

Tuto reversing

Comment enlever une protection anti-debug classique sur un executable ?

By Warr

INTRODUCTION

Je me décide à faire ce tuto très simple, mais néanmoins utile, car il montre comment se débarrasser d'une protection assez énervante. Il s'agit donc pour moi de vous montrer comment repérer et annuler une protection anti-debug contenue dans un exécutable. Nous utiliserons ici un exécutable lambda, le debugger Ollydbg et l'éditeur hexadécimal WinHex. Ce tuto peut également être assimilé à une initiation au reversing, car il montre les techniques assez souvent employées pour « cracker » un logiciel.

I/ Reconnaître une protection anti-debug

Cette étape la n'est pas la plus difficile :p. Nous ouvrons notre exécutable avec Ollydbg, puis nous appuyons sur F9 pour lancer (run) le programme. Voici ce qu'on obtient :

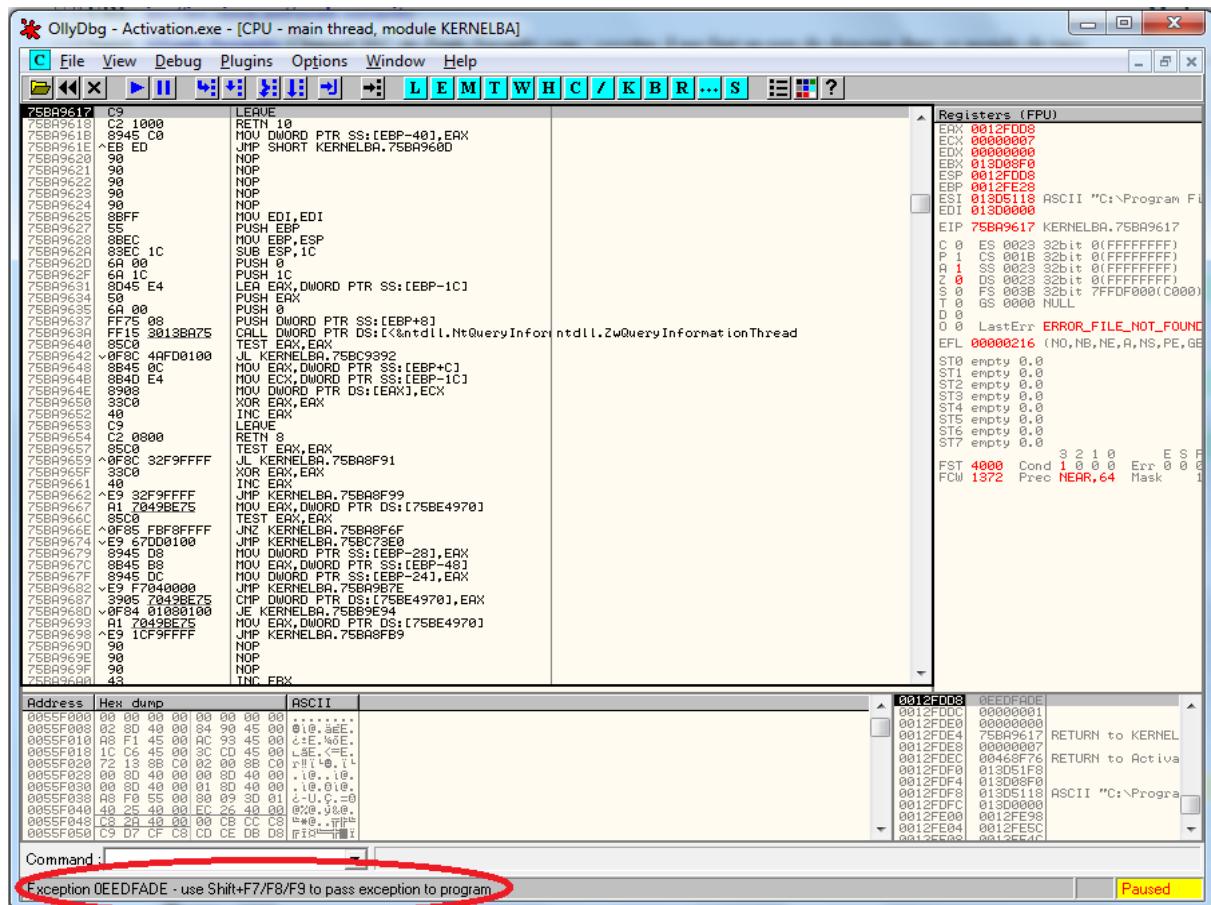


Figure 1 - Lancement du programme

La zone entourée en rouge nous indique que le programme a levé une exception (il est possible que cela soit dû au fait que nous l'avons lancé dans un debugger). On va tenter de la passer en appuyant sur Shift-F9. Si le même message persiste, on continue à appuyer sur Shift-f9. Une fois toutes les exceptions passées on tombe sur ceci.

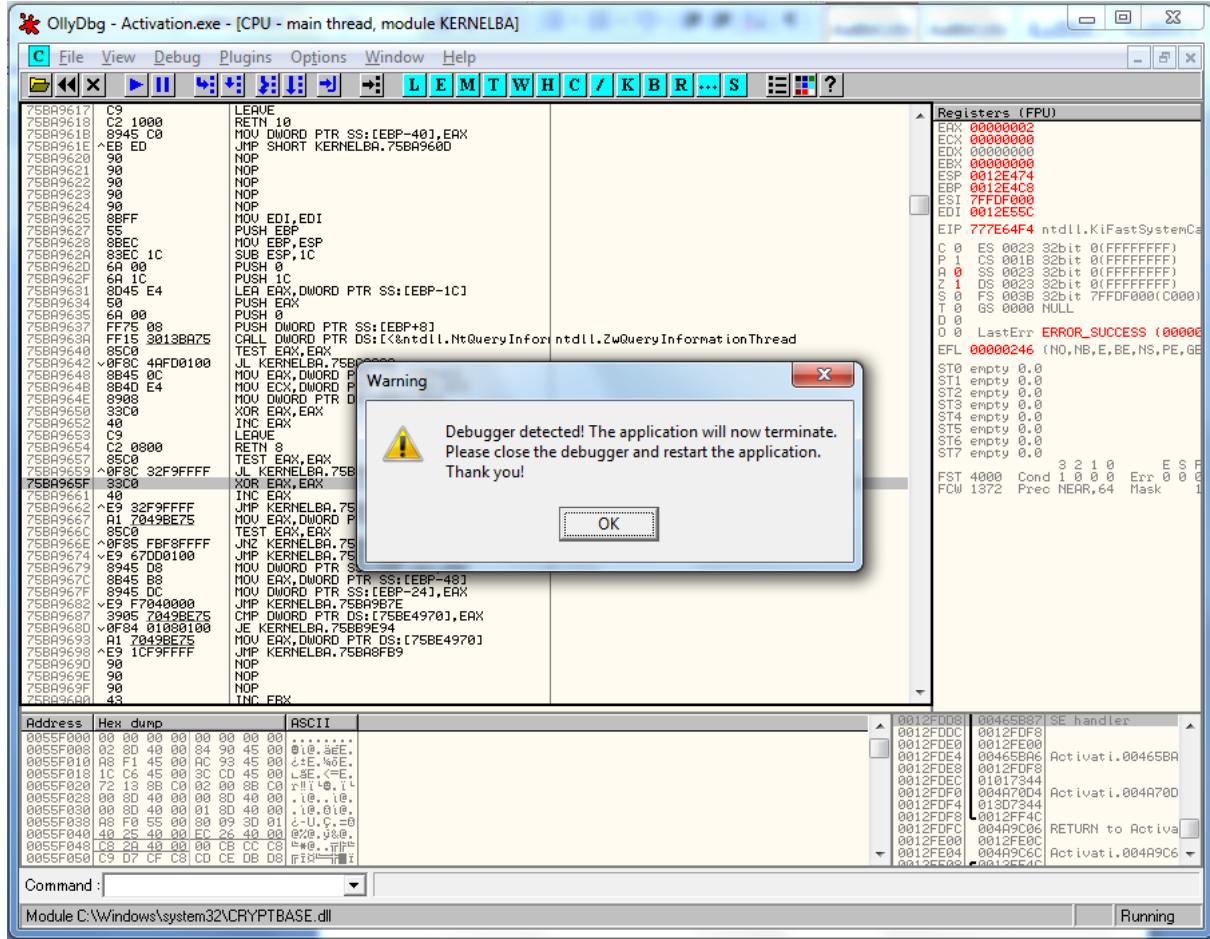


Figure 2 - Message d'erreur

Cet exécutable est donc bien protégé pas un debugger. Retenons bien le message de la boite de dialogue, il sera utile pour trouver la portion de code responsable de son affichage, et d'une manière générale responsable de la sécurité.

II/ Repérer l'emplacement de la sécurité dans le listing du code du programme

Pour faire sauter cette protection, il va falloir trouver le morceau de code (la routine) qui l'appelle. Pour cela, on va se servir du message qu'affiche la boîte de dialogue de la figure 2. En effet, la plupart des messages (pas tous malheureusement) utilisés dans les programmes sont visibles « en clair » dans leur code binaire. On va donc lancer une recherche sur le mot clé « debugger » pour trouver où se situe la protection.

Pour cela, rechargez le programme avec Ctrl-f2, puis clic droit au beau milieu du code -> search for -> all referenced strings.

Une nouvelle fenêtre s'affiche, contenant toutes les chaînes lisibles utilisées dans le programme. Dans cette nouvelle fenêtre clic droit -> search for text. Décochez la case « case sensitive » et cochez la case « entire scope ». Rentrez ensuite le mot « debugger » et lancez la recherche.

Une ligne se met en surbrillance. Double cliquez dessus pour qu'Olly nous emmène à l'endroit où cette chaîne est utilisée. Nous arrivons sur la portion de code qui nous intéresse, nous avons donc trouvé ce que nous recherchions.

III/ Désactiver la protection

Voilà donc le code vers lequel Olly nous a redirigés :

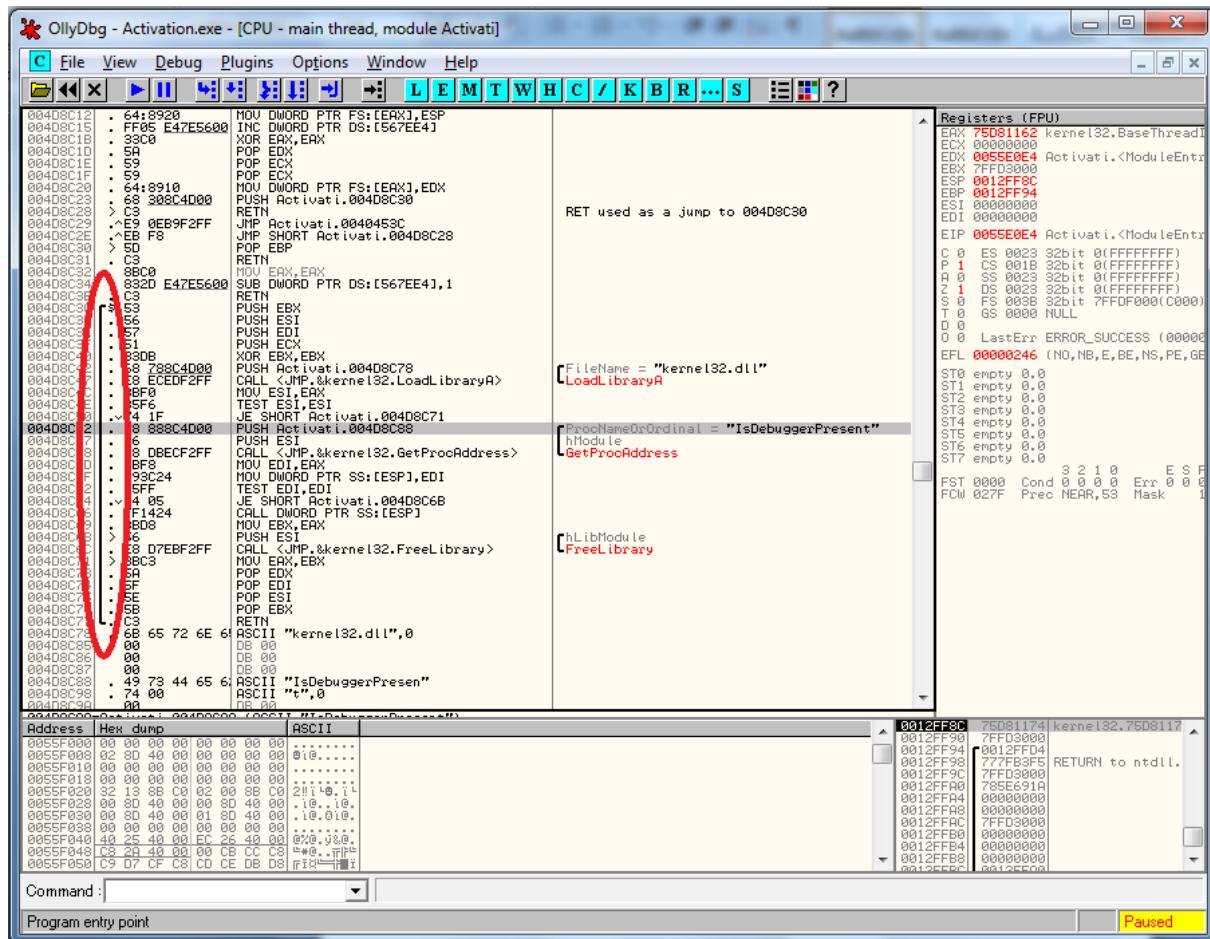


Figure 3

Grâce à l'accolade (entourée en rouge) nous voyons que notre chaîne est utilisée dans une fonction (un call) dont la première instruction est : 004D8C3C | \$ 53 | PUSH EBX

Ce call est certainement la protection que nous cherchons à faire sauter. Nous allons donc chercher à savoir où et quand il est appelé. Placez vous sur l'instruction citée ci-dessus puis regardez ci-dessous :

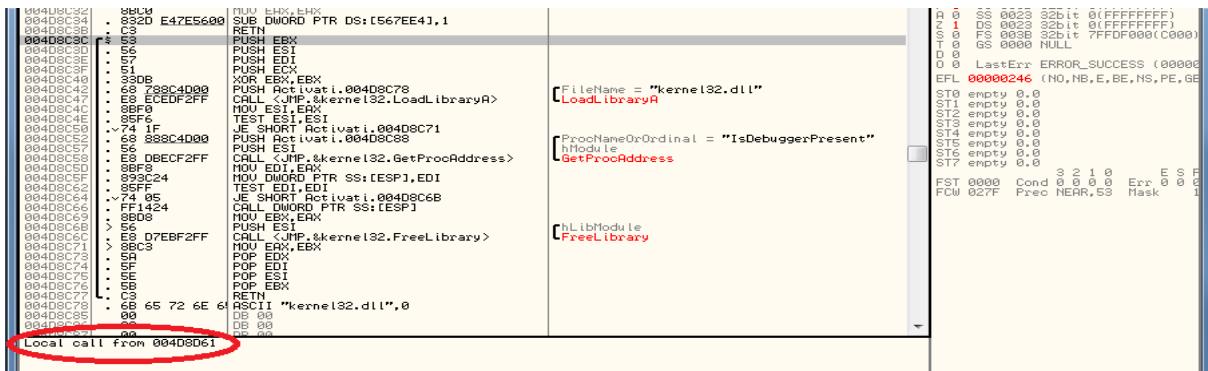


Figure 4

Dans la zone mise en évidence dans le cercle rouge, nous voyons l'adresse depuis laquelle ce call est appelé. Faites donc clic droit sur Local call from 004D8D61 (l'adresse peut être différente chez vous) puis « Go to CALL from 004D8D61 » pour arriver à l'endroit d'où la fonction est appelée. Nous arrivons ici :

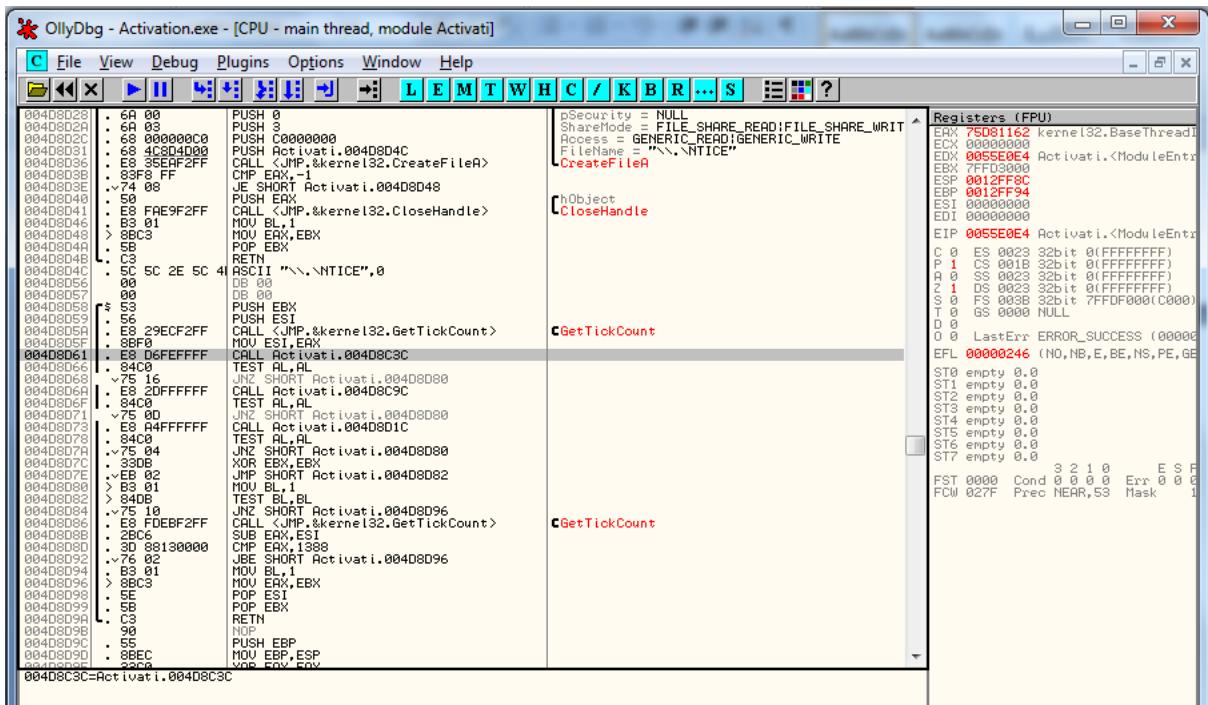


Figure 5

Placez un breakpoint sur la ligne mise en évidence par Olly en appuyant sur f2. Cela va permettre d'arrêter le programme au moment où il passera sur cette instruction. Maintenant lancez le programme avec f9. Passez les exceptions comme nous l'avons vu au point n°1. Nous arrivons ensuite sur notre breakpoint.

Analysons maintenant la suite du code ligne par ligne :

004D8D61 |. E8 D6FFFF CALL Activati.004D8C3C

Nous l'avons vu, cette ligne fait appel à la fonction vérifiant la présence d'un debugger

004D8D66 |. 84C0 TEST AL,AL

Cette ligne teste la valeur renournée par le call précédent, et contenue dans AL

004D8D68 75 16 JNZ SHORT Activati.004D8D80

Si cette valeur est différente de 0, on saute en 004D8D80

```
004D8D6A E8 2DFFFFFF CALL Activati.004D8C9C
```

Sinon, on continue et on fait donc appel à ce call

```
004D8D6F |. 84C0 TEST AL,AL
```

Cette ligne teste la valeur retournée par le call précédent, et contenue dans AL

```
004D8D71 75 0D JNZ SHORT Activati.004D8D80
```

Si cette valeur est différente de 0, on saute en 004D8D80 (au même endroit que le JNZ précédent)

```
004D8D73 |. E8 A4FFFFFF CALL Activati.004D8D1C
```

```
004D8D78 |. 84C0 TEST AL,AL
```

```
004D8D7A 75 04 JNZ SHORT Activati.004D8D80
```

Ces trois lignes font la même chose que précédemment, en appelant un call différent

```
004D8D7C |. 33DB XOR EBX,EBX
```

On met à 0 la valeur du registre EBX

```
004D8D7E |. EB 02 JMP SHORT Activati.004D8D82
```

On saute en 004D8D82

```
004D8D80 B3 01 MOV BL,1
```

Cette ligne met la valeur 1 dans BL (BL est un des deux registres 16bits contenu dans EBX qui lui fait donc 32bits)

```
004D8D82 |> 84DB TEST BL,BL
```

On teste la valeur de BL

```
004D8D84 |. 75 10 JNZ SHORT Activati.004D8D96
```

Si la valeur de BL est différente de 0 on saute en 004D8D96

```
004D8D86 |. E8 FDEBF2FF CALL <JMP.&kernel32.GetTickCount> ; [GetTickCount]
```

Sinon on exécute ce call

```
004D8D8B |. 2BC6 SUB EAX,ESI
```

On met dans EAX le résultat de la soustraction EAX-ESI

```
004D8D8D |. 3D 88130000 CMP EAX,1388
```

On compare EAX à la valeur 1388

```
004D8D92 |. 76 02 JBE SHORT Activati.004D8D96
```

Si la valeur dans EAX est plus grande que 1388 on saute en 004D8D96

```
004D8D94 B3 01 MOV BL,1
```

On met la valeur 1 dans BL

```
004D8D96 8BC3      MOV EAX,EBX
```

On met EBX dans EAX

```
004D8D98 5E      POP ESI  
004D8D99 5B      POP EBX  
004D8D9A C3      RETN
```

Ces 3 lignes marquent la fin du call elles ne nous intéressent donc pas.

Voilà donc la description de ce code assembleur. Maintenant je vais afficher seulement les instructions sur lesquelles passe le programme. On verra donc ce qui se passe lorsqu'on le lance avec un debugger (puisque c'est le cas actuellement :p). Pour voir les instructions, appuyez sur f8 afin de faire défiler l'exécution pas-à-pas.

```
004D8D61 |. E8 D6FFFF CALL Activati.004D8C3C  
004D8D66 |. 84C0      TEST AL,AL  
004D8D68 75 16      JNZ SHORT Activati.004D8D80  
004D8D80 B3 01      MOV BL,1  
004D8D82 |> 84DB     TEST BL,BL  
004D8D84 |. 75 10      JNZ SHORT Activati.004D8D96  
004D8D96 8BC3      MOV EAX,EBX  
004D8D98 5E      POP ESI  
004D8D99 5B      POP EBX  
004D8D9A C3      RETN
```

Voilà donc isolées les instructions exécutées par le programme s'il est lancé en debug. Maintenant une petite précision sur le fonctionnement de l'assembleur afin de mieux comprendre la suite.

Nous avons vu que le programme fait souvent appel à des « call », et que ces « call » sont souvent suivis d'un test (« test AL,AL » par exemple), et d'un saut conditionnel. La convention veut que le résultat d'un « call » soit placé dans le registre EAX. EAX(32bits) = AL(16bits)+AH(16bits). Ensuite EAX est testé à la suite du « call » pour savoir ce qu'a retourné son exécution.

Revenons à notre programme. Nous voyons que dans le cas d'une exécution par debugger, le programme effectue les instructions :

```
004D8D80 B3 01      MOV BL,1  
004D8D96 8BC3      MOV EAX,EBX  
004D8D98 5E      POP ESI  
004D8D99 5B      POP EBX  
004D8D9A C3      RETN
```

Le programme met donc la valeur 1 dans BL (première ligne), puis copie le contenu de EBX (donc de BL, donc 1) dans EAX. Ensuite on a droit à la séquence POP POP RET qui marque la fin du « call ». Pour résumer, notre « call » ne fait rien d'autre que retourner la valeur 1, puisqu'elle est contenue dans EAX avant le POP POP RET.

Mais dites-moi, cela voudrait donc dire que si notre programme est lancé avec un debugger, ce call retourne 1. Donc théoriquement, si on le force à retourner 0, on doit pouvoir lui faire croire que le programme s'exécute normalement, sans debugger. Eh bien allons-y !

Relancez le programme avec Ctrl-f2, puis lancez le f9, passez l'exception Shift-f9, vous arrivez sur notre breakpoint. Maintenant, on va forcer le call dans lequel nous sommes à retourner 0. Pour cela, double cliquez sur la ligne suivante (attention il y en a deux, prenez la première)

```
004D8D80 B3 01 MOV BL,1,
```

Une fenêtre d'édition s'ouvre, remplacez donc

```
004D8D80 B3 01 MOV BL,1
```

par

```
004D8D80 B3 01 MOV BL,0
```

puis cliquez sur « Assemble » et fermez la fenêtre.

Maintenant, continuons l'exécution du programme, appuyez 6 fois sur f8. Nous remarquons une chose, le programme ne passe pas par le même chemin qu'avant. En effet, le test

```
004D8D82 |> 84DB TEST BL,BL
```

```
004D8D84 |. 75 10 JNZ SHORT Activati.004D8D96
```

N'a plus le même résultat puisque nous avons mis dans BL la valeur 0. Le JNZ (jump if not zero) n'a donc pas lieu, et le programme passe aux instructions suivantes :

```
004D8D86 |. E8 FDEBF2FF CALL <JMP.&kernel32.GetTickCount>
004D8D8B |. 2BC6 SUB EAX,ESI
004D8D8D |. 3D 88130000 CMP EAX,1388
004D8D92 |. 76 02 JBE SHORT Activati.004D8D96
004D8D94 B3 01 MOV BL,1
004D8D96 8BC3 MOV EAX,EBX
004D8D98 5E POP ESI
004D8D99 5B POP EBX
004D8D9A C3 RETN
```

Ne nous occupons pas des 4 premières lignes. Ce qui est important c'est qu'avant de sortir du call, le programme va passer sur cette instruction, qui met la valeur 1 dans BL

```
004D8D94 B3 01 MOV BL,1
```

L'instruction suivante place BL dans EAX. EAX va donc se retrouver avec la valeur 1, ce qui va tout nous foutre en l'air puisqu'on veut absolument que notre call retourne 0 !

Il faut donc procéder à une deuxième modification et changer cette ligne

```
004D8D94 B3 01 MOV BL,1
```

en

```
004D8D94 B3 01 MOV BL,0
```

de la même façon que nous l'avons fait juste avant. On relance donc notre programme avec Ctrl-f2, f9, Shift-f9. Nous retombons sur notre breakpoint et cette fois ci, on édite ces deux lignes

```
004D8D80 B3 01 MOV BL,1
```

et

```
004D8D94 B3 01 MOV BL,1
```

en remplaçant 1 par un 0. Ensuite, appuyez sur f9 (pour relancer l'exécution mais pas « pas-à-pas ») et le programme se lance normalement ! pas de message d'erreur désagréable, on a réussi notre coup.

Maintenant il ne reste plus qu'à modifier le .exe avec un éditeur hexadecimal pour que les modifications soient durables (Olly ne modifie pas le fichier lui-même).

A présent, vous allez me détester :D regardez ça :

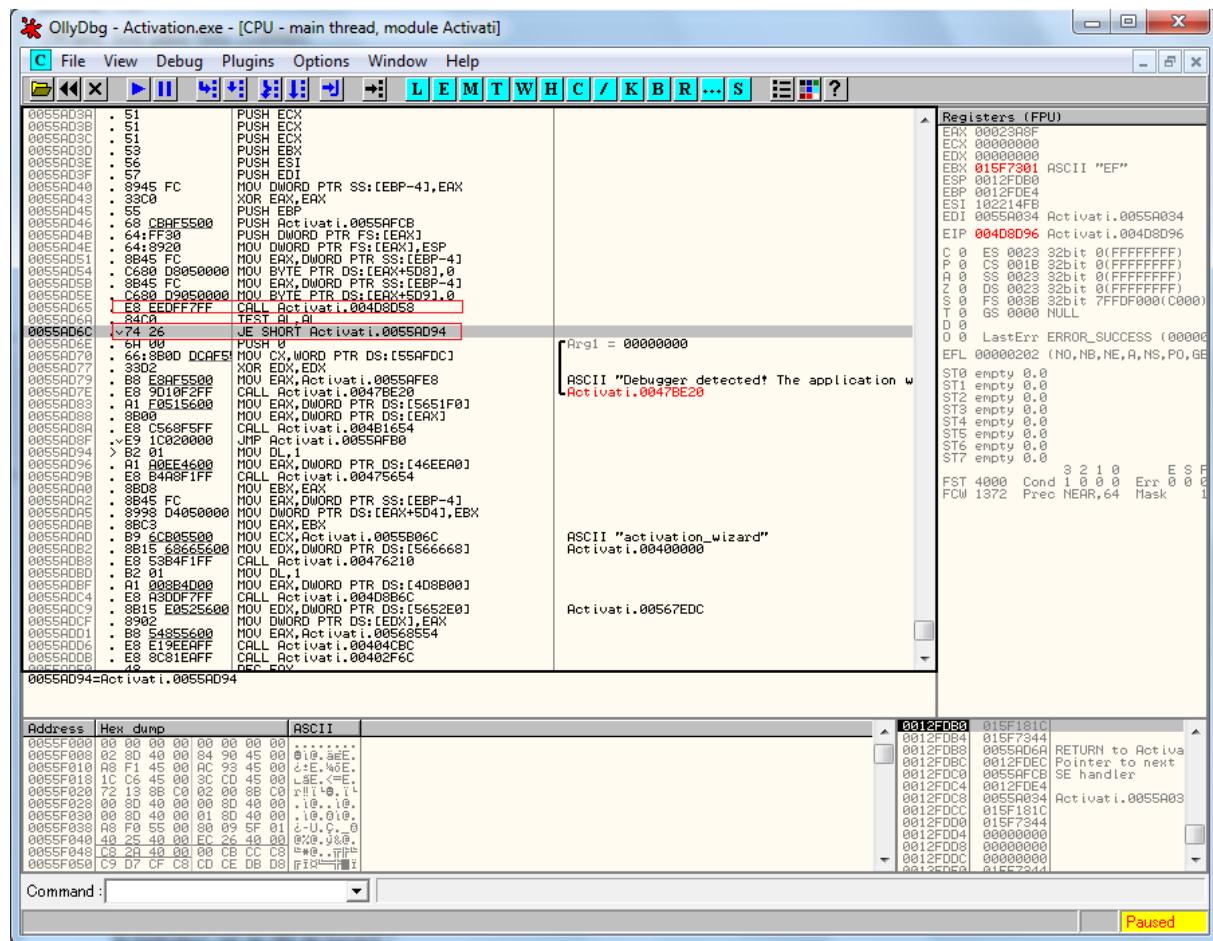


Figure 6

Le premier cadre rouge est l'appel au « call » que nous auscultons depuis tout à l'heure hihi. Ce call est suivi d'un JE (jump if equal) qui passe par-dessus le texte "Debugger detected! The application will now terminate. Please close the debugger and restart the application. Thank you!".

Donc il suffisait de remplacer le "JE" (saut étant fonction du retour du call) par un "JMP" (saut inconditionnel) pour éviter la protection anti-debug :p. De cette façon, le code

responsable de l'affichage du message d'erreur ne serait jamais appelé puisqu'on sauterait systématiquement au dessus.

Cela dit, on a appris bien plus de chose en apprenant la première solution, plutôt que si je vous avais balancé cette astuce du premier coup. En revanche, dans la vraie vie, on ne s'embêterait pas à comprendre tous les détails d'un call pour arriver à ses fins. On changerait directement le « JE » en « JMP » et basta. Voyons à présent comment modifier notre exécutable. Pour cela, nous nous serviront de l'éditeur hexadécimal Winhex.

IV/ Modification de l'exécutable grâce à WinHex

Lançons Winhex et chargeons notre exécutable dedans. Pour simplifier, je vais vous montrer comment effectuer le changement sur la deuxième et dernière méthode que nous avons vu pour faire sauter la protection anti-debug.

Vous avez donc pu remarquer que la deuxième colonne d'Olly nous montre les « opcodes » hexadécimaux correspondant à chaque instruction de la colonne de droite. C'est grâce à ces instructions que nous allons repérer dans Winhex le morceau de code à modifier.

Dans Winhex allez dans le menu Search -> find Hex Values.

Dans le champ de recherche tapez ceci :

8B45C680D9050000E8EEDFF7FF84C07426

Il faut taper un nombre assez significatif d'opcodes pour ne pas que la recherche ait plusieurs résultats. En effet, si vous recherchez juste 7426 (instruction JE), vous allez avoir un bon paquet de réponses et vous n'aurez pas trouvé le code que vous voulez. Les opcodes ci-dessus correspondent au code suivant :

```
0055AD5B . 8B45 FC    MOV EAX,DWORD PTR SS:[EBP-4]
0055AD5E . C680 D9050000 >MOV BYTE PTR DS:[EAX+5D9],0
0055AD65 . E8 EEDFF7FF CALL Activati.004D8D58
0055AD6A . 84C0      TEST AL,AL
0055AD6C . 74 26     JE SHORT Activati.0055AD94
```

Lancez la recherche, nous tombons donc sur la portion de code qui nous intéresse. Repérez l'instruction 7426 puis changez la en EB26 pour changer le « JE » en « JMP ». Enregistrez ensuite l'exécutable puis lancez le. Si vous avez bien écouté la leçon tout se passera normalement ☺

Bon courage pour la suite !!