: mov esp, ebp
09: pop ebp
10: retn
```



```
main:

01: push eax
02: ...
03: push ecx
04: call addme
05: add esp, 8
```

- Identifiez le `proloque` de la fonction `addme`
- Identifiez le `èpilogue` de la fonction `addme`
- A quel endroit du code la fonction `addme` est appelée ? ligne 4 : call addme
- Que réalise l'instruction `03` de la fonction `main` ? Met ARG2 dans la pile
- Que réalisent les instuctions `04` et `05` de la fonction `addme` ?
- Que réalise l'instruction `05` de la fonction `main` ? Nettoie la pile des arguments après l'appel de la fonction

Exercice 5

Expliquez le code ci-dessus et donnez optionnellement son équivalent en C

```
01: mov edi, ds:__imp__printf   ; adr de la fct "printf"
02: xor esi, esi                ; esi ← 0 (indice i)
03: lea ebx, [ebx+0]

04: loc_401010:

05: push esi                    ; ← ARG2
06: push offset Format ; "%d\n" ; ← ARG1
07: call edi ; __imp__printf
08: inc esi
09: add esp, 8                  ; ← Clean Arguments (ARG1, ARG2) 4+4 = 8
10: cmp esi, 0Ah                ; 10
11: jl short loc_401010
12: push offset aDone ; "done!\n" ; ← ARG de printf
13: call edi ; __imp__printf
14: add esp, 4                  ; ← clean Arguments
                                ; Un seul (4Φ)
```

```
i = 0;
while (i < 10)
{
  printf("%d\n", i);
} i++;
printf("done!\n");
```