CrackMe 5 – TTO

solution par mastermatt29

1.Introduction..2
2.Point de départ..3
3.L'importance des cases.......................................6
4.Le serial...12
5.Conclusion...14

1.Introduction

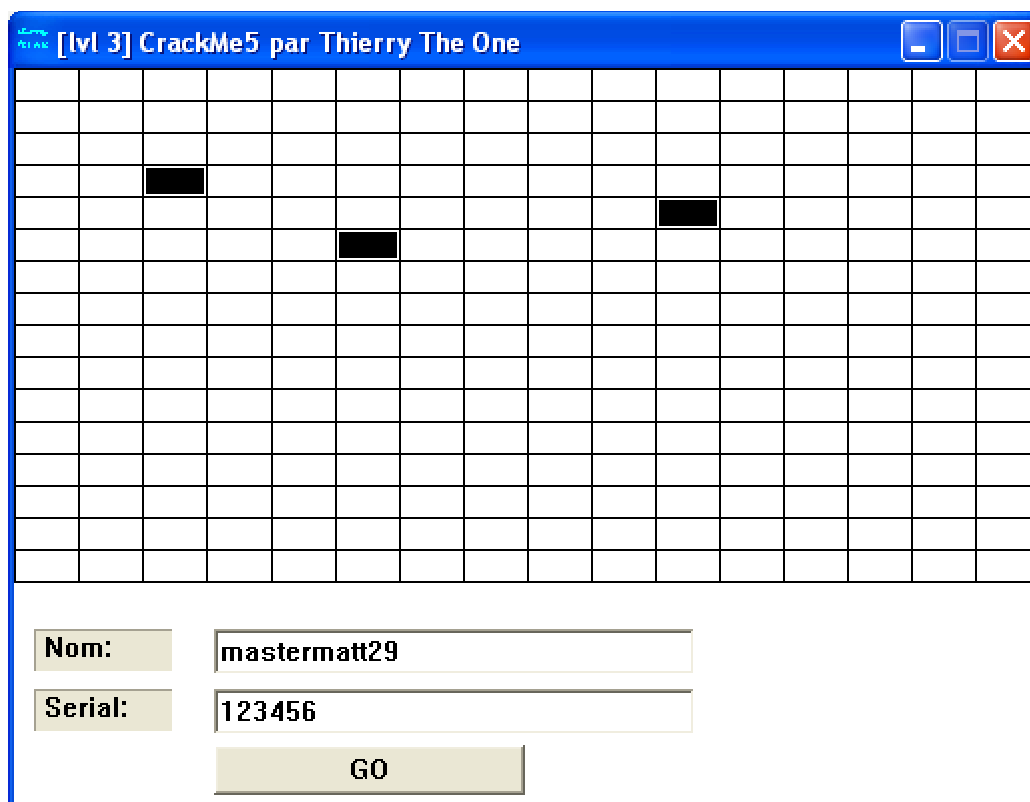
Je propose ici une solution au crackme 5 de Thierry The One. Ce tutorial fait partie d'un projet initié par Gamera pour la NAS (powaaaa) et regroupant les 8 solutions des 8 crackme de TTO.

Dans la suite de ce tutorial, **toute valeur sera systematiquement exprimée en héxadécimal** sauf indication contraire.

2.Point de départ

Ne sachant pas trop par où commencer pour rédiger ce tutorial, j'ai choisi d'exposer dans l'ordre la méthode que j'ai employé pour le résoudre (ce qui somme toute est assez logique).

Premièrement commençons par une petite analyse du fameux crackme une fois ouvert :



Donc *a priori* un couple name/serial. Reste ces petites cases de ce tableau de 10 sur 10 qui se noircissent dès que l'on clique dessus. Il va falloir trouver leur rôle, car vous vous en doutez, on n'aura un message de réussite que si les bonnes cases sont cliquées !

Etant donné que le crackme n'est pas packé, on peut le charger directement dans OllyDbg. Mon premier réflexe a été de poser un BP sur GetWindowTextA (fonction la plus courante pour récupérer du texte avec GetDlgItemTextA).

```
00401C6C  $-FF25 00204000  JMP DWORD PTR DS:[<&gdi32.CreateSolidBr  gdi32.CreateSolidBrush
00401C72  $-FF25 04204000  JMP DWORD PTR DS:[<&gdi32.DeleteObject>  gdi32.DeleteObject
00401C78  $-FF25 08204000  JMP DWORD PTR DS:[<&gdi32.ExtTextOutA>]  gdi32.ExtTextOutA
00401C7E  $-FF25 0C204000  JMP DWORD PTR DS:[<&gdi32.LineTo>]       gdi32.LineTo
00401C84  $-FF25 10204000  JMP DWORD PTR DS:[<&gdi32.MoveToEx>]     gdi32.MoveToEx
00401C8A  $-FF25 14204000  JMP DWORD PTR DS:[<&gdi32.SelectObject>  gdi32.SelectObject
00401C90  $-FF25 18204000  JMP DWORD PTR DS:[<&gdi32.SetBkColor>]   gdi32.SetBkColor
00401C96  $-FF25 1C204000  JMP DWORD PTR DS:[<&gdi32.SetTextColor>  gdi32.SetTextColor
00401C9C  $-FF25 20204000  JMP DWORD PTR DS:[<&gdi32.TextOutA>]     gdi32.TextOutA
00401CA2  $-FF25 9C204000  JMP DWORD PTR DS:[<&user32.wsprintfA>]   USER32.wsprintfA
00401CA8  $-FF25 98204000  JMP DWORD PTR DS:[<&user32.BeginPaint>]  USER32.BeginPaint
00401CAE  $-FF25 94204000  JMP DWORD PTR DS:[<&user32.CreateWindow  USER32.CreateWindowExA
00401CB4  $-FF25 90204000  JMP DWORD PTR DS:[<&user32.DefWindowPro  USER32.DefWindowProcA
00401CBA  $-FF25 8C204000  JMP DWORD PTR DS:[<&user32.DispatchMess  USER32.DispatchMessageA
00401CC0  $-FF25 54204000  JMP DWORD PTR DS:[<&user32.EndPaint>]    USER32.EndPaint
00401CC6  $-FF25 40204000  JMP DWORD PTR DS:[<&user32.FillRect>]    USER32.FillRect
00401CCC  $-FF25 44204000  JMP DWORD PTR DS:[<&user32.GetDC>]       USER32.GetDC
00401CD2  $-FF25 48204000  JMP DWORD PTR DS:[<&user32.GetMessageA>  USER32.GetMessageA
00401CD8  $-FF25 4C204000  JMP DWORD PTR DS:[<&user32.GetSystemMet  USER32.GetSystemMetrics
00401CDE  $-FF25 50204000  JMP DWORD PTR DS:[<&user32.GetWindowTex  USER32.GetWindowTextA
00401CE4  $-FF25 A0204000  JMP DWORD PTR DS:[<&user32.InvalidateRe  USER32.InvalidateRect
00401CEA  $-FF25 58204000  JMP DWORD PTR DS:[<&user32.KillTimer>]   USER32.KillTimer
00401CF0  $-FF25 5C204000  JMP DWORD PTR DS:[<&user32.LoadCursorA>  USER32.LoadCursorA
00401CF6  $-FF25 60204000  JMP DWORD PTR DS:[<&user32.LoadIconA>]   USER32.LoadIconA
00401CFC  $-FF25 64204000  JMP DWORD PTR DS:[<&user32.MessageBoxA>  USER32.MessageBoxA
00401D02  $-FF25 68204000  JMP DWORD PTR DS:[<&user32.PostQuitMess  USER32.PostQuitMessage
00401D08  $-FF25 6C204000  JMP DWORD PTR DS:[<&user32.RegisterClas  USER32.RegisterClassExA
00401D0E  $-FF25 70204000  JMP DWORD PTR DS:[<&user32.ReleaseDC>]   USER32.ReleaseDC
00401D14  $-FF25 74204000  JMP DWORD PTR DS:[<&user32.SendMessageA  USER32.SendMessageA
00401D1A  $-FF25 78204000  JMP DWORD PTR DS:[<&user32.SetRect>]     USER32.SetRect
00401D20  $-FF25 7C204000  JMP DWORD PTR DS:[<&user32.SetTimer>]    USER32.SetTimer
00401D26  $-FF25 80204000  JMP DWORD PTR DS:[<&user32.ShowWindow>]  USER32.ShowWindow
00401D2C  $-FF25 84204000  JMP DWORD PTR DS:[<&user32.TranslateMes  USER32.TranslateMessage
00401D32  $-FF25 88204000  JMP DWORD PTR DS:[<&user32.UpdateWindow  USER32.UpdateWindow
00401D38  .-FF25 38204000  JMP DWORD PTR DS:[<&kernel32.ExitProces  kernel32.ExitProcess
00401D3E  $-FF25 34204000  JMP DWORD PTR DS:[<&kernel32.GetCommandL  kernel32.GetCommandLineA
00401D44  $-FF25 30204000  JMP DWORD PTR DS:[<&kernel32.GetModuleH  kernel32.GetModuleHandleA
00401D4A  $-FF25 2C204000  JMP DWORD PTR DS:[<&kernel32.lstrcatA>]  kernel32.lstrcatA
00401D50  $-FF25 28204000  JMP DWORD PTR DS:[<&kernel32.lstrcmpA>]  kernel32.lstrcmpA
```

Ensuite hop un coup de F9 pour lancer le programme, on rentre des infos bidons (mastermatt29/123456 pour moi), on noirci 3 cases pour faire bien et on clique sur Go !

La on break sur notre BP. On fait un Ctrl+F9 (Execute Till Return) puis F8 pour arriver à l'endroit où a GetWindowTextA a été appelé, c'est à dire ici :

```
004011A6  . 68 00010000    PUSH 100                             ┌Count = 100 (256.)
004011AB  . 68 10334000    PUSH tto_Crac.00403310               │Buffer = tto_Crac.00403310
004011B0  . FF35 08314000  PUSH DWORD PTR DS:[403108]           │hWnd = 0012051A (class='Edit',parent=00740118)
004011B6  . E8 230B0000    CALL <JMP.&user32.GetWindowTextA>    └GetWindowTextA
004011BB  . 3A05 7A304000  CMP AL,BYTE PTR DS:[40307A]           Nombre de cases cocher
004011C1  .v0F85 8D010000  JNZ <tto_Crac.Bad Boy>
004011C7  . 68 90010000    PUSH 190                             ┌Bottom = 190 (400.)
004011CC  . 68 06020000    PUSH 206                             │Right = 206 (518.)
004011D1  . 6A 00          PUSH 0                               │Top = 0
004011D3  . 6A 00          PUSH 0                               │Left = 0
004011D5  . 68 39304000    PUSH tto_Crac.00403039               │pRect = tto_Crac.00403039
004011DA  . E8 3B0B0000    CALL <JMP.&user32.SetRect>           └SetRect
004011DF  . 6A 01          PUSH 1                               ┌Erase = TRUE
004011E1  . 68 39304000    PUSH tto_Crac.00403039               │pRect = 00403039 {162.,82.,191.,95.}
004011E6  . FF75 08        PUSH DWORD PTR SS:[EBP+8]            │hWnd
004011E9  . E8 F60A0000    CALL <JMP.&user32.InvalidateRect>    └InvalidateRect
004011EE  . 6A 00          PUSH 0                               ┌lParam = 0
004011F0  . 6A 00          PUSH 0                               │wParam = 0
004011F2  . 6A 0F          PUSH 0F                              │Message = WM_PAINT
004011F4  . FF75 08        PUSH DWORD PTR SS:[EBP+8]            │hWnd
004011F7  . E8 180B0000    CALL <JMP.&user32.SendMessageA>      └SendMessageA
004011FC  . 6A 00          PUSH 0                               ┌Timerproc = NULL
004011FE  . 6A 01          PUSH 1                               │Timeout = 1. ms
00401200  . 6A 0D          PUSH 0D                              │TimerID = D (13.)
00401202  . FF75 08        PUSH DWORD PTR SS:[EBP+8]            │hWnd
00401205  . E8 160B0000    CALL <JMP.&user32.SetTimer>          └SetTimer
0040120A  . C705 72304000  MOV DWORD PTR DS:[403072],-1
00401214  . C705 76304000  MOV DWORD PTR DS:[403076],0
0040121E  . C605 71304000  MOV BYTE PTR DS:[403071],0
00401225  . B8 00000000    MOV EAX,0
0040122A  . C9             LEAVE
0040122B  . C2 1000        RETN 10
```

Ce que l'on voit c'est que notre name (parce que c'est lui en l'occurence) est stocké en 403310 en que sa longueur est comparée à un nombre stocké en 40307A qui se révèle être le nombre de cases cochées, ou pour être plus exact, le nombre de cases cliquées.

Ensuite viennent deux fonctions (SetRect et InvalidateRect) qui ne nous sont pas utiles. En revanche ensuite vient SetTimer qui est beaucoup plus intéressant ! Voici ce que dis ma doc :

« The SetTimer function creates a timer with the specified time-out value. [...] If lpTimerFunc (Timerproc) is NULL, the system posts a WM_TIMER message to the application queue. »

Ainsi donc cette fonction va envoyer un message WM_TIMER dans notre Callback principal (il s'agit de la fonction traitant les messages que Windows envoie à notre application) dans 1 ms et nous quelques instructions plus tard on tombe sur un RET. L'intérêt est évident : poursuivre la vérification dans une autre partie de code !

Ainsi il ne nous reste plus qu'à trouver la fonction Callback et à poser un BP sur le cas WM_TIMER !

3.L'importance des cases

Bon autant vous le dire de suite la Callback se trouve en 40116C. Ce n'est pas difficile de le déterminer, il suffit de regarder la structure WNDCLASSEX passée à RegisterClassExA.

Cette callback traite plusieurs messages, le premier est WM_COMMAND :

```
00401172  . 817D 0C 11010( CMP DWORD PTR SS:[EBP+C],111          WM_COMMAND
00401179  .v0F85 FA010000  JNZ tto_Crac.00401379                 Va voir le message suivant
0040117F  . 817D 10 F4010( CMP DWORD PTR SS:[EBP+10],1F4
00401186  .v0F85 DF010000  JNZ tto_Crac.0040136B
0040118C  . 833D 51304000  CMP DWORD PTR DS:[403051],1
00401193  .v0F84 95000000  JE tto_Crac.0040122E
00401199  . 833D 51304000  CMP DWORD PTR DS:[403051],2
004011A0  .v0F8D C5010000  JGE tto_Crac.0040136B
004011A6  . 68 00010000    PUSH 100                              [Count = 100 (256.)
004011AB  . 68 10334000    PUSH tto_Crac.00403310                |Buffer = tto_Crac.00403310
004011B0  . FF35 08314000  PUSH DWORD PTR DS:[403108]            |hWnd = NULL
004011B6  . E8 230B0000    CALL <JMP.&user32.GetWindowTextA>     [GetWindowTextA
004011BB  . 3A05 7A304000  CMP AL,BYTE PTR DS:[40307A]           Nombre de cases cocher
004011C1  .v0F85 8D010000  JNZ <tto_Crac.Bad Boy>
```

et le second WM_TIMER :

```
00401379  > 817D 0C 13010( CMP DWORD PTR SS:[EBP+C],113          WM_TIMER
00401380  .v0F85 3F010000  JNZ tto_Crac.004014C5
00401386  . C705 51304000  MOV DWORD PTR DS:[403051],1           Mise a 1 de 403051 (necessaire lors du WM_COMMAND)
00401390  . FF75 08        PUSH DWORD PTR SS:[EBP+8]             [hWnd
00401393  . E8 34090000    CALL <JMP.&user32.GetDC>              [GetDC
```

Les autres messages ne nous serons pas utiles.

En regardant le début du code du WM_TIMER on s'apercoit qu'il y a une mise à 1 de 403051 dès le début du code. Or dans le début du code du WM_COMMAND, on compare d'abord le wParam à 1F4 puis on compare 403051 à 1 puis à 2 avec différents comportements pour chacun. Si on regarde un peu plus bas du code WM_TIMER on voit ceci :

```
00401499  > 6A 0D          PUSH 0D                               [TimerID = D (13.)
0040149B  . FF75 08        PUSH DWORD PTR SS:[EBP+8]             |hWnd
0040149E  . E8 47080000    CALL <JMP.&user32.KillTimer>          [KillTimer
004014A3  . 6A 00          PUSH 0                                [lParam = 0
004014A5  . 68 F4010000    PUSH 1F4                              |wParam = 1F4
004014AA  . 68 11010000    PUSH 111                              |Message = WM_COMMAND
004014AF  . FF75 08        PUSH DWORD PTR SS:[EBP+8]             |hWnd
004014B2  . E8 5D080000    CALL <JMP.&user32.SendMessageA>       [SendMessageA
004014B7  . B8 00000000    MOV EAX,0
004014BC  . C9             LEAVE
004014BD  . C2 1000        RETN 10
```

Donc on tue notre timer et on envoie un message WM_COMMAND à notre application avec 1F4 en wParam !

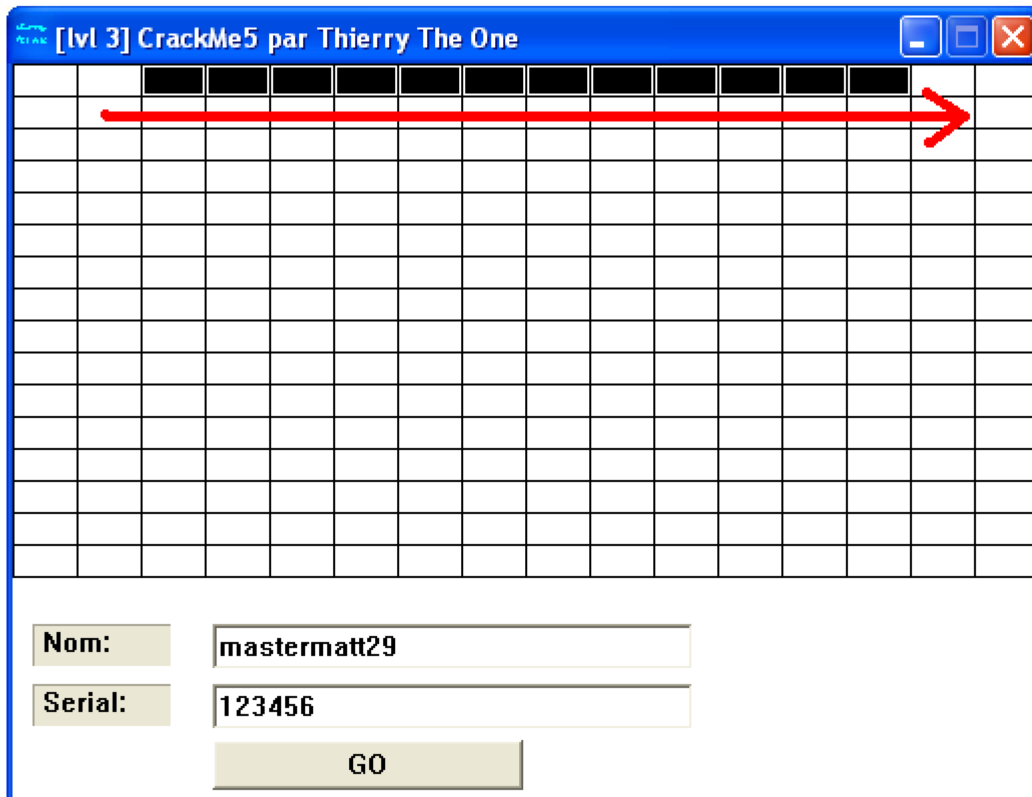
Ainsi donc, lors du WM_TIMER, on met 403051 à 1 et on envoie un message WM_COMMAND. Ainsi lors du traitement de ce message on arrive en 40122E (cf le début du code WM_COMMAND).

```
0040122E  > C705 51304000 MOV DWORD PTR DS:[403051],2            403051 = 2
00401238  . C705 65304000 MOV DWORD PTR DS:[403065],0
00401242  . C605 10354000 MOV BYTE PTR DS:[403510],0
00401249  > FF05 65304000 INC DWORD PTR DS:[403065]
0040124F  . B8 FFFFFFFF   MOV EAX,-1
00401254  > 40            INC EAX
00401255  . 3D 00010000   CMP EAX,100
0040125A  .v74 58         JE SHORT tto_Crac.004012B4             Va a la comparaison des chaines
0040125C  . 33DB          XOR EBX,EBX
0040125E  . 8A98 10314000 MOV BL,BYTE PTR DS:[EAX+403110]        Charge un octet de l'espace 403110
00401264  . 3B1D 65304000 CMP EBX,DWORD PTR DS:[403065]          On verifie qu'il correspond a 403065 sinon on prend le suivant
0040126A  .^75 E8         JNZ SHORT tto_Crac.00401254
0040126C  . 50            PUSH EAX                               [Arg1
0040126D  . E8 32080000   CALL tto_Crac.00401AA4                 [tto_Crac.00401AA4
00401272  . 0105 7B304000 ADD DWORD PTR DS:[40307B],EAX
00401278  . 50            PUSH EAX                               [<%.8lX>
00401279  . 68 4B304000   PUSH tto_Crac.0040304B                 |Format = "%.8lX"
0040127E  . 68 10344000   PUSH tto_Crac.00403410                 |s = tto_Crac.00403410
00401283  . E8 1A0A0000   CALL <JMP.&user32.wsprintfA>           [wsprintfA
00401288  . 83C4 0C       ADD ESP,0C
0040128B  . 803D 10354000 CMP BYTE PTR DS:[403510],0
00401292  .v74 0F         JE SHORT tto_Crac.004012A3
00401294  . 68 49304000   PUSH tto_Crac.00403049                 [StringToAdd = "-"
00401299  . 68 10354000   PUSH tto_Crac.00403510                 |ConcatString = ""
0040129E  . E8 A70A0000   CALL <JMP.&kernel32.lstrcatA>          [lstrcatA
004012A3  > 68 10344000   PUSH tto_Crac.00403410                 [StringToAdd = ""
004012A8  . 68 10354000   PUSH tto_Crac.00403510                 |ConcatString = ""
004012AD  . E8 980A0000   CALL <JMP.&kernel32.lstrcatA>          [lstrcatA
004012B2  .^EB 95         JMP SHORT tto_Crac.00401249
004012B4  > 68 CB304000   PUSH tto_Crac.004030CB                 [String2 = "52025EF2-83048FF4-52065EF6-83088FF8-520A5EFA-830C8FFC"
004012B9  . 68 10354000   PUSH tto_Crac.00403510                 |String1 = ""
004012BE  . E8 8D0A0000   CALL <JMP.&kernel32.lstrcmpA>          [lstrcmpA
```

Donc ici il va falloir relancer le crackme et cliquer sur autant de cases que l'on a de lettres dans notre nom pour passer la première vérification. Ainsi on peut tranquillement poser notre BP sur 40122E.

Ici il est intéressant de regarder l'espace mémoire en 403110.

Avec le crackme remplit comme ceci :



On obtient ceci :

Address	Hex dump	ASCII
004030F0	32 30 41 35 45 46 41 2D	20A5EFA-
004030F8	38 33 30 43 38 46 46 43	830C8FFC
00403100	00 42 55 54 54 4F 4E 00	.BUTTON.
00403108	BE 04 3F 00 90 04 25 00	¥♦?.É♦%.
00403110	00 00 01 02 03 04 05 06	..☺☻♥♦♣♠
00403118	07 08 09 0A 0B 0C 00 00	•◘..♂...
00403120	00 00 00 00 00 00 00 00	....
00403128	00 00 00 00 00 00 00 00	....
00403130	00 00 00 00 00 00 00 00	....
00403138	00 00 00 00 00 00 00 00	....
00403140	00 00 00 00 00 00 00 00	....
00403148	00 00 00 00 00 00 00 00	....
00403150	00 00 00 00 00 00 00 00	....
00403158	00 00 00 00 00 00 00 00	

Ainsi si vous regardez bien, les cases sur lesquelles on a cliqué apparaissent dans l'ordre avec le numéro 1 pour la première 2 pour la 2e...etc.

En regardant le code, on s'apercoit donc que la boucle est :

- trouver la position de la case courante
- passer cette valeur dans l'algorithme en 401AA4
- convertir ce nombre en une chaine le représentant en héxadécimal
- concaténer cette chaine à une grande chaine qui sera finalement comparée à 52025EF2-83048FF4-52065EF6-83088FF8-520A5EFA-830C8FFC

Une conséquence directe du fait que cette chaine ait 6 groupes de caractères est que le name fera obligatoirement 6 caractères de long.

Maintenant étudions l'algorithme en 401AA4.

```
00401AA4  $ 55            PUSH EBP
00401AA5  . 8BEC          MOV EBP,ESP
00401AA7  . 83C4 E4       ADD ESP,-1C
00401AAA  . C745 E4 00000 MOV DWORD PTR SS:[EBP-1C],0
00401AB1  . 8B45 08       MOV EAX,DWORD PTR SS:[EBP+8]          [EBP+8] represente la position actuellement hashee
00401AB4  . C1E8 04       SHR EAX,4
00401AB7  . C1E0 04       SHL EAX,4                             les 4 bits de poids faibles ont ete enlever
00401ABA  . 8B5D 08       MOV EBX,DWORD PTR SS:[EBP+8]
00401ABD  . 2BD8          SUB EBX,EAX                           EBX contient les bits de poids faibles
00401ABF  . C1E8 04       SHR EAX,4
00401AC2  . 895D EC       MOV DWORD PTR SS:[EBP-14],EBX         on stocke les 4 bits de poids faible
00401AC5  . 8945 E8       MOV DWORD PTR SS:[EBP-18],EAX         on stocke les 4 bits de poids fort
00401AC8  . 8B45 E8       MOV EAX,DWORD PTR SS:[EBP-18]
00401ACB  . C1E0 04       SHL EAX,4
00401ACE  . 33DB          XOR EBX,EBX
00401AD0  . 8A98 10324000 MOV BL,BYTE PTR DS:[EAX+403210]       charge le 1er caractere de la zone 403210
00401AD6  . 8B45 E4       MOV EAX,DWORD PTR SS:[EBP-1C]         EAX = 0
00401AD9  . C1E0 08       SHL EAX,8
00401ADC  . 8AC3          MOV AL,BL
00401ADE  . 8945 E4       MOV DWORD PTR SS:[EBP-1C],EAX         on sauvegarde EAX
00401AE1  . 8B45 EC       MOV EAX,DWORD PTR SS:[EBP-14]
00401AE4  . 33DB          XOR EBX,EBX
00401AE6  . 8A98 10324000 MOV BL,BYTE PTR DS:[EAX+403210]       charge le 2e caractere de la zone 403210
00401AEC  . 8B45 E4       MOV EAX,DWORD PTR SS:[EBP-1C]         restaure EAX
00401AEF  . C1E0 08       SHL EAX,8                             decale d'un octet
00401AF2  . 8AC3          MOV AL,BL
00401AF4  . 8945 E4       MOV DWORD PTR SS:[EBP-1C],EAX         sauvegarde de EAX
00401AF7  . 8B45 E8       MOV EAX,DWORD PTR SS:[EBP-18]
00401AFA  . C1E0 04       SHL EAX,4
00401AFD  . 83C0 0F       ADD EAX,0F
00401B00  . 33DB          XOR EBX,EBX
00401B02  . 8A98 10324000 MOV BL,BYTE PTR DS:[EAX+403210]       charge le 3e caractere de la zone 403210
00401B08  . 8B45 E4       MOV EAX,DWORD PTR SS:[EBP-1C]         restaure EAX
00401B0B  . C1E0 08       SHL EAX,8                             decale d'un octet
00401B0E  . 8AC3          MOV AL,BL
00401B10  . 8945 E4       MOV DWORD PTR SS:[EBP-1C],EAX         sauvegarde de EAX
00401B13  . B8 0F000000   MOV EAX,0F
00401B18  . C1E0 04       SHL EAX,4
00401B1B  . 0345 EC       ADD EAX,DWORD PTR SS:[EBP-14]
00401B1E  . 33DB          XOR EBX,EBX
00401B20  . 8A98 10324000 MOV BL,BYTE PTR DS:[EAX+403210]       charge le 4e caractere de la zone 403210
00401B26  . 8B45 E4       MOV EAX,DWORD PTR SS:[EBP-1C]         restaure EAX
00401B29  . C1E0 08       SHL EAX,8                             decale d'un octet
00401B2C  . 8AC3          MOV AL,BL
00401B2E  . C9            LEAVE
00401B2F  . C2 0400       RETN 4
```

Ici on va en quelque sorte hasher la position qui nous est passé en paramètre. Comme on peut le constater, le schéma est répétitif. Tout d'abord, la position est séparée en 2 groupes de 4 bits : les faibles (en EBP-14) et les forts (en EBP-18). Ensuite on prend une valeur dans l'espace mémoire commençant en 403210 avec comme index EAX. Les valeurs de EAX sont :

- 1er cas : bits de poids fort en position originale (par ex pour 00101100 on aurait 00100000)
- 2e cas : bits de poids faible
- 3e cas : valeur du 1er cas + 0F
- 4e cas : valeur du 2e cas + F0

Reste à savoir comment la zone de 403210 est calculée. Et là pour expliquer je sens que je vais avoir du mal !

Ici n'entrent en jeu que le name et les cases cochées. Première chose à savoir, les positions des cases dans notre tableau de 10*10 sont numérotées en partant de 0, de la gauche vers la droite et du haut vers le bas.
Imaginons que nous ayons à notre disposition une liste comprenant tous les chiffres compris entre 0 et 255. On enlève de cette liste tous les caractères du name. Ces derniers seront placés par ordre dans les emplacements définis par les numéros des cases cochées (ex : premier caractère du name : 'M', première case cochées : 45 et bien on mettra 'M' en 45e position).

Les emplacements restants seront remplis par les caractères qui n'étaient pas dans le name.

Regardons les 6 valeurs à trouver :

```
  52 | 02 | 5E | F2
- 83 | 04 | 8F | F4
- 52 | 06 | 5E | F6
- 83 | 08 | 8F | F8
- 52 | 0A | 5E | FA
- 83 | 0C | 8F | FC
--------------------
   1    2    3    4
```

Note : je ne cherche absolument pas à résoudre tous les cas possibles. Je prend donc dans tous les cas les chemins les plus simples même si ils induisent des restrictions

J'ai empilé les valeurs car dans chaque colonne chaque octet est issu de la même opération (bits de poids fort pour la colonne 1...etc).

Les colonnes les plus simples sont les colonnes 2 et 4. En effet les nombres de la deuxieme colonne correspondent aux octets de poids faible et ceux de la 4e colonne sont ceux à trouver F0 octets plus loin. Etant donné que en ajoutant F0 à la 2e colonne on obtient directement la 4e, il n'y a rien de spécial à dire, si ce n'est que les poids faibles des 6 positions sont dans l'ordre 2, 4, 6, 8, A et C.

Dans la colonne 1, il s'agit des valeurs à trouver à des emplacements tels que 10,20,30,70... car ce sont les 4 bits de poids forts (càd des nombres modulo 10). La colonne 3 représente les nombres à trouver F octets plus loin. On s'aperçoit que de 52 à 5E il y a C octets, càd 3 de moins que ce que ca devrait être. De même de 83 à 8F il manque 3 octets.

Avec toutes ces informations, on peut déduire que les positions des cases sont dans l'ordre :

- 52 = 50 + 2
- 84 = 80 + 4
- 56 = 50 + 6
- 88 = 80 + 8

- 5A = 50 + A
- 8C = 80 + C

Pour que ces valeurs conviennent, il est cependant nécessaire que le name contienne :

- 2 caractères dont le code est inférieur à 52 (pour que les décalages amènent 52 en position 50)
- 4 caractères dont le code est compris entre 5F et 83 (1 pour amener 83 en 80 et les autres pour compenser les ajouts faits par les cases)

Bon c'est sur on peut pas faire les name que l'on veut, mais au moins on est assuré d'avoir une solution valide !

[lvl 3] CrackMe5 par Thierry The One

0 0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
1															
2															
3															
4															
5															
6															
7															
8															
9															
A															
B															
C															
D															
E															
F															

Nom: NasPow

Serial:

GO

Donc pour fabriquer vos names, référez vous à une table ASCII et respectez les contraintes décrites au-dessus.

4.Le serial

Il s'agit de la dernière partie des vérifications. Reprenons à la comparaison des chaines en 4012C3.

```
004012B4  > 68 CB304000    PUSH tto_Crac.004030CB                     String2 = "52025EF2-83048FF4-52065EF6-83088FF8-520A5EFA-830C8FFC"
004012B9  . 68 10354000    PUSH tto_Crac.00403510                     String1 = ""
004012BE  . E8 8D0A0000    CALL <JMP.&kernel32.lstrcmpA>              lstrcmpA
004012C3  . 83F8 00        CMP EAX,0
004012C6  .v0F85 88000000  JNZ <tto_Crac.Bad Boy>
004012CC  . 68 00010000    PUSH 100                                   Count = 100 (256.)
004012D1  . 68 10344000    PUSH tto_Crac.00403410                     Buffer = tto_Crac.00403410
004012D6  . FF35 0C314000  PUSH DWORD PTR DS:[40310C]                 hWnd = NULL
004012DC  . E8 FD090000    CALL <JMP.&user32.GetWindowTextA>          GetWindowTextA
004012E1  . B8 FFFFFFFF    MOV EAX,-1
004012E6  . 33C9           XOR ECX,ECX
004012E8  > 40             INC EAX
004012E9  . 33DB           XOR EBX,EBX
004012EB  . 8A98 10344000  MOV BL,BYTE PTR DS:[EAX+403410]
004012F1  . 80FB 00        CMP BL,0                                   Switch (cases 0..39)
004012F4  .v74 14          JE SHORT tto_Crac.0040130A                 Fin du serial
004012F6  . 80EB 30        SUB BL,30
004012F9  . 80FB 00        CMP BL,0
004012FC  .v7C 56          JL SHORT <tto_Crac.Bad Boy>
004012FE  . 80FB 09        CMP BL,9
00401301  .v7F 51          JG SHORT <tto_Crac.Bad Boy>                Verifie s'il s'agit bien d'un caractere reprûsentant un chiffre
00401303  . 6BC9 0A        IMUL ECX,ECX,0A                            Cases 30 ('0'),31 ('1'),32 ('2'),33 ('3'),34 ('4'),35 ('5'),36 ('
00401306  . 03CB           ADD ECX,EBX                                Dans le but de convertir le serial en entier exploitable
00401308  .^EB DE          JMP SHORT tto_Crac.004012E8
0040130A  > 81F9 005ED0B2  CMP ECX,B2D05E00                           Serial valant au moins 3 000 000 000; Case 0 of switch 004012F1
00401310  .v7C 42          JL SHORT <tto_Crac.Bad Boy>
00401312  . E8 36090000    CALL tto_Crac.00401C4D                     Dernier CALL avant les felicitations
00401317  . 83F8 00        CMP EAX,0
0040131A  .v75 38          JNZ SHORT <tto_Crac.Bad Boy>
0040131C  . C705 51304000  MOV DWORD PTR DS:[403051],3
00401326  . 6A 00          PUSH 0                                     lParam = 0
00401328  . 6A 00          PUSH 0                                     wParam = 0
0040132A  . 6A 0F          PUSH 0F                                    Message = WM_PAINT
0040132C  . FF75 08        PUSH DWORD PTR SS:[EBP+8]                  hWnd
0040132F  . E8 E0090000    CALL <JMP.&user32.SendMessageA>            SendMessageA
00401334  . 6A 00          PUSH 0                                     Style = MB_OK|MB_APPLMODAL
00401336  . 68 00304000    PUSH tto_Crac.00403000                     Title = "[lvl 3] CrackMe5 par Thierry The One"
0040133B  . 68 92304000    PUSH tto_Crac.00403092                     Text = "Toutes mes fûlicitations!"
00401340  . FF35 29304000  PUSH DWORD PTR DS:[403029]                 hOwner = NULL
00401346  . E8 B1090000    CALL <JMP.&user32.MessageBoxA>             MessageBoxA
```

Ensuite on récupère le serial grâce à un appel à GetWindowTextA puis on le convertir en entier exploitable dans ECX. Le serial doit être exprimé en notation décimale. Première chose : il doit au moins être égal à B2D05E00 (càd 3 000 000 000 en décimal).

La dernière vérification se déroule en 401C4D :

```
00401C4D  $ BB 02000000    MOV EBX,2
00401C52  > 8BC1           MOV EAX,ECX
00401C54  . 33D2           XOR EDX,EDX
00401C56  . F7F3           DIV EBX
00401C58  . 83FA 00        CMP EDX,0
00401C5B  .v74 0E          JE SHORT tto_Crac.00401C6B
00401C5D  . 43             INC EBX
00401C5E  . 81FB 01000100  CMP EBX,10001
00401C64  .^7C EC          JL SHORT tto_Crac.00401C52
00401C66  . B8 00000000    MOV EAX,0
00401C6B  > C3             RETN
```

On vérifie que notre serial n'est divisible par aucun des nombres inférieurs à 10001, c'est à dire que ce nombre est premier.

Note : ici la comparaison s'effectue sur le reste de la division euclidienne de EAX (serial) par EBX (valeur allant de 2 à 10000), stocké dans EDX, qui ne doit pas être nul.

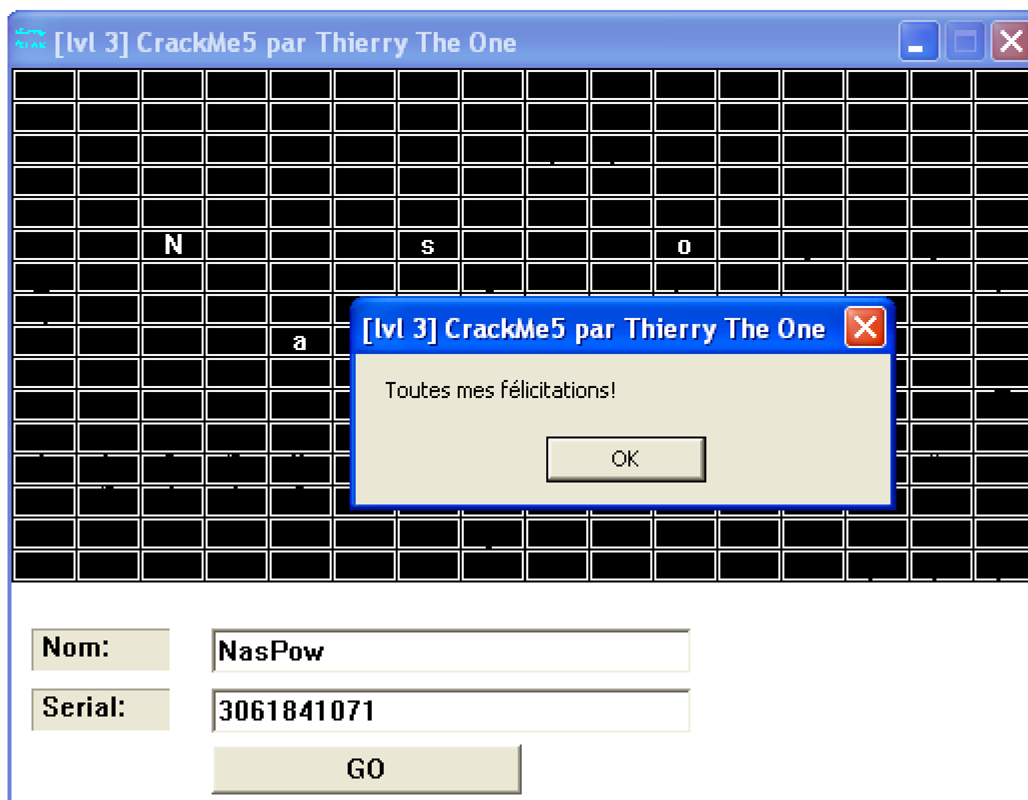
Bref, il suffit de trouver un nombre premier compris entre B2D05E00 et FFFFFFFF (car on ne peut pas aller plus loin sur 32 bits) tel qu'il ne soit pas divisible par aucun des nombres compris

entre 2 et 10000.

Quand je dis il suffit, il faut bien voir que ce n'est pas si simple. J'ai heureusement finit par tomber sur une liste de googolplex premiers (si vous ne savez pas ce que c'est cherchez ;-)) qui correspondent à nos conditions :

3061841071
3231644077
3278269147
3360968123
3408499249
3453742093
3523411999
3532577467
3593229751
3636363637
3694736321
3712650679
3807431923
4066647841
4399855387

Il ne reste qu'a mettre l'un des nombres en serial et on obtient :



5.Conclusion

Voilà ce crackme est enfin résolu ! Il n'est pas franchement facile mais je suis content d'avoir réussi à le terminer.

Pour conclure, je voudrai remercier Thierry The One pour son crackme sur lequel j'ai passé quelques heures et tous les membres de la NAS, en particulier Gamera pour m'avoir laissé participer à son projet.

15/09/06

mastermatt29^NAS