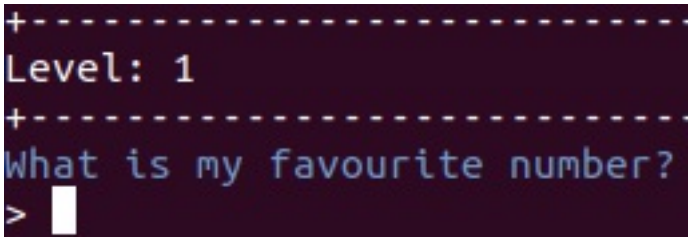
Gollum

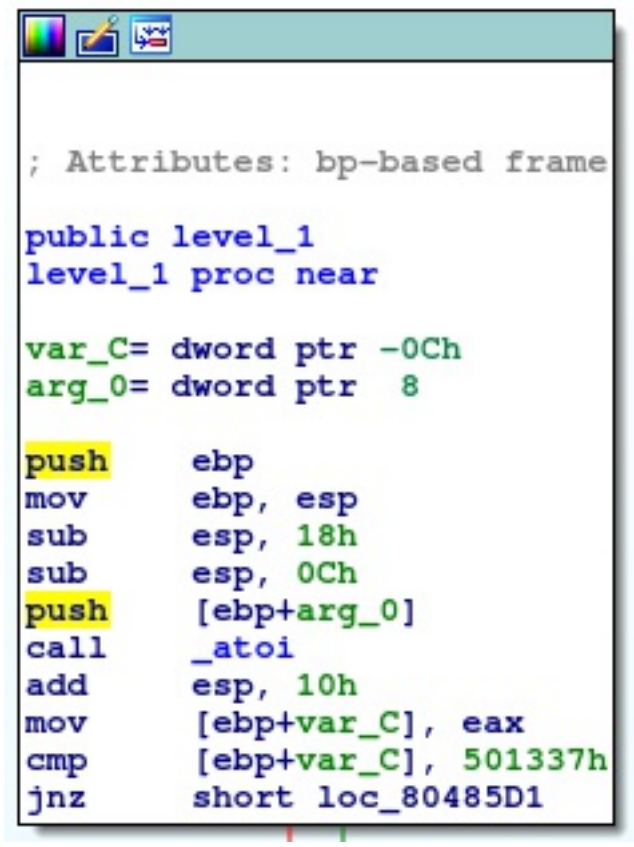
- Arthur CATRISSE
- Kaan DAGERI
- Hugo DALBOUSSIERE

Level 1

Le premier niveau nous demande d'entrer un nombre :



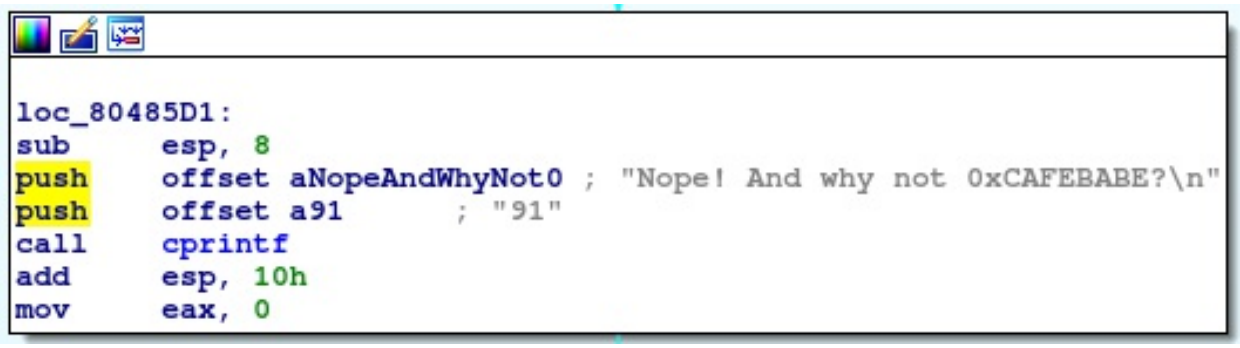
Etudions le premier bloc.



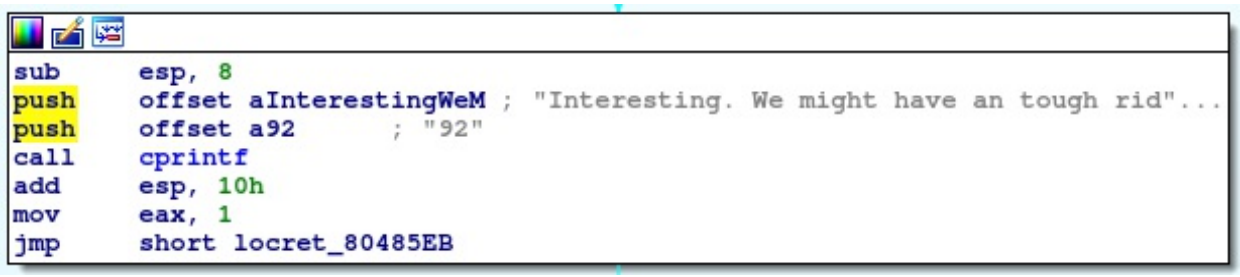
```
; Attributes: bp-based frame

public level_1
level_1 proc near

var_C= dword ptr -0Ch
arg_0= dword ptr  8

push    ebp
mov     ebp, esp
sub     esp, 18h
sub     esp, 0Ch
push    [ebp+arg_0]
call    _atoi
add     esp, 10h
mov     [ebp+var_C], eax
cmp     [ebp+var_C], 501337h
jnz     short loc_80485D1
```

On push l'entrée utilisateur dans la stack afin d'appeler la fonction atoi.

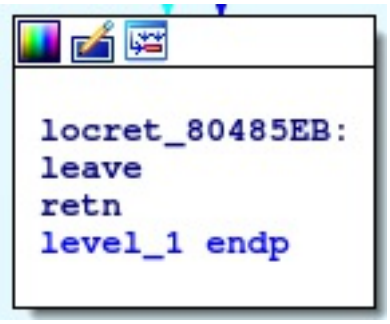
Cette fonction convertit la string en un int.
On récupère le résultat de eax et on le déplace dans l'adresse ebp+var_C puis on compare cette valeur à 501337h ou encore 5247799 en décimal.
Si cela ne correspond pas (jump if not zero) on saute vers le bloc suivant qui affichera un message d'erreur puis quittera la fonction level_1 :



Sinon, l'entrée utilisateur correspond (le zero flag est à 1) et l'on saute sur le bloc suivant qui affiche un message de succès en vert :



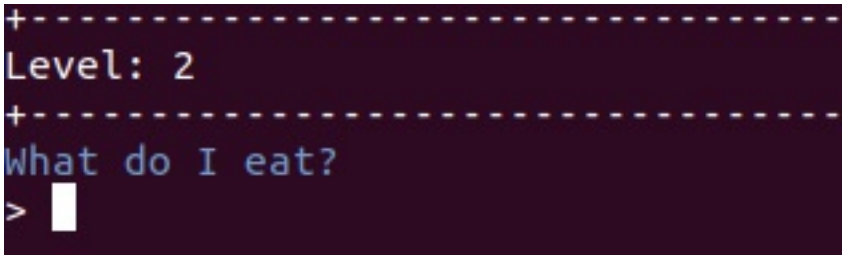
On met aussi 1 dans eax pour indiquer la bonne terminaison de la fonction.
Enfin, on arrive au dernier bloc qui ne fait que retourner à la fonction appelante.



Le flag est donc 5247799.

Level 2

Le second niveau nous demande d'entrer un mot de passe :



Etudions le premier bloc.

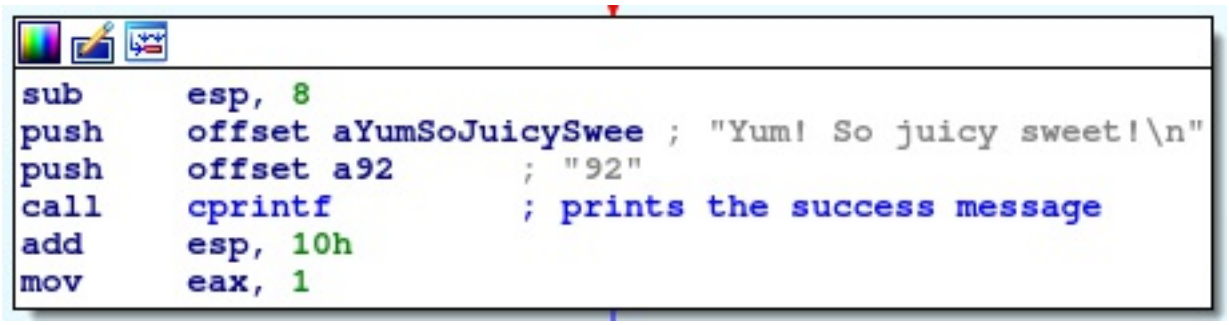
```
; Attributes: bp-based frame

public level_2
level_2 proc near

user_input_length= dword ptr -1Ch
long_text_length= dword ptr -18h
long_text= dword ptr -14h
counter_2= dword ptr -10h
counter_1= dword ptr -0Ch
user_input= dword ptr  8

push    ebp
mov     ebp, esp
sub     esp, 28h
mov     [ebp+long_text], offset aAaaAasaabaaAan ; "AAA%AAsAABAA$AAnAACAA-AA(AADAA;AA)AAEAA"...
sub     esp, 0Ch
push    [ebp+long_text]
call    _strlen
add     esp, 10h
mov     [ebp+long_text_length], eax
sub     esp, 0Ch
push    [ebp+user_input]
call    _strlen
add     esp, 10h
mov     [ebp+user_input_length], eax
mov     [ebp+counter_1], 0
mov     [ebp+counter_2], 256
jmp     short loc_804866A
```

On peut voir qu'une longue chaîne de caractères est stockée à l'adresse ebp+long_text et est poussée dans la stack.
Aussi, sa longueur est stockée à l'adresse ebp+long_text_length.
L'entrée utilisateur est stockée à l'adresse ebp+user_input et sa longueur est stockée à l'adresse ebp+user_input_length.
Enfin deux compteurs sont initialisés et stockés :
counter_1 à 0 dans l'adresse ebp+counter_1
counter_2 à 256 dans l'adresse ebp+counter_2

```
loc_804866A:
mov     eax, [ebp+counter_2]
cmp     eax, [ebp+long_text_length]
jb      short loop      ; jumps if the length of the long_text is below the counter_2 value
```

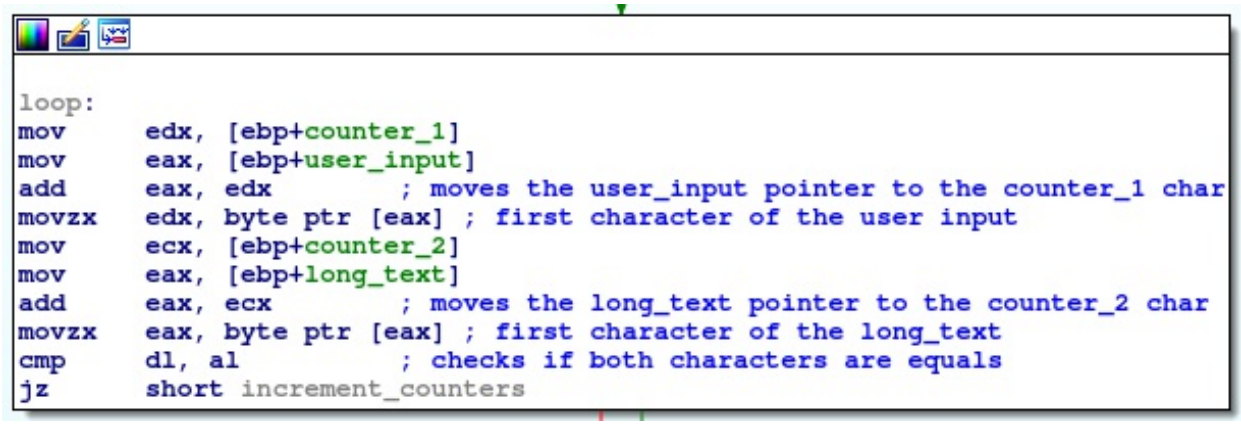
Dans le bloc suivant, on aperçoit un jump conditionnel qui est valide uniquement lorsque la

longueur de long_text est inférieure à la valeur de counter_2.
Si la condition est validée, le bloc suivant affichera un message de succès en vert puis quittera la fonction level_2 :



```
sub     esp, 8
push    offset aYumSoJuicySwee ; "Yum! So juicy sweet!\n"
push    offset a92             ; "92"
call    cprintf                ; prints the success message
add     esp, 10h
mov     eax, 1
```

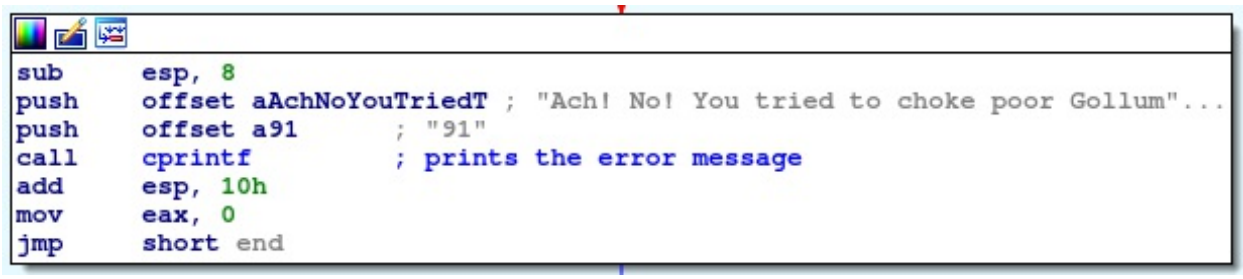
Sinon, on continue dans un bloc qui semble être une boucle :



```
loop:
mov     edx, [ebp+counter_1]
mov     eax, [ebp+user_input]
add     eax, edx          ; moves the user_input pointer to the counter_1 char
movzx   edx, byte ptr [eax] ; first character of the user input
mov     ecx, [ebp+counter_2]
mov     eax, [ebp+long_text]
add     eax, ecx          ; moves the long_text pointer to the counter_2 char
movzx   eax, byte ptr [eax] ; first character of the long_text
cmp     dl, al            ; checks if both characters are equals
jz      short increment_counters
```

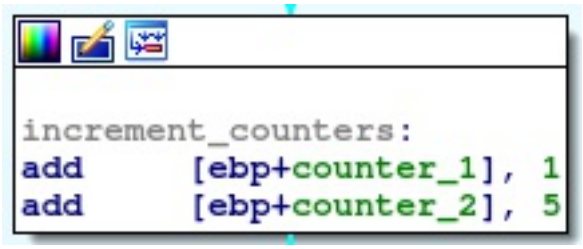
Cette boucle déplace les pointeurs des chaînes de caractères user_input et long_text en fonction des valeurs de counter_1 et counter_2.
counter_1 s'incrémentant toujours de 1, le pointeur de user_input se déplacera de 1 caractère.
counter_2 s'incrémentant toujours de 5, le pointeur de long_text se déplacera de 5 caractère.
Attention, il faut prendre en compte que counter_2 a été initialisé à la valeur 256.
Enfin, on récupère les premiers caractères de user_input et de long_text puis on les compare pour voir s'ils sont identiques.

S'ils ne sont pas identiques, le bloc suivant affichera un message d'erreur en rouge puis quittera la fonction level_2 :



```
sub     esp, 8
push    offset aAchNoYouTriedT ; "Ach! No! You tried to choke poor Gollum"...
push    offset a91             ; "91"
call    cprintf                ; prints the error message
add     esp, 10h
mov     eax, 0
jmp     short end
```

S'ils sont identiques, on passe au bloc suivant qui va incrémenter les compteurs :



```
increment_counters:
add     [ebp+counter_1], 1
add     [ebp+counter_2], 5
```

Après l'incrémentation des compteurs, on revient dans la boucle pour tester les prochains caractères.

Suite à ces observations, on peut déduire que le flag à trouver commence au caractère 256 de la chaîne de caractères long_text et contiendra toutes les lettres qui se situent par incrément de 5 plus loin, jusqu'à ce que l'on sorte de long_text.
Le counter_1 ne sert qu'à itérer sur l'entrée de l'utilisateur.
On peut faire un script Python simpliste qui implémente ce comportement :

```python
#!/usr/bin/python
mysterious_string = "AAA%AAsAABAA$AAnAACAA-AA(AADAA;AA)AAEAAaAA0AAFAAbAA1AAGAAcAA2A

index = 256
done = False
flag = ''

while not done:
    if len(mysterious_string) >= index:
        flag += mysterious_string[index]
        index += 5
    else:
        done = True

print "The flag is " + flag
```

On trouve alors le mot de passe en clair : cavefishandgoblin.

Level 3

En premier lieu, on teste le lvl 3, et on parvient à obtenir 2 messages d'erreur différents. Le premier lorsqu'on écrit quelque chose, et le deuxième lorsqu'on tape « entrée » sans rien écrire.

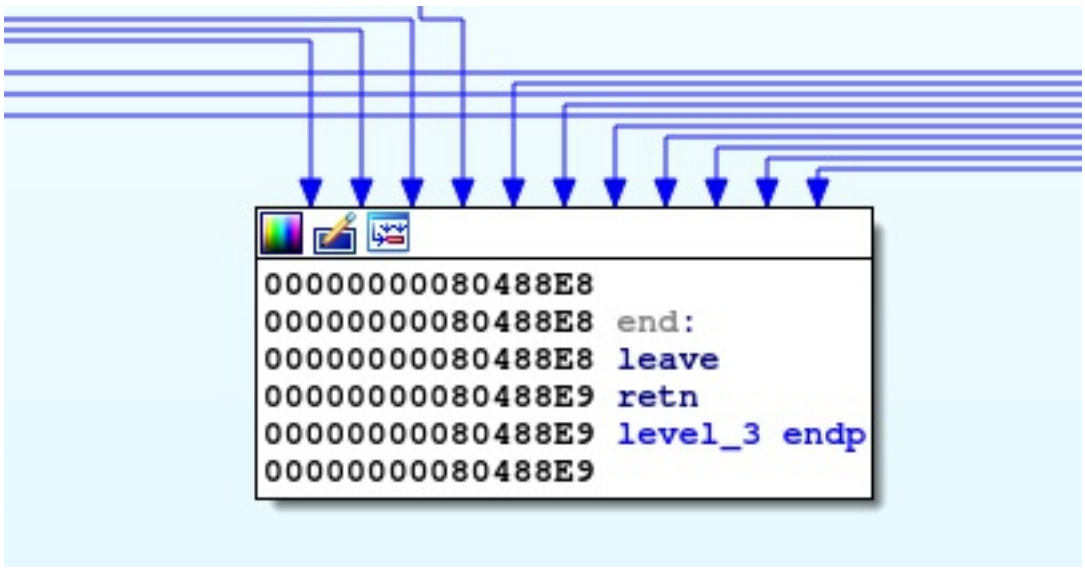
```
+-----------------------------
Level: 3
+-----------------------------
What is my name?
>
Pfff, can't you do better?!
```

```
What is my name?
>
Pfff, can't you do better?!
```

Une fois dans IDA, après avoir remarqué que le schéma représentant le code de lvl_3 était très étendu en largeur, nous avons choisi d'analyser en premier les blocs du bas. Il y a 2 blocs vers lesquels arrivent tous les autres.
Sur celui-ci, qui est situé au milieu en bas, on constate un return avec l'instruction « retn ». C'est donc le bloc qui terminera le programme. On renomme ce bloc en « end : ».

```
What is my name?
> arthur
Can't even get the letter at position 0, so lame!
```

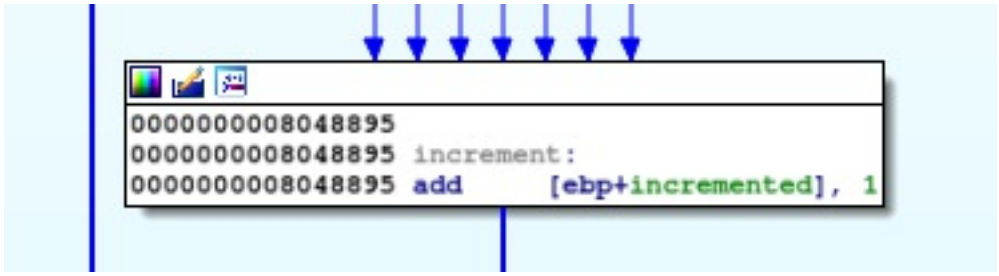
Sur le deuxième bloc situé en bas (à gauche), la seule instruction effectuée est celle de rajouter 1 à une valeur située dans une case mémoire, puis de reboucler vers un bloc situé bien plus haut. C'est donc une instruction d'incrémentation. On renomme donc le bloc en « increment : » et la case mémoire en « [ebp+incremented] ».



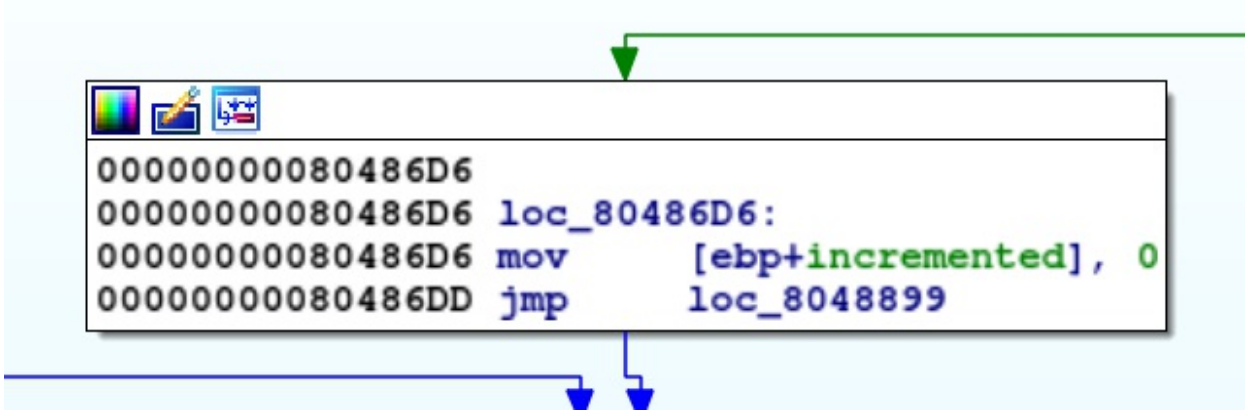
En remontant les blocs d'un niveau vers le haut, on différentie 2 types de blocs. Les premiers affichent un message, en appelant la fonction cprintf. Le message est stocké en tant que

chaine de caractères grâce à « db ». Parmi les messages qui existent, on retrouve les 2 sur lesquels on était tombés en testant le programme. Après avoir affiché le message, le bloc d'instructions effectue un jump au bloc que nous avons renommé « end » pour terminer le process. Le deuxième type de bloc ne fait rien mis à part un jmp au bloc que nous avons renommé « increment ».

On va maintenant remonter en haut des blocs.
Le bloc tout en haut est celui qui est appelé lorsqu'on choisit le niveau 3. On sait que c'est dans ce bloc que le code teste la présence ou l'absence de lettres entrées par l'utilisateur, puisque le jump conditionnel peut aller directement au bloc de message d'erreur que l'on reçoit lorsqu'on ne rentre aucun caractère avant de valider.
Dans le cas où des caractères sont rentrés, on tombe sur un petit bloc d'instructions qui initialise à 0 la valeur de [ebp+incremental] grâce à l'instruction mov.

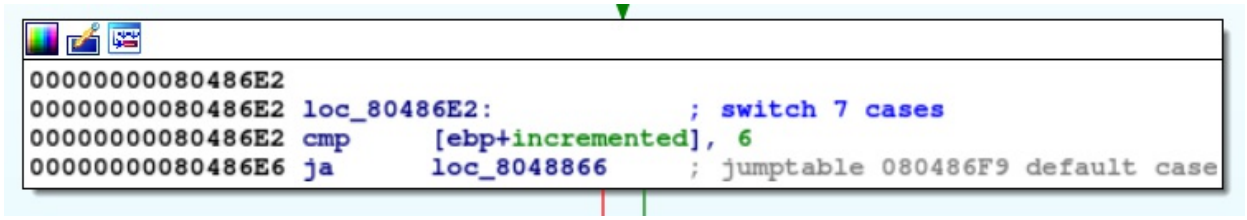


Ensuite, le bloc jump jusqu'à un autre bloc, le même que celui vers lequel le bloc d'incrémentation retourne. C'est donc le bloc de départ d'une boucle. La sortie de cette boucle s'effectue uniquement lorsque le joueur a trouvé la solution, donc lorsque le programme a itéré à travers toutes les lettres rentrées sans être tombé sur un message d'erreur (et donc être sorti du programme plus tôt que prévu). Le test de sorti s'effectue entre les valeurs contenues dans eax et edx. Sachant que notre valeur incrémentée [ebp+incremented] a été mov dans eax, la sortie de cette boucle s'effectue bien lorsque notre incrémentation atteint une certaine valeur.

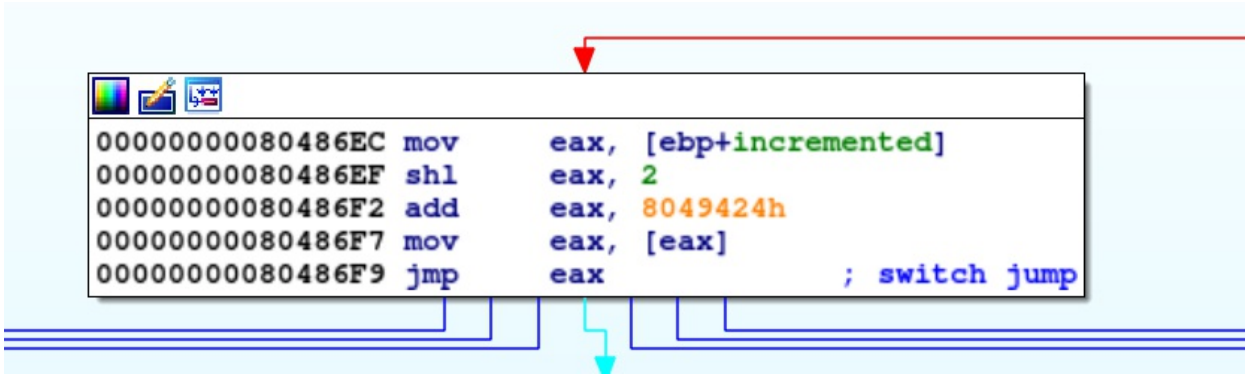


En descendant d'un niveau, on tombe sur un bloc qui nous donne un indice, en comparant la valeur de [ebp+incremented] à 6 avec cmp. De plus, derrière cette instruction, on trouve des

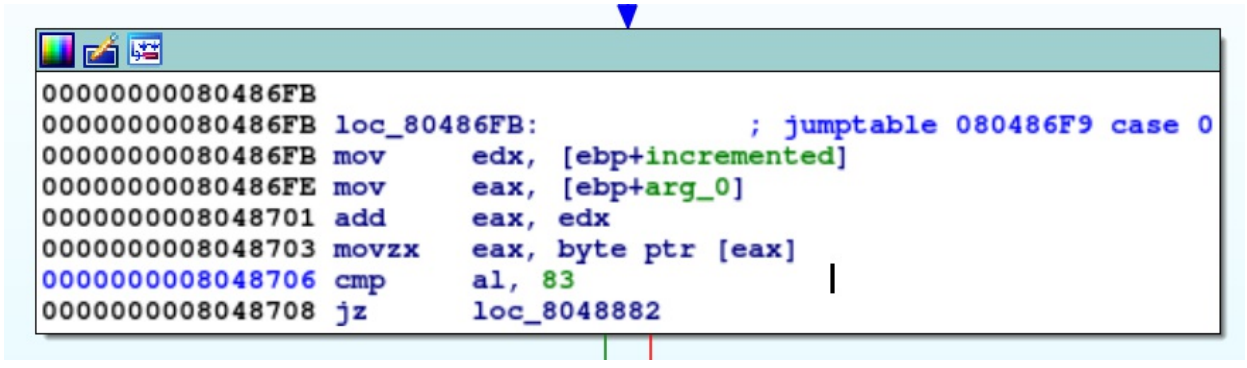
blocs représentant un switch, avec 7 possibilités. On en déduit que le nombre de caractères du mot à trouver est de 7. (On boucle 7 fois, de 0 à 6).



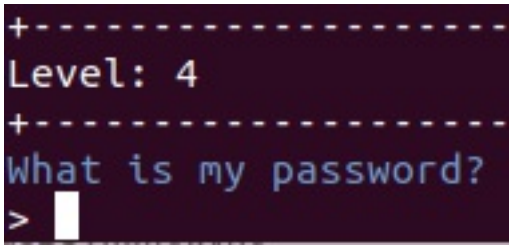
On se penche maintenant sur les 7 blocks du switch. Ce sont ces blocks qui déterminent si l'on continue à boucler, ou on s'arrête avec un message d'erreur.
L'instruction cmp s'effectue entre la valeur de al, qui est les 8 bits les plus bas de eax, et un chiffre (Ici 53h en hexadecimal pour le premier). En prenant la valeur en ascii de 83 (l'équivalent de 53 en décimal), on trouve une lettre correspondante (ici un S majuscule)



Lorsqu'on teste cette première lettre dans le programme, le message d'erreur change. On nous indique maintenant que l'on n'arrive pas à trouver la lettre à la position 1 (et non plus à la position 0). C'est donc la bonne technique. On convertissant les 6 autres nombres hexadécimaux des autres blocks du switch en ascii, on trouve le flag Smeagol.



Level 4



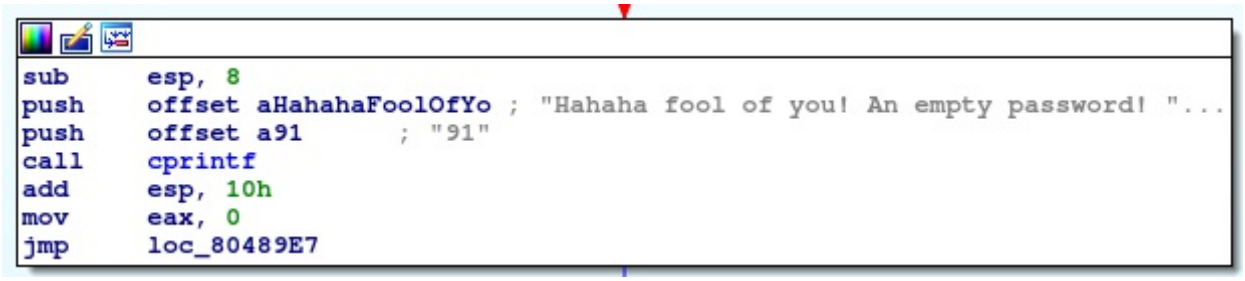
On peut apercevoir que le programme nous demande un mot de passe dans le quatrième niveau.

```
; Attributes: bp-based frame

public level_4
level_4 proc near

var_10= dword ptr -10h
var_C= dword ptr -0Ch
var_4= dword ptr -4
input_string= dword ptr  8

push    ebp
mov     ebp, esp
push    ebx
sub     esp, 14h
mov     [ebp+var_10], offset unk_8049440
sub     esp, 0Ch
push    [ebp+input_string]
call    _strlen
add     esp, 10h
lea     edx, [eax-1]
mov     eax, [ebp+input_string]
add     eax, edx        ; put the last character \n in eax
mov     byte ptr [eax], 0 ; replace the \n by a \0
mov     eax, [ebp+8]    ; put the final string in eax without the \n
movzx   eax, byte ptr [eax] ; replaces all the input by 0 except the first character
test    al, al          ; checks if the first byte is not 0
jnz     short input_not_empty ; jumps if the input is not empty
```

Le premier bloc traite l'entrée utilisateur pour supprimer le caractère "entrée", ou "\n".
On peut voir qu'une chaîne de caractères mystérieuse est aussi stockée en mémoire dans ebp+var_10.
IDA nous facilite la tâche grâce à la HEX view qui nous permet de voir les caractères stockés au format hexadécimal :

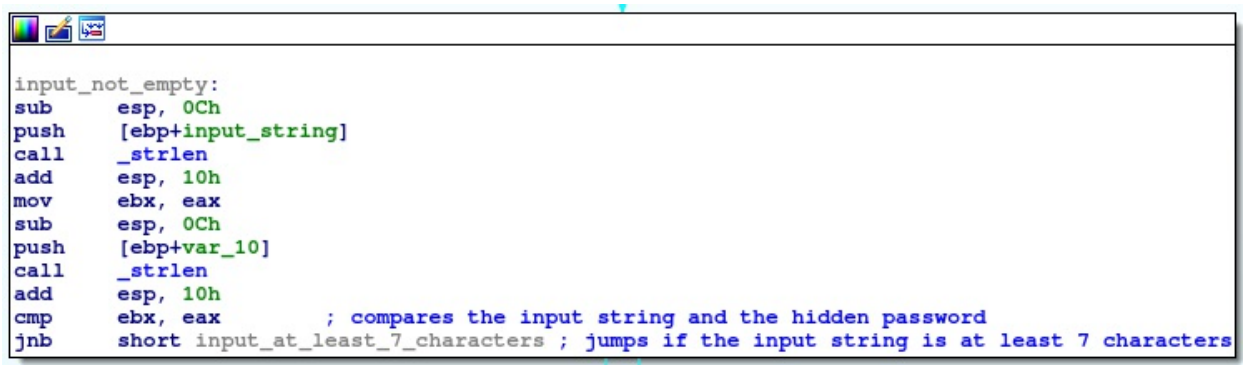
```
08049430  9A 87 04 08 CF 87 04 08
08049440  DF DE D9 CE DC D6 88 00
08049450  68 61 20 66 6F 6F 6C 20
```

Ensuite, le bloc regarde si des caractères sont présents dans l'entrée utilisateur.
Dans le cas contraire, le bloc suivant affichera un message d'erreur puis quittera la fonction level_4 :



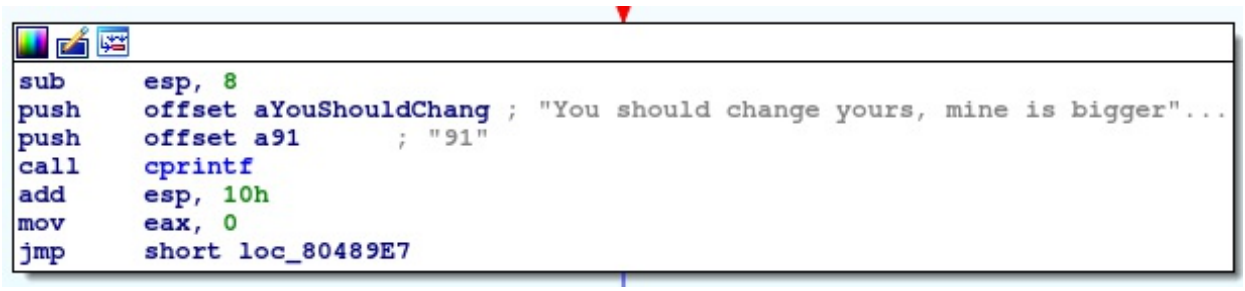
```
sub     esp, 8
push    offset aHahahaFoolOfYo ; "Hahaha fool of you! An empty password! "...
push    offset a91             ; "91"
call    cprintf
add     esp, 10h
mov     eax, 0
jmp     loc_80489E7
```

Si l'entrée utilisateur contient au moins 1 caractère, on arrive sur ce bloc :



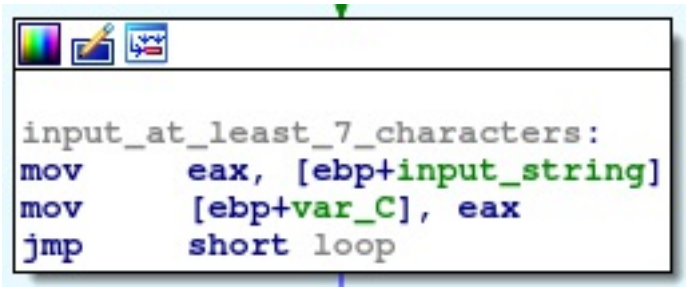
```
input_not_empty:
sub     esp, 0Ch
push    [ebp+input_string]
call    _strlen
add     esp, 10h
mov     ebx, eax
sub     esp, 0Ch
push    [ebp+var_10]
call    _strlen
add     esp, 10h
cmp     ebx, eax        ; compares the input string and the hidden password
jnb     short input_at_least_7_characters ; jumps if the input string is at least 7 characters
```

Ce bloc vérifie que l'entrée utilisateur fait au moins la taille du mot de passe "caché".
On stocke la longueur de l'entrée utilisateur dans ebx, puis la longueur du mot de passe "caché" contenu dans ebp+var_10 dans eax.
On peut voir que le mot de passe caché fait 7 caractères en débuggant avec gdb.
On compare ebx et eax puis on saute si l'entrée utilisateur fait au moins 7 caractères (jump not below).
Dans le cas contraire, le bloc suivant affichera un message d'erreur puis quittera la fonction level_4 :



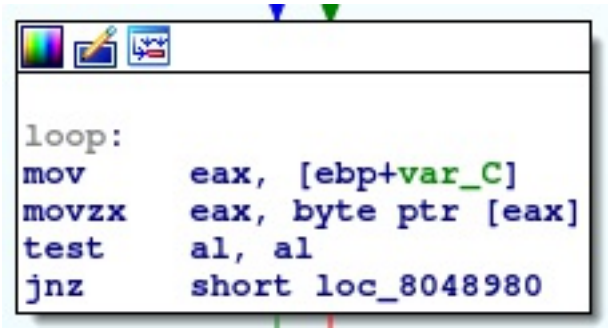
```
sub     esp, 8
push    offset aYouShouldChang ; "You should change yours, mine is bigger"...
push    offset a91             ; "91"
call    cprintf
add     esp, 10h
mov     eax, 0
jmp     short loc_80489E7
```

Si l'entrée utilisateur fait au moins 7 caractères, on arrive sur ce bloc qui déplace l'entrée utilisateur dans ebp+var_C :



```
input_at_least_7_characters:
mov     eax, [ebp+input_string]
mov     [ebp+var_C], eax
jmp     short loop
```

On saute directement au prochain bloc.

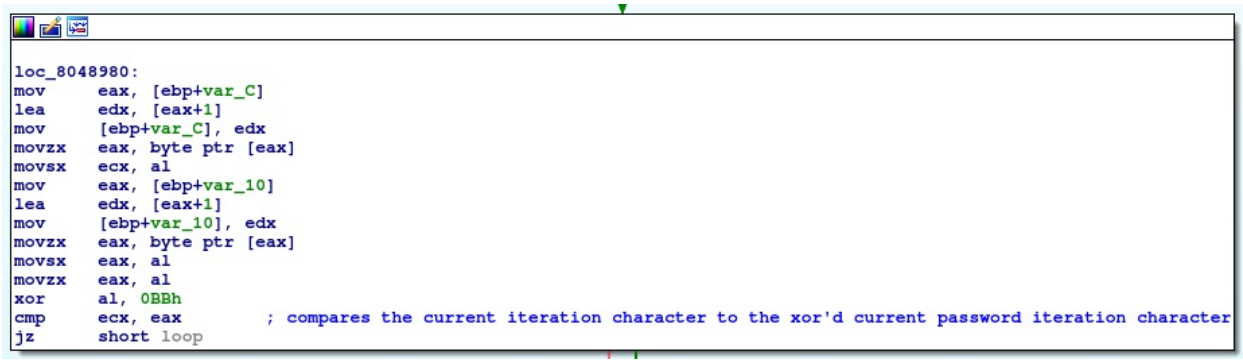


```
loop:
mov     eax, [ebp+var_C]
movzx   eax, byte ptr [eax]
test    al, al
jnz     short loc_8048980
```

On arrive à l'entrée d'une boucle.
C'est là que le code devient intéressant.
Ce bloc déplace la string contenue à l'adresse ebp+var_C dans eax puis regarde si son premier caractère n'est pas nul.
Si le caractère est nul, le bloc suivant affichera un message de succès (couleur verte) puis quittera la fonction level_4 :

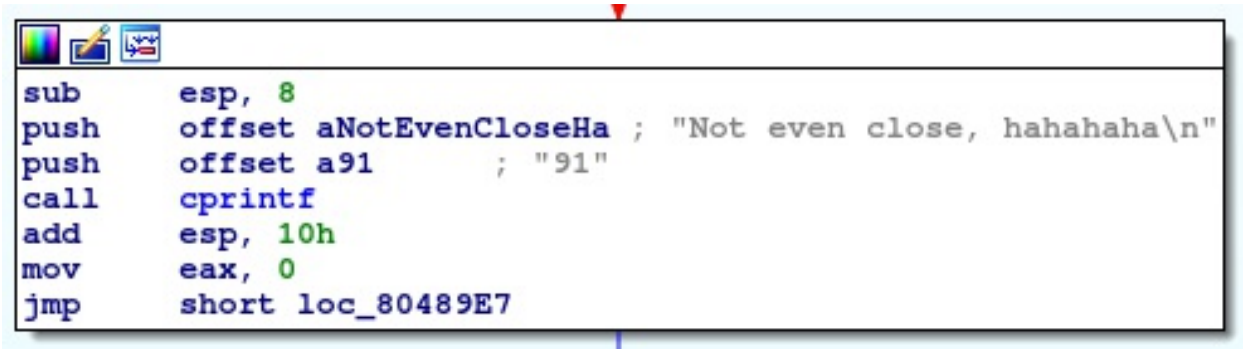
```
sub     esp, 8
push    offset aScrewYou ; "Screw you!\n"
push    offset a92       ; "92"
call    cprintf
add     esp, 10h
mov     eax, 1
```

Si le caractère n'est pas nul, la boucle continue.
Ce bloc stocke le caractère du mot de passe caché de l'itération courante dans ecx.
Il récupère aussi le caractère de l'entrée utilisateur de l'itération courante, puis effectue une opération XOR avec le nombre 187 dessus.
Enfin, la valeur est stockée dans eax, et plus précisément dans al (premier byte de eax).
Une fois ces deux caractères stockés, on les compare.



```
loc_8048980:
mov     eax, [ebp+var_C]
lea     edx, [eax+1]
mov     [ebp+var_C], edx
movzx   eax, byte ptr [eax]
movsx   ecx, al
mov     eax, [ebp+var_10]
lea     edx, [eax+1]
mov     [ebp+var_10], edx
movzx   eax, byte ptr [eax]
movsx   eax, al
movzx   eax, al
xor     al, 0BBh
cmp     ecx, eax        ; compares the current iteration character to the xor'd current password iteration character
jz      short loop
```

S'ils ne sont pas égaux (code ASCII différent), le bloc suivant affichera un message d'erreur puis quittera la fonction level_4 :



```
sub     esp, 8
push    offset aNotEvenCloseHa ; "Not even close, hahahaha\n"
push    offset a91      ; "91"
call    cprintf
add     esp, 10h
mov     eax, 0
jmp     short loop_80489E7
```

S'ils sont égaux (même code ASCII), on revient au début de la boucle.

On comprend maintenant que le mot de passe à trouver correspond à une chaîne de 7 caractères dont chaque caractère correspond au xor de 187 des caractères suivants :
0DF
0DE
0D9
0CE
0DC
0D6
88

On peut faire un simple script en Python qui effectue le calcul puis affiche le mot de passe :

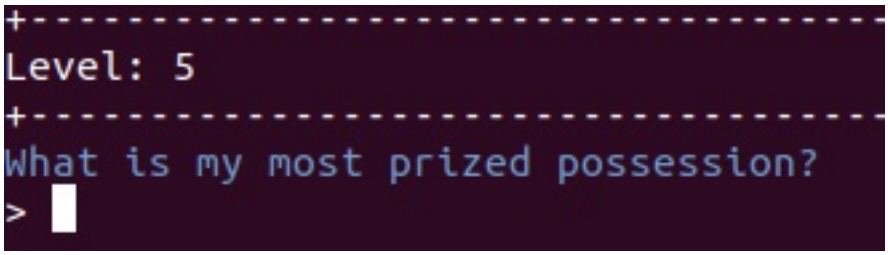
```python
#!/usr/bin/python
hidden_password = [0xDF, 0xDE, 0xD9, 0xCE, 0xDC, 0xD6, 0x88]

password = ''

for i in hidden_password:
    password += chr(i ^ 187)

print password
```

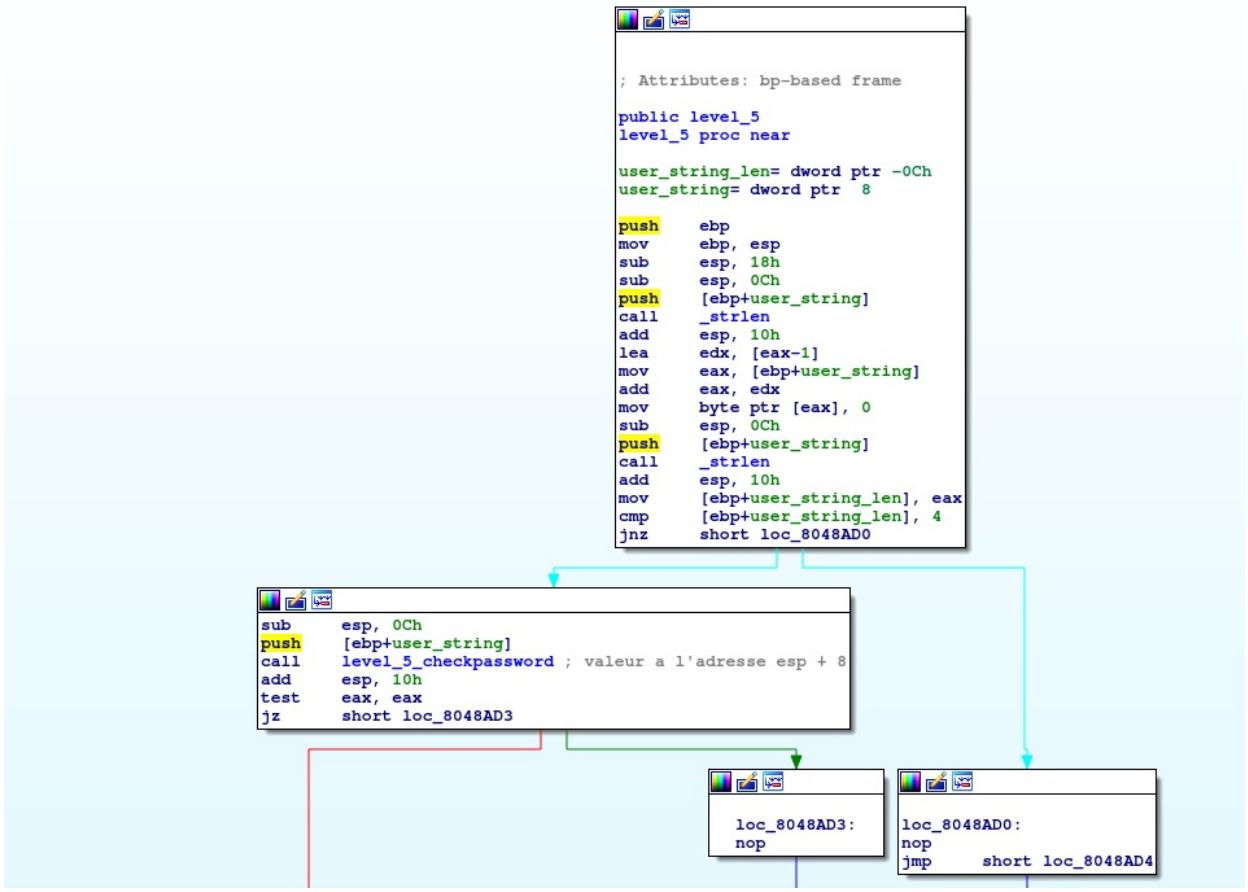
On trouve alors le mot de passe en clair : debugm3.

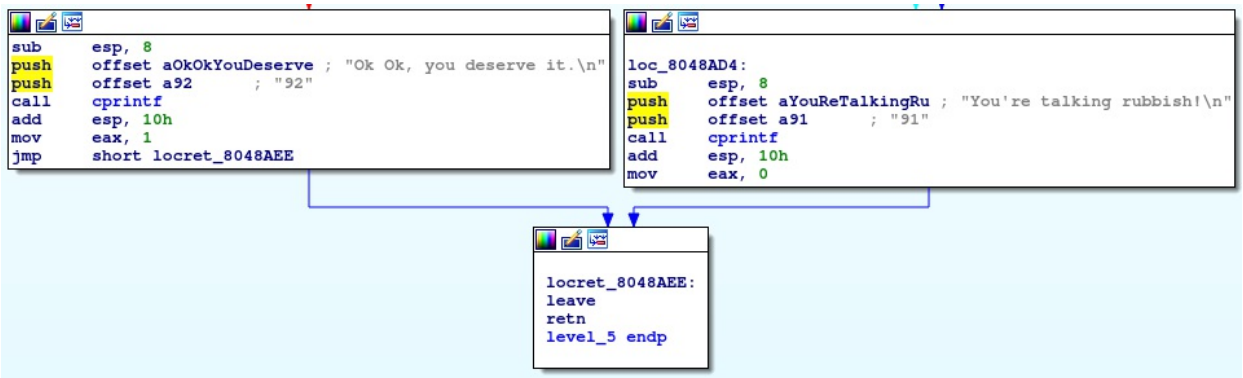
Level 5

Même sans connaître Lord of the RINGs nous pouvons déduire le flag, qui sera probablement 'RING' mais sûrement avec un leetspeak équivalent.

Si je décidais de bruteforce ce challenge étant donné la taille probable de la chaîne, ce serait un jeu d'enfant, mais continuons d'analyser le code sinon cela ne serait pas amusant.







Dans le premier bloc la chaîne entrée en paramètre est récupérée et le \n est remplacé par un \0 d'une part, et ensuite il y a une vérification de la taille de la chaîne entrée par l'utilisateur.

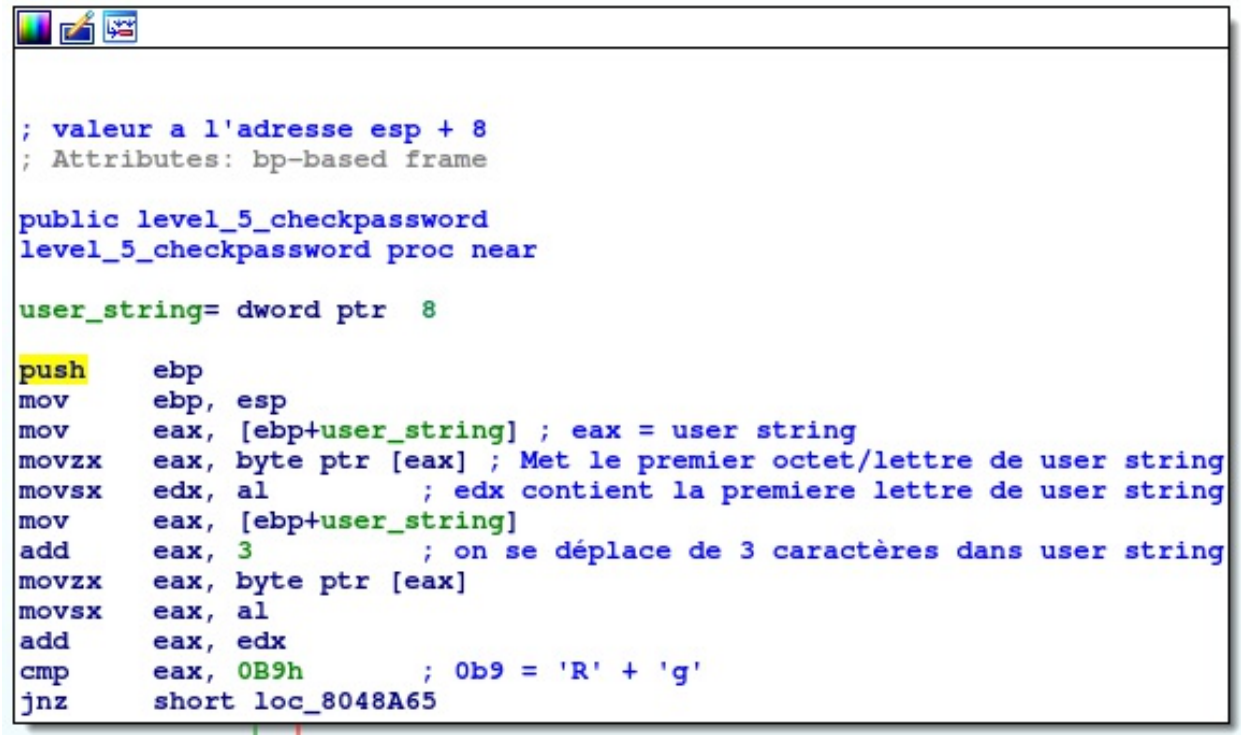
Cela valide notre hypothèse sur le flag 'RING'.

Dans le cas où la taille de la chaîne n'est pas de 4, on jump vers le print_fail_msg et comme son nom l'indique c'est le message de la défaite.

Dans le cas contraire, cela appelle la fonction level_5_checkpassword avec la saisie utilisateur en paramètre.

Ensuite, si la fonction renvoie une réponse différente de zéro, le programme va afficher le message de succès.
Puis nous passons au bloc de fin de fonction.

Cette partie maintenant analysée, nous allons nous pencher sur le contenu de la fonction level_5_checkpassword.

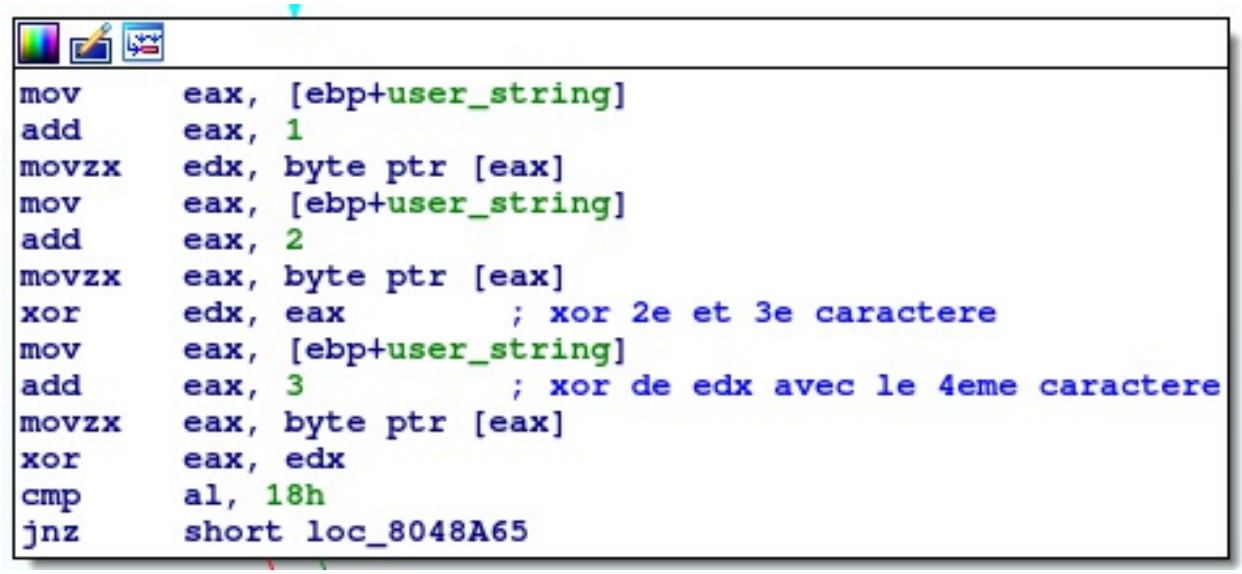


```
; valeur a l'adresse esp + 8
; Attributes: bp-based frame

public level_5_checkpassword
level_5_checkpassword proc near

user_string= dword ptr  8

push    ebp
mov     ebp, esp
mov     eax, [ebp+user_string] ; eax = user string
movzx   eax, byte ptr [eax] ; Met le premier octet/lettre de user string
movsx   edx, al             ; edx contient la premiere lettre de user string
mov     eax, [ebp+user_string]
add     eax, 3              ; on se déplace de 3 caractères dans user string
movzx   eax, byte ptr [eax]
movsx   eax, al
add     eax, edx
cmp     eax, 0B9h           ; 0b9 = 'R' + 'g'
jnz     short loc_8048A65
```

Nous pouvons résumer cette partie par l'équation ci-dessous :

$$c_0 + c_3 = 0B9h = 185$$

Si l'on rentre dans les détails, on récupère le premier et le dernier (4e) caractère, on les additionne est on les compare avec la valeur `0xB9`.



```
mov     eax, [ebp+user_string]
add     eax, 1
movzx   edx, byte ptr [eax]
mov     eax, [ebp+user_string]
add     eax, 2
movzx   eax, byte ptr [eax]
xor     edx, eax            ; xor 2e et 3e caractere
mov     eax, [ebp+user_string]
add     eax, 3              ; xor de edx avec le 4eme caractere
movzx   eax, byte ptr [eax]
xor     eax, edx
cmp     al, 18h
jnz     short loc_8048A65
```

Dans ce second bloc, on récupère les trois derniers caractères et on applique une opération arithmétique de type `XOR`. Enfin, le résultat est comparé avec `0x18`.

Nous obtenons donc l'équation suivante :

$$(c_1 \oplus c_2) \oplus c_3 = 018h = 24$$

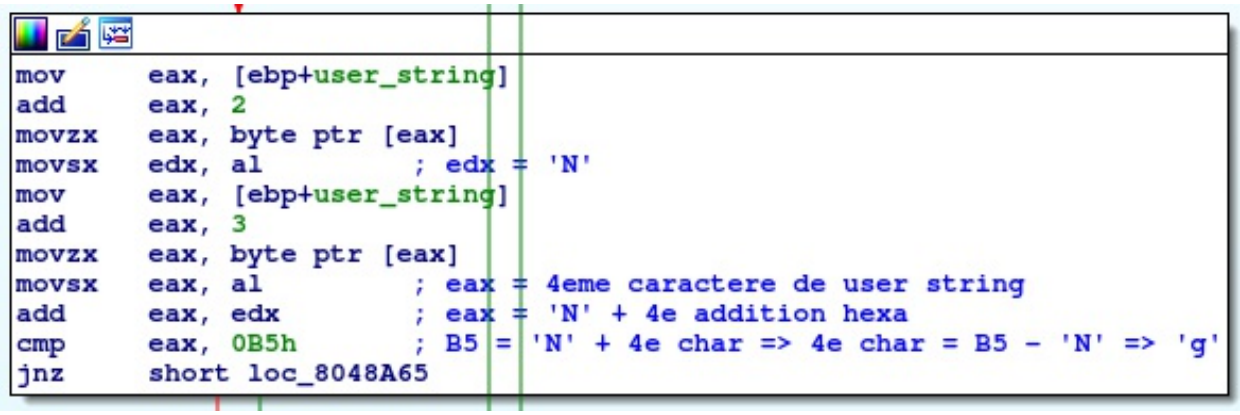
Ce bloc, simple mais essentiel à la résolution de ce challenge, nous permet de voir que le 3ème caractère est comparé avec la valeur `0x4E` qui vaut 'N'

```
mov     eax, [ebp+user_string]
add     eax, 2
movzx   eax, byte ptr [eax]
cmp     al, 'N'             ; verifie si le 3e caractere est 'N'
jnz     short loc_8048A65
```

Ci-dessous l'équation associée :

$$c_2 = 4Eh = 78 = N$$

Ce dernier bloc de code compare la valeur `0xB5` avec la somme des deux derniers caractères.



```
mov     eax, [ebp+user_string]
add     eax, 2
movzx   eax, byte ptr [eax]
movsx   edx, al             ; edx = 'N'
mov     eax, [ebp+user_string]
add     eax, 3
movzx   eax, byte ptr [eax]
movsx   eax, al             ; eax = 4eme caractere de user string
add     eax, edx            ; eax = 'N' + 4e addition hexa
cmp     eax, 0B5h           ; B5 = 'N' + 4e char => 4e char = B5 - 'N' => 'g'
jnz     short loc_8048A65
```

L'équation correspondante :

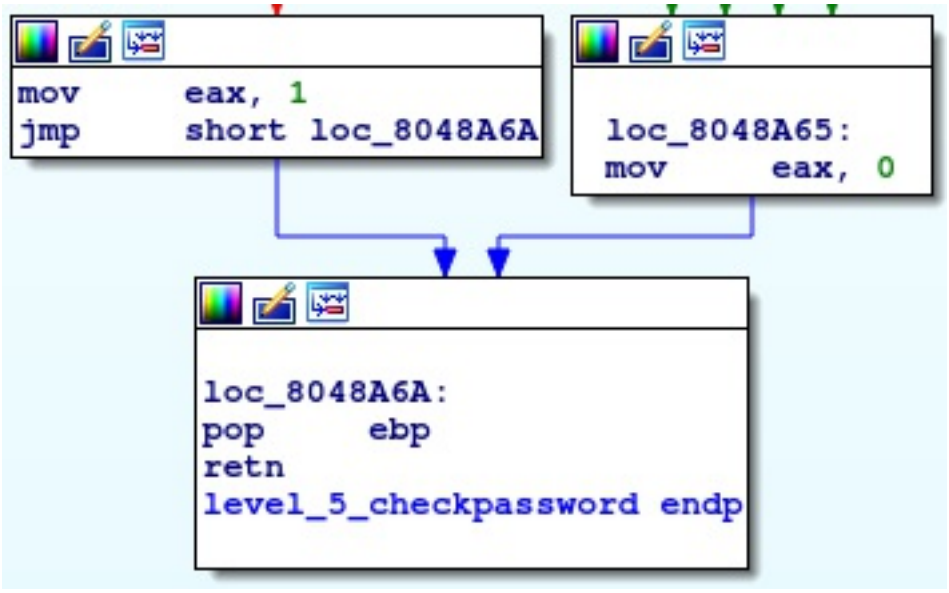
$$c_2 + c_3 = 0B5h = 181$$

Pour terminer nous apercevons les deux blocs possibles avant d'arriver au bloc de fin de fonction.

Celui à gauche est atteint si les quatre blocs du dessus sont validés sinon nous arrivons dans le bloc de droite.

Nous pouvons voir que la valeur de retour est 1 si les conditions sont validées (0 sinon) ce qui

valide notre analyse précédente de la fonction level_5.



Afin de trouver le flag nous devons résoudre le système d'équations ci-dessous :

$$\begin{cases} c_2 & = 04Eh = N \\ c_2 + c_3 & = 0B5h \\ c_0 + c_3 & = 0B9h \\ (c_1 \oplus c_2) \oplus c_3 & = 018h \end{cases}$$

Ayant déjà le troisième caractère en clair ('N'), nous allons commencer par résoudre les équations avec celui-ci.

$$\Leftrightarrow \begin{cases} c_2 & = 04Eh = N \\ c_3 & = 0B5h - c_2 = g \\ c_0 + c_3 & = 0B9h \\ (c_1 \oplus c_2) \oplus c_3 & = 018h \end{cases}$$

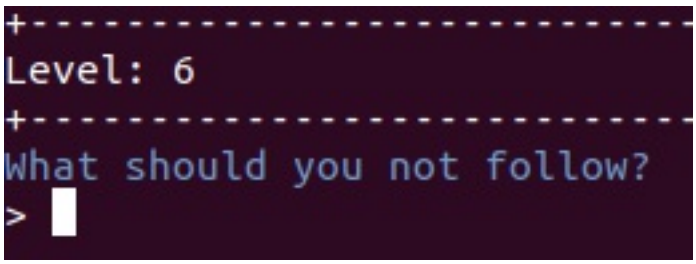
$$\Leftrightarrow \begin{cases} c_2 & = 04Eh = N \\ c_3 & = 0B5h - c_2 = g \\ c_0 & = 0B9h - c_3 = R \\ (c_1 \oplus c_2) \oplus c_3 & = 018h \end{cases}$$

$$\Leftrightarrow \begin{cases} c_2 & = 04Eh = N \\ c_3 & = 0B5h - c_2 = g \\ c_0 & = 0B9h - c_3 = R \\ (c_1 \oplus c_2) & = 018h \oplus c_3 = 07Fh \end{cases}$$

$$\Leftrightarrow \begin{cases} c_2 & = 04Eh = N \\ c_3 & = 0B5h - c_2 = g \\ c_0 & = 0B9h - c_3 = R \\ c_1 & = 07Fh \oplus c_2 = 1 \end{cases}$$

On trouve donc le flag : R1Ng.

Level 6



Admettons tout d'abord que nous ne connaissons pas Lord of the Rings sinon l'exercice sera moins drôle.

```
push    ebp
mov     ebp, esp
push    edi
sub     esp, 254h
sub     esp, 0Ch
push    [ebp+user_string]
call    _strlen
add     esp, 10h
lea     edx, [eax-1]
mov     eax, [ebp+user_string]
add     eax, edx
mov     byte ptr [eax], 0 ; replace \n to \0
```

Tout d'abord ce morceau permet de remplacer le retour chariot (\n) par le caractère fin de chaine appelé null (\0).

```
sub     esp, 0Ch
push    [ebp+user_string]
call    _strlen
add     esp, 10h
mov     [ebp+user_string_len], eax ; récupère la taille de la chaine
```

Ensuite nous récupérons la taille réelle (sans retour chariot) et nous la mettons dans la variable user_string_len.

```
mov     [ebp+cmpt], 0
lea     edx, [ebp+var_D4] ; edx <- adresse of var_d4
mov     eax, 0
mov     ecx, 48
mov     edi, edx
rep stosd               ; memset operation : memset(edx, 0, 48)
mov     [ebp+var_CC], 1
mov     [ebp+var_AE], 1
mov     [ebp+var_88], 1
mov     [ebp+var_6C], 1
mov     [ebp+var_4E], 1
mov     [ebp+var_26], 1
lea     edx, [ebp+var_254]
mov     eax, 0
mov     ecx, 60h
mov     edi, edx
rep stosd
mov     [ebp+var_224], 1
mov     [ebp+var_210], 1
mov     [ebp+var_1B8], 1
mov     [ebp+var_174], 1
mov     [ebp+var_138], 1
mov     [ebp+var_108], 1
cmp     [ebp+user_string_len], 6
jz      short len_equals_6
```

Cette partie du code peut paraitre au premier abord compliquée mais nous allons voir que la compréhension de cette partie est essentielle dans la résolution de l'exercice.

En premier lieu, une opération `stosd` : cette opération peut s'apparenter à la fonction `memset` en C.

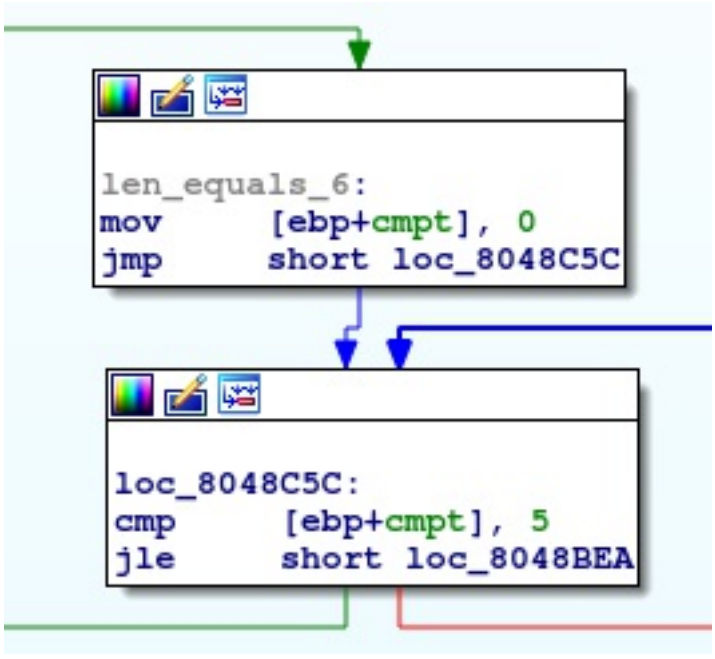
En prenant les paramètres edi, ecx et edx en compte nous avons `memset(var_D4, 0, 48)` pour le premier buffer (partant de var_D4).
Il en est de même pour le second buffer partant de var_254 `memset(var_254, 0, 96)`
Dans les deux cas nous avons après chaque opération stosd une suite de 6 affectations pour chaque buffer.

Pour faire simple nous initialisons deux buffers à 0, l'un de taille 48 et l'autre de taille 96, puis nous mettons 12 valeurs à 1, 6 par buffer.

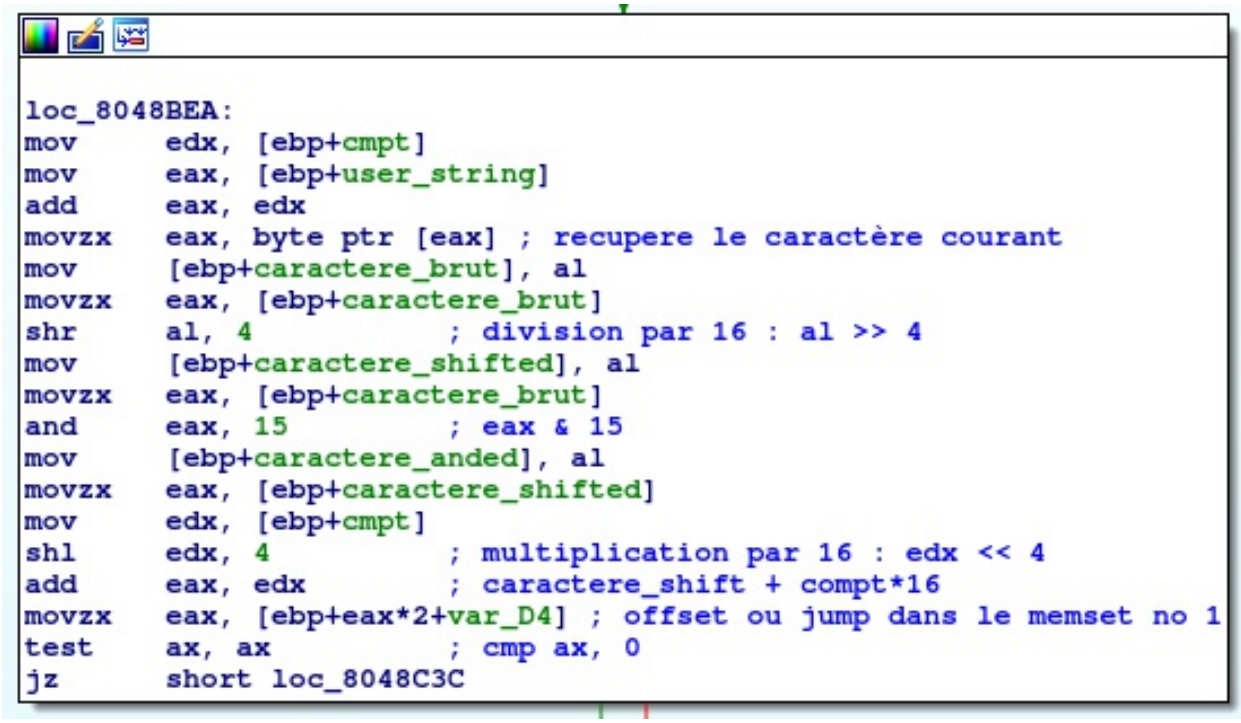
Une vérification de la taille de la chaine saisie par l'utilisateur a lieu en dernier.

Premier indice : la taille de la chaine est de 6 caractères.

Dans le contexte actuel, nous avons `256^6` possibilités. Avec les processeurs actuels nous pouvons bruteforcer facilement ce challenge en un temps restreint. Mais pour le plaisir nous allons continuer notre analyse.



Ensuite, nous pouvons voir que nous entrons dans une boucle, en paramètre un compteur et une condition de sortie qui est la taille de la chaine.



```
loc_8048BEA:
mov     edx, [ebp+cmpt]
mov     eax, [ebp+user_string]
add     eax, edx
movzx   eax, byte ptr [eax] ; recupere le caractère courant
mov     [ebp+caractere_brut], al
movzx   eax, [ebp+caractere_brut]
shr     al, 4           ; division par 16 : al >> 4
mov     [ebp+caractere_shifted], al
movzx   eax, [ebp+caractere_brut]
and     eax, 15         ; eax & 15
mov     [ebp+caractere_anded], al
movzx   eax, [ebp+caractere_shifted]
mov     edx, [ebp+cmpt]
shl     edx, 4          ; multiplication par 16 : edx << 4
add     eax, edx        ; caractere_shift + compt*16
movzx   eax, [ebp+eax*2+var_D4] ; offset ou jump dans le memset no 1
test    ax, ax          ; cmp ax, 0
jz      short loc_8048C3C
```

Dans ce bloc nous parcourons la chaine de caractères en se déplaçant à l'aide du compteur afin de le mettre dans une variable que nous appellerons ici caractere_brut.

Ensuite nous appliquons une opération de shift right de 4 sur le caractere_brut ce qui revient à diviser par 16 la valeur décimale correspondant au caractère dans la table ASCII.

Cette valeur est stockée dans une variable que nous appellerons caractere_shifted.

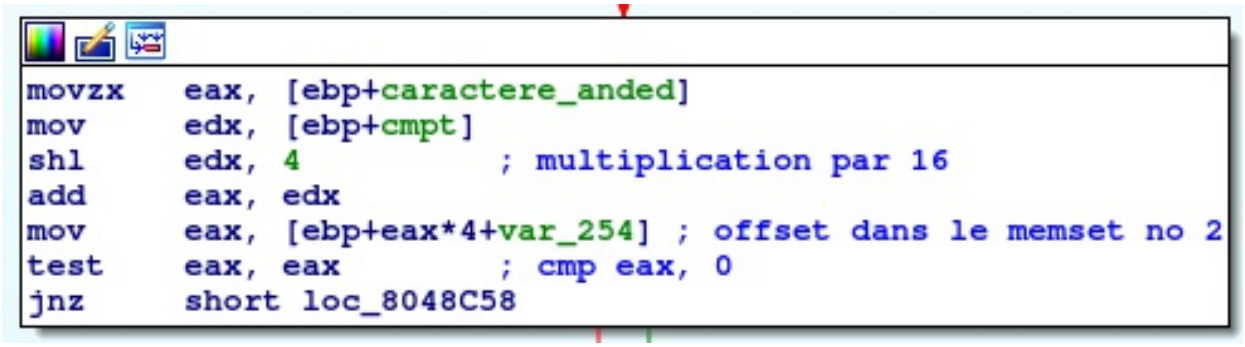
Ensuite nous reprenons le caractere_brut afin d'y appliquer une opération ET logique avec la valeur 15 et on la stocke dans la variable caractere_anded.

Puis nous récupérons le caractere_shifted dans eax et le compteur dans edx que nous multiplions par 16 suite à une opération de shift left par 4.

Après cela nous effectuons la somme (caractere_shifted + cmp * 4).

Ensuite, nous allons chercher dans le buffer 1 (var_D4 memset) à l'adresse [ebp+eax*2+var_4] afin de vérifier si nous tombons sur un 0 ou un 1, si la valeur est un 1 cela veut dire que le caractère courant passe la première condition, dans le cas contraire ou l'on tombe sur un zero on jump vers le message d'erreur.

En résumé, pour passer la condition de ce bloc il faut que le caractère courant passe la condition suivante :
var_D4[(char/16 + cmpt*16) * 2] == 1

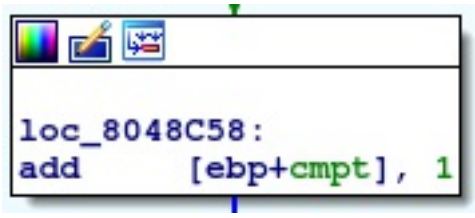


```
movzx   eax, [ebp+caractere_anded]
mov     edx, [ebp+cmpt]
shl     edx, 4          ; multiplication par 16
add     eax, edx
mov     eax, [ebp+eax*4+var_254] ; offset dans le memset no 2
test    eax, eax        ; cmp eax, 0
jnz     short loc_8048C58
```

Dans ce bloc nous effectuons un second test sur la chaine entrée par l'utilisateur. Ce test n'est pas effectué si l'on ne passe par la condition précédente.

Le second test est similaire au premier sauf que cette fois on vérifie dans le second buffer en utilisant caracter_anded.

En résumé, pour passer la condition de ce bloc il faut que le caractère courant passe la condition suivante :
var_254[()(char & 15) + (cmpt*16)) * 4] == 1



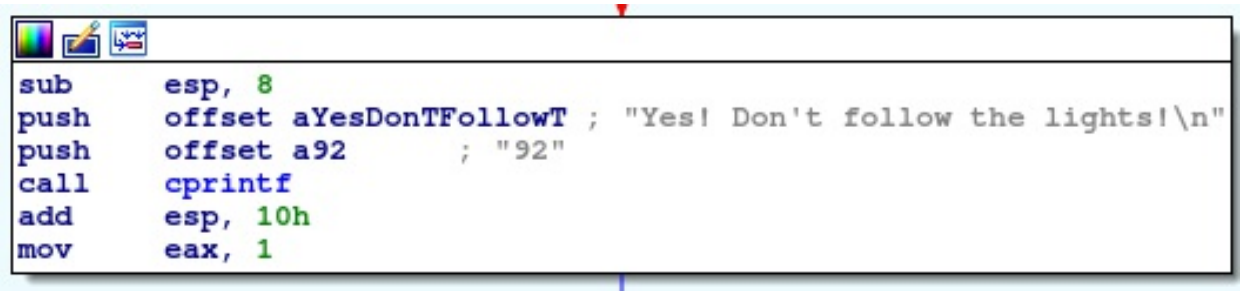
```
loc_8048C58:
add     [ebp+cmpt], 1
```

Une fois le second test passé, on incrémente le compteur afin de vérifier le caractère suivant.

A ce stade afin de récupérer le flag nous pourrions utiliser le script suivant :

```python
#!/usr/bin/python
adresses_buffer1 = [0xD4, 0xCC, 0xAE, 0x88, 0x6C, 0x4E, 0x26]
adresses_buffer2 = [0x254, 0x224, 0x210, 0x1B8, 0x174, 0x138, 0x108]

buf1_target = []
buf2_target = []

for i in range(1, len(adresses_buffer1)):
    buf1_target.append(adresses_buffer1[0]-adresses_buffer1[i])
    buf2_target.append(adresses_buffer2[0]-adresses_buffer2[i])

res = "Flag is : "
for i in range (0,6):
    for c in range(0,127):
        if ((c/16) + (i*16))*2 == buf1_target[i]:
            if ((c & 15) + (i*16))*4 == buf2_target[i]:
                res += str(chr(c))
```

```
print res
```

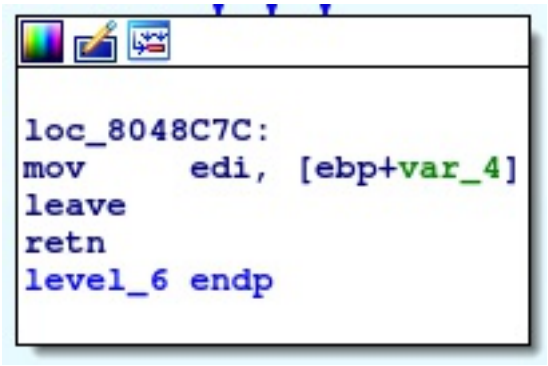


```
sub     esp, 8
push    offset aYesDonTFollowT ; "Yes! Don't follow the lights!\n"
push    offset a92             ; "92"
call    cprintf
add     esp, 10h
mov     eax, 1
```

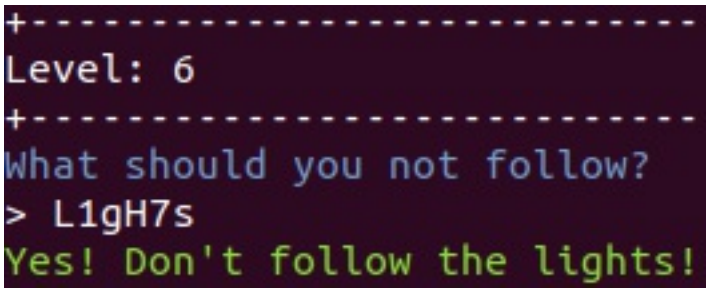
Une fois que les 6 caractères passent les deux conditions nous sortons de la boucle pour arriver au bloc affichant le message de succès et aussi la "réponse" du moins un gros indice permettant de déterminer le flag.

En prenant en compte cet indice nous pouvons améliorer notre script et passer à celui ci-dessous :

```
#!/usr/bin/python
adresses_buffer1 = [0xD4, 0xCC, 0xAE, 0x88, 0x6C, 0x4E, 0x26]
adresses_buffer2 = [0x254, 0x224, 0x210, 0x1B8, 0x174, 0x138, 0x108]

buf1_target = []
buf2_target = []

chars = [
    ['L', 'l'],
    ['I', 'i', '1'],
    ['G', 'g', '6', '9'],
    ['H', 'h'],
    ['T', 't', '7'],
    ['S', 's', '5']
]

for i in range(1, len(adresses_buffer1)):
    buf1_target.append(adresses_buffer1[0]-adresses_buffer1[i])
    buf2_target.append(adresses_buffer2[0]-adresses_buffer2[i])

res = "Flag is : "

for i in range (0,6):
    for c in chars[i]:
        if ((ord(c)/16) + (i*16))*2 == buf1_target[i]:
            if ((ord(c) & 15) + (i*16))*4 == buf2_target[i]:
                res += str(chr(ord(c)))

print res
```



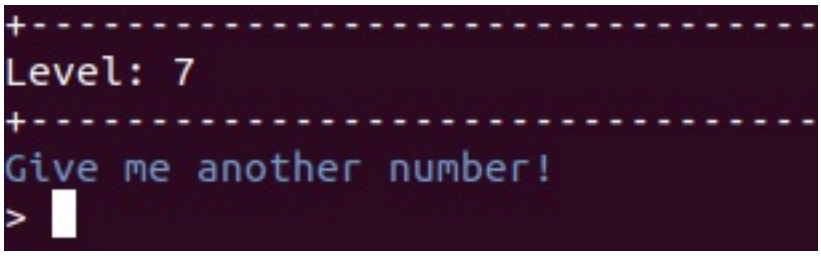
Dans tous les cas, pour terminer, nous passons par ce bloc qui est la fin de la fonction.



Une fois le script lancé, nous trouvons le flag : L1gH7s.

Level 7

Le niveau 7 nous demande d'entrer un nombre.



Contrairement aux autres niveaux, il n'est pas possible de visualiser le diagramme de blocs avec IDA.
Ainsi, nous allons suivre les instructions directement dans le mode linéaire de IDA.

```
                public level_7
level_7:                                ; CODE XREF: challenge+2F3↓p
                push    ebp
                mov     ebp, esp
                sub     esp, 18h
                sub     esp, 0Ch
                push    dword ptr [ebp+8] ; pushes the user_input

debut:                                  ; CODE XREF: .text:08048CA8↓j
                call    _atoi           ; converts the user_input to an int
                add     esp, 10h
                mov     [ebp-0Ch], eax  ; stores the user_input int
                push    eax
                xor     eax, eax        ; eax will be 0 in every case
                jz      short near ptr loc_8048CAC+1 ; always jumps since eax equals 0
                call    near ptr 684987F5h ; unreacheable instruction
                mov     word ptr [esi], ss ; unreacheable instruction
                icebp                   ; unreacheable instruction
                or      bh, [ebx-35h]   ; unreacheable instruction
                jns     short debut     ; unreacheable instruction
                jb      short near ptr loc_8048CB1+1 ; unreacheable instruction
```

Le premier bloc récupère l'entrée utilisateur et la stocke en mémoire.

On passe ensuite au second bloc "début" qui appelle la fonction atoi pour convertir l'entrée utilisateur en un int.
On récupère le résultat de eax pour le stocker à l'adresse ebp-0Ch.
Le résultat est aussi poussé dans la pile, mais un xor sur eax avec lui-même est effectué ce qui a comme conséquence de mettre la valeur 0 dans eax (quel que soit sa valeur précédente car un xor d'un nombre avec lui-même renverra toujours 0).

On peut voir qu'un jump if zero suit ce xor.
Nous avons déduit que eax aurait toujours la valeur 0 après le xor, on peut donc en déduire que toutes les instructions du bloc suivant ce jump if zero ne seront jamais atteintes.

On saute donc à l'étiquette loc_8048CAC+1 soit loc_8048CAD.
IDA n'affiche pas le bloc à cette adresse, mais on peut voir dans gdb que l'instruction effectuée à cette adresse est un pop de eax.
La dernière valeur poussée dans la stack étant l'entrée utilisateur au format int, on insérera celle-ci dans le registre eax.

```
loc_8048CAC:                            ; CODE XREF: .text:08048C9B↑j
                adc     al, 88          ; unreacheable instruction
```

Le prochain bloc d'une seule ligne n'est jamais atteint.

```
loc_8048CAE:                            ; CODE XREF: .text:08048CCB↓j
                mov     eax, [ebp-0Ch]
```

Ensuite, on récupère l'entrée utilisateur au format int en mémoire et on la place dans eax.

```
loc_8048CB1:                            ; CODE XREF: .text:08048CAA↑j
                sar     eax, 3          ; eax = eax >> 3
                cmp     eax, 4919
                jnz     short fail
                push    eax
                xor     eax, eax        ; eax will be 0 in every case
                jz      short near ptr loc_8048CCF+1 ; always jumps since eax equals 0
                call    near ptr 68498818h ; unreacheable instruction
                mov     word ptr [esi], ss ; unreacheable instruction
                icebp                   ; unreacheable instruction
                or      bh, [ebx-35h]   ; unreacheable instruction
                jns     short near ptr loc_8048CAE+2 ; unreacheable instruction
                jb      short near ptr win+1 ; unreacheable instruction
```

Le bloc suivant va effectuer l'opération shift arithmetic right qui décale de n bits vers la droite la valeur du registre passé en paramètre.
Par exemple, si eax contient le nombre 255, ou 1111 1111 en binaire, l'opération "sar eax, 3" décalera les bits 3 fois vers la droite.
Le résultat est alors le nombre 31, ou 0001 1111 en binaire. Le résultat écrase la précédente valeur du registre, ici eax.
On compare ensuite eax à 4919.
Si c'est égal, le bloc suivant affichera un message de succès (couleur verte) puis quittera la fonction level_7 :

```
win:                                    ; CODE XREF: .text:08048CCD↑j
                push    offset aErfOkThatWasTo ; "Erf, ok, that was too easy!\n"
                push    offset a92      ; "92"
                call    cprintf
                add     esp, 10h
                mov     eax, 1
                jmp     short locret_8048D07
```

Sinon, un autre xor est fait pour passer eax à 0 puis un jump if zero est effectué (dans tous les cas car eax sera toujours à 0).
On peut déduire de ce saut que toutes les autres instructions du bloc ne sont pas atteignables.
On saute alors au bloc suivant qui affichera un message d'erreur (couleur rouge) puis quittera la fonction level_7 :

```
fail:                                   ; CODE XREF: .text:08048CB9↑j
                sub     esp, 8
                push    offset aYesAlmostHahah ; "Yes! Almost! ... \nHahah not at all act"...
                push    offset a91      ; "91"
                call    cprintf
                add     esp, 10h
                mov     eax, 0
```

On peut aussi noter qu'un bloc inatteignable est aussi présent :

```
loc_8048CCF:                            ; CODE XREF: .text:08048CBE↑j
                adc     al, 88          ; unreacheable instruction
                sub     esp, 8          ; unreacheable instruction
```

Suite à ces observations, on peut en déduire que le nombre demandé doit être égal à 4919 après avoir été shifté de 3 bits vers la droite.
On peut faire un simple script en Python qui commence avec le nombre 4919 shifté de 3 bits vers la gauche.
Le script incrémentera de 1 le nombre jusqu'à ce que le 4ème bit en partant de la droite soit changé (cette valeur sera exclue).

```
#!/usr/bin/python
target = 4919
binary_target_start = bin(4919 << 3)

done = False
current_target = int(binary_target_start, 2)
flags = [current_target]

while not done:
    flags.append(current_target + 1)
    current_target += 1
    # checks if the next binary number 4th bit changes
    if bin(current_target + 1)[2:][-4] == str(0):
        done = True

print "Valid flags are:"
for flag in flags:
    print flag
```

On trouve alors les flags suivants :
39352 39353 39354 39355 39356 39357 39358 39359