

CIBLE: **DBKeygenme** from Kharneth
URL : <http://kharneth.free.fr/>
NIVEAU: à vous d'en juger en fait.
AUTEUR: Requiem [FRET]

Outils utilisés :
 OllyDbg avec le plugin OllyDump,
 PEiD, MASM.



Ce crackme a donc été codé par **Kharneth** et comme son nom nous l'indique, le but sera de réaliser un keygen ; petites précisions de la part de **Kharneth**, il a été codé avec DevC++ 4, et on doit s'attendre à un leurre quelque part. Un scan avec PEiD nous apprendra que ce keygen est compressé avec UPX, je vous laisse le rendre clean, on peut par exemple utiliser OllyDump c'est très rapide et efficace.

D'un pas léger et détendu on lance le crackme sous Olly, bien entendu on trouve l'api **GetWindowTextA**, une routine de vérification et des messages d'erreurs :

00401317 00401319 0040131C 0040131D 00401323 00401328	. 6A 14 . 8D5D E0 . 53 . FF35 04404000 . E8 980E0000 . C745 DC 000000	PUSH 14 LEA EBX,DWORD PTR SS:[EBP-20] PUSH EBX PUSH DWORD PTR DS:[404004] CALL <JMP.&USER32.GetWindowTextA> MOV DWORD PTR SS:[EBP-24],0	Count := 14 (20.) Buffer hWnd = 000E0604 ('Nom...',class='Edit',parent=001C05A4) GetWindowTextA
00401381 00401383 00401385 0040138A 00401390 00401395 00401397 0040139A 0040139C 0040139E 004013A3 004013A8 004013AE 004013B3 004013B5 004013B7	> 6A 00 . 6A 00 . 68 EA030000 . FF35 00404000 . E8 330E0000 . 89C3 . 3B5D DC ✓ 74 1B . 6A 30 . 68 11124000 . 68 30124000 . FF35 00404000 . E8 1D0E0000 . B0 00 ✓ EB 02 > B0 01	PUSH 0 PUSH 0 PUSH 3EA PUSH DWORD PTR DS:[404004] CALL <JMP.&USER32.GetDlgItemInt> MOV EBX,EAX CMP EBX,DWORD PTR SS:[EBP-24] JE SHORT DBKeyGen.004013B7 PUSH 30 PUSH DBKeyGen.00401211 PUSH DBKeyGen.00401230 PUSH DWORD PTR DS:[404004] CALL <JMP.&USER32.MessageBoxA> MOV AL,0 JMP SHORT DBKeyGen.004013B9 MOV AL,1	IsSigned = FALSE pSuccess = NULL ControlID = 3EA (1002.) hWnd = 001C05A4 ('Crack_Me_By_Kharneth',class='WindowsApp') GetDlgItemInt Style = MB_OK MB_ICONEXCLAMATION MB_APPLMODAL Title = "Numéro de série incorrect!" Text = "Le numéro de série que vous avez entré ne correspond pas." hOwner = 001C05A4 ('Crack_Me_By_Kharneth',class='WindowsApp') MessageBoxA
00401A65 00401A67 00401A6D 00401A6F 00401A74 00401A79 00401A7B	. 84C0 ✓ 0F84 E8020000 . 6A 10 . 68 30194000 . 68 50194000 . 6A 00 . E8 50070000	TEST AL,AL JE DBKeyGen.00401D55 PUSH 10 PUSH DBKeyGen.00401930 PUSH DBKeyGen.00401950 PUSH 0 CALL <JMP.&USER32.MessageBoxA>	Style = MB_OK MB_ICONHAND MB_APPLMODAL Title = "Tentative de piratage détectée!" Text = "Vos noms et adresses viennent d'être envoyés à la base de données." hOwner = NULL MessageBoxA

J'ai volontairement coupé des parties du code car en fait tout ceci ne nous intéresse pas, **Kharneth** nous avait signalé la présence d'un leurre, le voici. Quelque soit le couple nom/serial inséré, on aura toujours un message négatif. Je vous laisse tracer le code pour vous en convaincre. Mais comment sont donc obtenues les informations d'enregistrement...

On va jeter un œil dans les imports (la petite case bleue notée **R** en haut à droite), et pourquoi pas une petit **SendMessage** ? Effectivement la solution va se situer de ce côté, plus précisément avec l'utilisation de messages **WM_GETTEXT**. Voici le code :

<pre> 00401EA9 . 50 PUSH EAX 00401EAA . 6A 23 PUSH 23 00401EAC . 6A 00 PUSH 00 00401EAE . FF35 00404000 PUSH DWORD PTR DS:[404000] 00401EB4 . E8 47030000 CALL <JMP.&USER32.SendMessageA> 00401EB9 . 8BC3 MOV BL,AL 00401EBB . 68 31304000 PUSH DBKeyGen.00403031 00401EC0 . 6A 0A PUSH 0A 00401EC2 . 6A 00 PUSH 00 00401EC4 . FF35 04404000 PUSH DWORD PTR DS:[404004] 00401ECA . E8 31030000 CALL <JMP.&USER32.SendMessageA> 00401ECF . 83C4 10 ADD ESP,10 00401ED2 . A8 01 TEST AL,1 00401ED4 . 74 02 JE SHORT DBKeyGen.00401ED8 00401ED6 . FEC0 INC AL 00401ED8 . 0FB6F3 MOVZX ESI,BL 00401EDB . 0FB6D8 MOVZX EBX,AL 00401EDE . 8D0C5B LEA ECX,DWORD PTR DS:[EBX+EBX*2] 00401EE1 . C0E8 01 SHR AL,1 00401EE4 . 25 FF000000 AND EAX,0FF 00401EE9 . 8D4C08 FF LEA ECX,DWORD PTR DS:[EAX+ECX-1] 00401EED . 89DF MOV EDI,EBX 00401EEF . 39CE CMP ESI,ECX 00401EF1 . 75 7B JNZ SHORT DBKeyGen.00401F6E 00401EF3 . 8B55 C4 MOV EDX,DWORD PTR SS:[EBP-3C] 00401EF6 . 8955 CC MOV DWORD PTR SS:[EBP-34],EDX 00401EF9 . C645 CA 00 MOV BYTE PTR SS:[EBP-36],0 00401EFD . 31F6 XOR ESI,ESI 00401EFF . 39FE CMP ESI,EDI 00401F01 . 7D 6B JGE SHORT DBKeyGen.00401F6E </pre>	<pre> iParam = 23 wParam = 23 Message = WM_GETTEXT hWnd = F05C4 SendMessage iParam = 403031 wParam = A Message = WM_GETTEXT hWnd = E0604 SendMessage Test si le nombre est pair ou non Si il est impair on l'incrémente (il devient pair) Longueur serial dans ESI ECX = (longueur name "+1") * 3 EAX = (longueur name "+1") / 2 Compare l'expression precedente et la longueur du serial C'est c'est pas bon => EXIT buffer du serial On le met sur la pile On initialise cette variable a zero Le name ne peut etre nul </pre>
--	---

Que faut-il retenir ? Principalement le calcul de la taille du serial. Un point intéressant, le calcul dépend de la parité de la longueur de nom. Si cette longueur est impaire, on l'incrémente de manière à se retrouver avec un nombre pair. Cela nous donne alors **[longueur serial] = [longueur nom]*(7/2) – 1**. A ce niveau rien de plus à dire sinon que le nom ne peut être nul, la suite deviendrait évidente.

On peut maintenant s'intéresser à l'algorithme à proprement parler, un point spécifique, sa complexité dépend partiellement de la parité de la longueur du nom, une longueur impaire le rend très facile à reverser. Mais avant de commencer je vais expliquer un peu le fonctionnement de **strtol**, voici un extrait de la **MSDN** :

Convert strings to an unsigned long-integer value.

```

unsigned long strtoul(
    const char *nptr,
    char **endptr,
    int base
);

```

strtoul expects *nptr* to point to a string of the following form:

[whitespace] [{+ | -}] [0 [{ x | X }]] [digits]

A *whitespace* may consist of space and tab characters, which are ignored; *digits* are one or more decimal digits. The first character that does not fit this form stops the scan. If *base* is between 2 and 36, then it is used as the base of the number.

Pour résumer cette fonction analyse la string qu'on lui présente et retourne dans **eax**, la valeur du nombre représenté (dans la base indiquée), la string est parcouru jusqu'à la découverte d'un espace. On peut donc représenter plusieurs nombres séparés par des espaces dans une même string.

Maintenant le code de l'algorithme en lui-même :

00401F22	> 83C4 FC	ADD ESP,-4	
00401F06	. 6A 10	PUSH 10	
00401F08	. 8D45 CC	LEA EAX,DWORD PTR SS:[EBP-34]	
00401F0B	. 50	PUSH EAX	
00401F0C	. FF75 CC	PUSH DWORD PTR SS:[EBP-34]	
00401F0F	. E8 54020000	CALL <JMP.&msvcrt.strtoul>	
00401F14	. 83C4 10	ADD ESP,10	
00401F17	. 85C0	TEST EAX,EAX	On recupere dans EAX
00401F19	. 75 07	JNZ SHORT DBKeyGen.00401F22	la valeur en hexa du premier bloc
00401F1B	. B8 01000000	MOV EAX,1	
00401F20	. EB 50	JMP SHORT DBKeyGen.00401F72	
00401F22	> FF45 CC	INC DWORD PTR SS:[EBP-34]	
00401F25	. 8BC1	MOV CL,AL	On deplace la partie basse dans CL
00401F27	. C1E8 08	SHR EAX,8	On prend la partie haute dans AL
00401F2A	. 8B45 CB	MOV BYTE PTR SS:[EBP-35],AL	On la stocke ici
00401F2D	. C1E8 08	SHR EAX,8	Il reste quelque chose ?
00401F30	. 25 FF000000	AND EAX,0FF	(on aura interet a avoir AL = 0 a cet endroit
00401F35	. 8D0440	LEA EAX,DWORD PTR DS:[EAX+EAX*2]	EAX * 3
00401F38	. C1E0 04	SHL EAX,4	EAX * 16 (d)
00401F3B	. 0FB6D1	MOVZX EDX,CL	
00401F3E	. 8955 C0	MOV DWORD PTR SS:[EBP-40],EDX	
00401F41	. 99	CDQ	
00401F42	. F77D C0	IDIV DWORD PTR SS:[EBP-40]	
00401F45	. 8BC1	MOV CL,AL	
00401F47	. 0FB645 CB	MOVZX EAX,BYTE PTR SS:[EBP-35]	partie haute
00401F4B	. 8D0440	LEA EAX,DWORD PTR DS:[EAX+EAX*2]	EAX * 3 * 16 (d)
00401F4E	. C1E0 04	SHL EAX,4	
00401F51	. 99	CDQ	
00401F52	. F77D C0	IDIV DWORD PTR SS:[EBP-40]	on divise par la partie basse
00401F55	. 328E 31304000	XOR CL,BYTE PTR DS:[ESI+403031]	XOR CL, byte (n)
00401F58	. 004D CA	ADD BYTE PTR SS:[EBP-36],CL	On additione le resultat ici
00401F5E	. 328E 32304000	XOR AL,BYTE PTR DS:[ESI+403032]	XOR CL, byte (n+1)
00401F64	. 0045 CA	ADD BYTE PTR SS:[EBP-36],AL	Et ici (sur la meme byte)
00401F67	. 83C6 02	ADD ESI,2	On utilise les bytes du name 2 par 2
00401F6A	. 39FE	CMP ESI,EDI	Nous sommes a la fin ?
00401F6E	> 7C 95	JL SHORT DBKeyGen.00401F03	
00401F72	> 0FB645 CA	MOVZX EAX,BYTE PTR SS:[EBP-36]	Nous verrons plus tard que
00401F75	. 8D65 A8	LEA ESP,DWORD PTR SS:[EBP-58]	cette byte doit etre egale a zero
00401F76	. 5B	POP EBX	
00401F77	. 5E	POP ESI	
00401F78	. 5F	POP EDI	
00401F79	. C9	LEAVE	
00401F79	. C3	RETN	

Concrètement que se passe-t-il ? Notre serial devra être composé d'autant de blocs qu'il y a de paires de lettres dans notre nom (n'oubliez pas l'incrémentation de la longueur fonction de la parité), tous séparés par des espaces. Pour chaque bloc on va agir sur les bytes correspondantes n et n+1 du nom. Pour simplifier pour la suite il serait bon de prendre des blocs de la forme 00xxxx, d'une manière à dans un premier temps uniformiser tous les blocs pour écrire un keygen plus simple puis aussi à récupérer à chaque fois un **dword** où l'on pourra différencier une partie haute et une partie basse soit 00hhbb. Une précision supplémentaire mais loin d'être superflue, un bloc ne peut être nul, on peut donc déjà oublier le bloc 000000.

Pour chaque paire voici le principe : la partie haute du **dword** est multipliée par 30h puis divisée par la partie basse. On **XOR** le quotient avec la byte n et le reste avec la byte n+1 et on additionne chacun des résultats sur une byte : **[EBP-36]**. Pour l'instant on ne le voit pas mais plus tard on apprendra qu'il nous faut avoir cette byte égale à zéro.

Maintenant que l'on a une idée du fonctionnement du crackme voici comment le résoudre, il y a plusieurs méthodes, on va essayer de faire simple :

- **Le nom est d'une longueur impair** : si on prend pour tous les blocs sauf le dernier, une valeur de la forme 0000xx, xx différent de zéro, on va réaliser une simple somme des lettres du nom (**XOR** avec 0 puis **ADD**), la byte **[EBP-36]** sera donc la « partie basse » de cette somme. Et pour avoir au final zéro, on « ajuste » avec le dernier bloc. On prend la partie basse du dernier bloc égale à 30h (pour annuler la multiplication de la partie haute). Ainsi la partie haute que l'on XOR normalement avec la byte n+1 sera ici **XORée** avec 0 et donc sera une

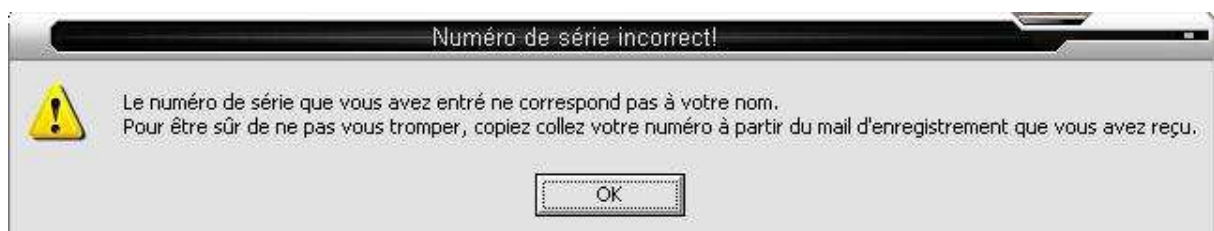
addition directe, qui nous permettra de faire que la somme des lettres soit égales à 00h sur **[EBP-36]** (peu importe ce qu'il y a dans les bytes adjacentes, c'est-à-dire si la somme dépasse FFh). L'intérêt du fait que la longueur du nom soit impair est que la byte n+1 que l'on va **XORer** avec la partie haute sera toujours 00h et l'on aura donc toute liberté d'action (de 00h à FFh) sur **[EBP-36]**, chose que l'on ne retrouve pas avec une longueur paire.

- **Le nom est d'une longueur pair** : la méthode sera sensiblement similaire à ceci près que la compensation finale devra se faire par le résultat du **XOR** entre la partie haute du dernier **dword** et la dernière lettre du nom.

Voilà je crois qu'on a fait le tour du crackme, on a toutes les clés en main pour le keygenner et si tout ce passe bien on doit se retrouver avec un crackme ressemblant à ceci :



Il est à souligner tout de même que l'on peut considérer le bouton « valider » comme le leurre en lui-même, il ne sert à rien si ce n'est à nous afficher un message négatif de la forme:



Conclusion : un crackme sympathique qu'il vaut mieux prendre par le bon bout pour ne pas risquer le pire. Bien analyser l'algorithme est aussi nécessaire si on ne veut pas se compliquer outre mesure les opérations pour le keygen. Vous trouverez le mien dans l'archive, codé avec MASM.

Greetz : **Kharneth** pour son crackme, Eedy31, meat, Neitsa et tous les FRETiens puis à tous ceux que je connais.

A bientôt. Requiem [FRET].