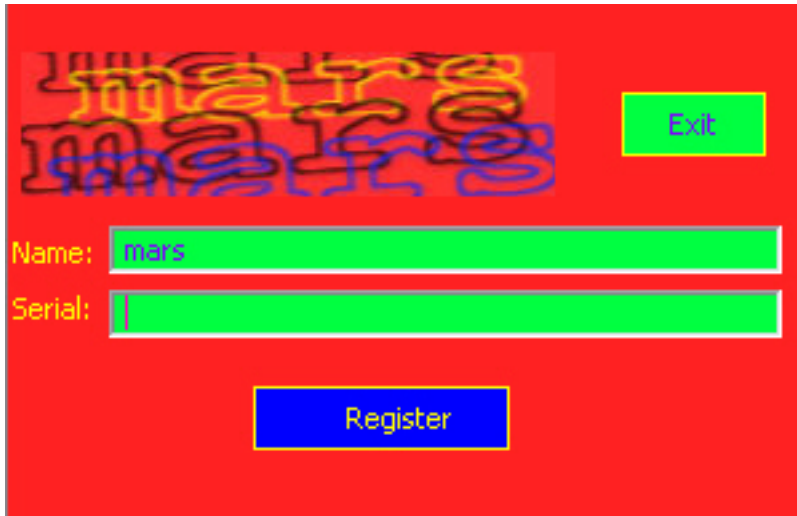


Tutorial 17 – Crackme 5 by mars

Commençons par lancer ce crackme (non packé et codé en asm) assez flashy :)



Rentrons un serial : il ne se passe rien. Il en est de même en cliquant sur Register...
Il est tant d'ouvrir la bête sous Ollydbg, pour voir ce qu'elle renferme.

On va d'abord regarder les différentes informations exploitables, rien qu'en regardant le listing asm :
– Le crackme contient très peu de SDRs.
La seule intéressante est celle-ci :

00401235	> 6A 40	PUSH 40	[Style = MB_OK MB_ICONASTERISK MB_APPLMODAL Title = "Information" Text = "The serial is invalid" hOwner MessageBoxA ExitProcess]
00401237	. 68 B4604000	PUSH def_i_mar.004060B4	
0040123C	. 68 84604000	PUSH def_i_mar.00406084	
00401241	. FF75 08	PUSH DWORD PTR SS:[EBP+8]	
00401244	. E8 79070000	CALL <JMP.&user32.MessageBoxA>	
00401249	. E8 B6070000	CALL <JMP.&kernel32.ExitProcess>	

– En regardant attentivement la table des jmp [API], on trouve des informations très intéressantes et notamment ceci :

0040195C	\$-FF25 CC504000	JMP DWORD PTR DS:[<user32.CallWindowProcA>]	user32.CallWindowProcA
00401962	\$-FF25 D0504000	JMP DWORD PTR DS:[<user32.DialogBoxParamA>]	user32.DialogBoxParamA
00401968	\$-FF25 D4504000	JMP DWORD PTR DS:[<user32.DrawFocusRect>]	user32.DrawFocusRect
0040196E	\$-FF25 D8504000	JMP DWORD PTR DS:[<user32.DrawTextA>]	user32.DrawTextA
00401974	\$-FF25 DC504000	JMP DWORD PTR DS:[<user32.EnableWindow>]	user32.EnableWindow
0040197A	\$-FF25 E0504000	JMP DWORD PTR DS:[<user32.EndDialog>]	user32.EndDialog
00401980	\$-FF25 E4504000	JMP DWORD PTR DS:[<user32.FillRect>]	user32.FillRect
00401986	\$-FF25 E8504000	JMP DWORD PTR DS:[<user32.GetClientRect>]	user32.GetClientRect
0040198C	\$-FF25 EC504000	JMP DWORD PTR DS:[<user32.GetDlgCtrlID>]	user32.GetDlgCtrlID
00401992	\$-FF25 C4504000	JMP DWORD PTR DS:[<user32.GetDlgItem>]	user32.GetDlgItem
00401998	\$-FF25 C0504000	JMP DWORD PTR DS:[<user32.GetDlgItemTextA>]	user32.GetDlgItemTextA
0040199E	\$-FF25 BC504000	JMP DWORD PTR DS:[<user32.GetForegroundWindow>]	user32.GetForegroundWindow
004019A4	\$-FF25 B8504000	JMP DWORD PTR DS:[<user32.GetMenu>]	user32.GetMenu
004019AA	\$-FF25 B4504000	JMP DWORD PTR DS:[<user32.GetParent>]	user32.GetParent
004019B0	\$-FF25 B0504000	JMP DWORD PTR DS:[<user32.InflateRect>]	user32.InflateRect
004019B6	\$-FF25 AC504000	JMP DWORD PTR DS:[<user32.IsWindowEnabled>]	user32.IsWindowEnabled
004019BC	\$-FF25 A8504000	JMP DWORD PTR DS:[<user32.LoadIconA>]	user32.LoadIconA
004019C2	\$-FF25 A4504000	JMP DWORD PTR DS:[<user32.MessageBoxA>]	user32.MessageBoxA
004019C8	\$-FF25 A0504000	JMP DWORD PTR DS:[<user32.OffsetRect>]	user32.OffsetRect
004019CE	\$-FF25 9C504000	JMP DWORD PTR DS:[<user32.PostMessageA>]	user32.PostMessageA
004019D4	\$-FF25 98504000	JMP DWORD PTR DS:[<user32.SendDlgItemMessageA>]	user32.SendDlgItemMessageA
004019DA	\$-FF25 94504000	JMP DWORD PTR DS:[<user32.SendMessageA>]	user32.SendMessageA
004019E0	\$-FF25 90504000	JMP DWORD PTR DS:[<user32.SetDlgItemTextA>]	user32.SetDlgItemTextA
004019E6	\$-FF25 8C504000	JMP DWORD PTR DS:[<user32.SetFocus>]	user32.SetFocus
004019EC	\$-FF25 88504000	JMP DWORD PTR DS:[<user32.SetWindowLongA>]	user32.SetWindowLongA
004019F2	\$-FF25 84504000	JMP DWORD PTR DS:[<user32.SetWindowRgn>]	user32.SetWindowRgn
004019F8	\$-FF25 80504000	JMP DWORD PTR DS:[<user32.SetWindowTextA>]	user32.SetWindowTextA
DS:[004050C0]=77D6AC06 (user32.GetDlgItemTextA)			
Local calls from 004010B9, 004011D1, 004011E7, 0040146E			

Nous y reviendrons plus tard.

Lançons maintenant le crackme. On s'aperçoit que Ollydbg et le crackme se ferment assez vite :(. En regardant toujours la même table de jmp [API], on détecte 2 APIs suspectes :

004019FE	\$-FF25 7C504000	JMP DWORD PTR DS:[<user32.keybd_event>]	user32.keybd_event
00401A04	\$-FF25 68504000	JMP DWORD PTR DS:[<kernel32.ExitProcess>]	kernel32.ExitProcess
00401A0A	\$-FF25 6C504000	JMP DWORD PTR DS:[<kernel32.FindResourceA>]	kernel32.FindResourceA
00401A10	\$-FF25 70504000	JMP DWORD PTR DS:[<kernel32.GetModuleHandleA>]	kernel32.GetModuleHandleA
00401A16	\$-FF25 74504000	JMP DWORD PTR DS:[<kernel32.GlobalAlloc>]	kernel32.GlobalAlloc
00401A1C	\$-FF25 64504000	JMP DWORD PTR DS:[<kernel32.GlobalFree>]	kernel32.GlobalFree
00401A22	\$-FF25 60504000	JMP DWORD PTR DS:[<kernel32.LoadResource>]	kernel32.LoadResource
00401A28	\$-FF25 5C504000	JMP DWORD PTR DS:[<kernel32.LockResource>]	kernel32.LockResource
00401A2E	\$-FF25 50504000	JMP DWORD PTR DS:[<kernel32.SetUnhandledExceptionFilter>]	kernel32.SetUnhandledExceptionFilter
00401A34	\$-FF25 54504000	JMP DWORD PTR DS:[<kernel32.SizeofResource>]	kernel32.SizeofResource
DS:[0040507C]=77D66365 (user32.keybd_event)			
Local calls from 00401824, 00401831, 0040183E, 0040184B, 00401858, 00401868			

L'API SetUnhandledExceptionFilter ne nous intéresse pas ici, car elle n'est pas responsable de la "détection d'Ollydbg" et de sa fermeture. D'ailleurs, elle ne semble pas être appelée (Pas de "Local calls", en cliquant dessus).

Intéressons nous à L'API keybd_event :

keybd_event Function

The **keybd_event** function synthesizes a keystroke. The system can use such a synthesized keystroke to generate a [WM_KEYUP](#) or [WM_KEYDOWN](#) message. The keyboard driver's interrupt handler calls the **keybd_event** function.

Windows NT/2000/XP/Vista: This function has been superseded. Use [SendInput](#) instead.

Syntax

VOID keybd_event(BYTE *bVk*, BYTE *bScan*, DWORD *dwFlags*, PTR *dwExtraInfo*)

Parameters

bVk

[in] Specifies a virtual-key code. The code must be a value in the range 1 to 254. For a complete list, see [Virtual-Key Codes](#).

bScan

Specifies a hardware scan code for the key.

dwFlags

[in] Specifies various aspects of function operation. This parameter can be one or more of the following values.

KEYEVENTF_EXTENDEDKEY

If specified, the scan code was preceded by a prefix byte having the value 0xE0 (224).

KEYEVENTF_KEYUP

If specified, the key is being released. If not specified, the key is being depressed

dwExtraInfo

[in] Specifies an additional value associated with the key stroke

Return Value

This function has no return value.

Remarks:

An application can simulate a press of the PRINTSCRN key in order to obtain a screen snapshot and save it to the clipboard. To do this, call **keybd_event** with the *bVk* parameter set to VK_SNAPSHOT.

Windows NT/2000/XP: The **keybd_event** function can toggle the NUM LOCK, CAPS LOCK, and SCROLL LOCK keys.

Windows 95/98/Me: The **keybd_event** function can toggle only the CAPS LOCK and SCROLL LOCK keys. It cannot toggle the NUM LOCK key.

The following sample program toggles the NUM LOCK light by using **keybd_event()** with a virtual key of VK_NUMLOCK. It takes a Boolean value that indicates whether the light should be turned off (FALSE) or on (TRUE). The same technique can be used for the CAPS LOCK key (VK_CAPITAL) and the SCROLL LOCK key (VK_SCROLL).

Cette API permet donc de simuler la frappe de touches.

00401811	> 83FA 64	CMP EDX,64	
00401814	. 90	NOP	
00401815	. 90	NOP	
00401816	. 90	NOP	
00401817	√ 7E 54	JLE SHORT defi_mar.0040186D	
00401819	. 6A 00	PUSH 0	ExtraInfo = 0
0040181B	. 6A 00	PUSH 0	Flags = 0
0040181D	. 68 B8000000	PUSH 0B8	ScanCode = B8 (184.)
00401822	. 6A 12	PUSH 12	Key = VK_MENU
00401824	. E8 D5010000	CALL <JMP.&user32.keybd_event>	keybd_event
00401829	. 6A 00	PUSH 0	ExtraInfo = 0
0040182B	. 6A 00	PUSH 0	Flags = 0
0040182D	. 6A 00	PUSH 0	ScanCode = 0
0040182F	. 6A 4B	PUSH 4B	Key = 4B ('K')
00401831	. E8 C8010000	CALL <JMP.&user32.keybd_event>	keybd_event
00401836	. 6A 00	PUSH 0	ExtraInfo = 0
00401838	. 6A 00	PUSH 0	Flags = 0
0040183A	. 6A 00	PUSH 0	ScanCode = 0
0040183C	. 6A 58	PUSH 58	Key = 58 ('X')
0040183E	. E8 BB010000	CALL <JMP.&user32.keybd_event>	keybd_event
00401843	. 6A 00	PUSH 0	ExtraInfo = 0
00401845	. 6A 02	PUSH 2	Flags = KEYEVENTF_KEYUP
00401847	. 6A 00	PUSH 0	ScanCode = 0
00401849	. 6A 58	PUSH 58	Key = 58 ('X')
0040184B	. E8 AE010000	CALL <JMP.&user32.keybd_event>	keybd_event
00401850	. 6A 00	PUSH 0	ExtraInfo = 0
00401852	. 6A 02	PUSH 2	Flags = KEYEVENTF_KEYUP
00401854	. 6A 00	PUSH 0	ScanCode = 0
00401856	. 6A 4B	PUSH 4B	Key = 4B ('K')
00401858	. E8 A1010000	CALL <JMP.&user32.keybd_event>	keybd_event
0040185D	. 6A 00	PUSH 0	ExtraInfo = 0
0040185F	. 6A 02	PUSH 2	Flags = KEYEVENTF_KEYUP
00401861	. 68 B8000000	PUSH 0B8	ScanCode = B8 (184.)
00401866	. 6A 12	PUSH 12	Key = VK_MENU
00401868	. E8 91010000	CALL <JMP.&user32.keybd_event>	keybd_event
0040186D	> 33C0	XOR EAX,EAX	
0040186F	. C9	LEAVE	
00401870	. C2 1000	RETN 10	

Sachant que le virtual-key code VK_MENU (0x12) correspond à la touche ALT et que le flag KEYEVENTF_KEYUP correspond au "relâchement" d'une touche, on comprend alors qu'un ALT+X est simulé. Il suffit d'aller voir dans le menu 'File' d'Olydbg, pour se rendre compte que cette combinaison sert à quitter le debugger en question, fermeture, qui induit alors celle de l'application. Par contre, la simulation du ALT+K est inutile (puisque'elle ne sert qu'à afficher la "Call stack").

Un saut inconditionnel (jmp) en 00401817 et c'en est fini de cet anti-ollydbg spécial :p.

1ère étape : vérification du serial :

Maintenant, on peut étudier l'algorithme de vérification de serial :).

Nous avons vu que l'API GetDlgItemTextA est appelée à 4 endroits dans le code.

Pour nous faciliter la tâche, on va récupérer l'ID des différents contrôles. Pour cela, on peut utiliser des outils spécifiques comme Window Hack 3.0, l'analyse du code, etc... Je ne m'étendrai pas dessus, car il existe 36000 moyens de les obtenir...

00401075	. 6A 00	PUSH 0	lParam = 0
00401077	. 6A 1F	PUSH 1F	wParam = 1F
00401079	. 68 C5000000	PUSH 0C5	Message = EM_LIMITTEXT
0040107E	. 68 94010000	PUSH 194	ControlID = 194 (404.)
00401083	. FF75 08	PUSH DWORD PTR SS:[EBP+8]	hWnd
00401086	. E8 49090000	CALL <JMP.&user32.SendDlgItemMessageA>	SendDlgItemMessageA
0040108B	. 68 60604000	PUSH defi_mar.00406060	Text = "mars"
00401090	. 68 95010000	PUSH 195	ControlID = 195 (405.)
00401095	. FF75 08	PUSH DWORD PTR SS:[EBP+8]	hWnd
00401098	. E8 43090000	CALL <JMP.&user32.SetDlgItemTextA>	SetDlgItemTextA

L'ID du contrôle du champ "serial" est : 0x194.

Celui du champ "name" est : 0x195.

Enfin, celui du bouton "Register" est : 0x197.

On va mettre un bp au niveau de ce code, chargé de récupérer le champ "serial" (l' ID Control est 0x194) :

004010A4	\$ 55	PUSH EBP	
004010A5	. 8BEC	MOV EBP,ESP	
004010A7	. 53	PUSH EBX	
004010A8	. 56	PUSH ESI	
004010A9	. 57	PUSH EDI	
004010AA	. 6A 20	PUSH 20	Count = 20 (32.)
004010AC	. 68 28624000	PUSH defi_mar.00406228	Buffer = defi_mar.00406228
004010B1	. 68 94010000	PUSH 194	ControlID = 194 (404.)
004010B6	. FF75 08	PUSH DWORD PTR SS:[EBP+8]	hWnd
004010B9	. E8 DA080000	CALL <JMP.&user32.GetDlgItemTextA>	GetDlgItemTextA

La suite du code est commentée :

004010AA	. 6A 20	PUSH 20	Count = 20 (32.)
004010AC	. 68 28624000	PUSH def_mar.00406228	Buffer = def_mar.00406228
004010B1	. 68 94010000	PUSH 194	ControlID = 194 (404.)
004010B6	. FF75 08	PUSH DWORD PTR SS:[EBP+8]	hWnd
004010B9	. E8 DA080000	CALL <JMP.>user32.GetDlgItemTextA	GetDlgItemTextA
004010BE	. 83F8 04	CMP EAX,4	; y a-t-il 4 caractères de récupérés ?
004010C1	. 0F8C B6000000	JL def_mar.0040117D	; si c'est inférieur, on quitte
004010C7	. 33C9	XOR ECX,ECX	; mise à 0 du compteur
004010C9	. 8BC8	MOV ECX,EAX	; eax est la longueur de la chaîne récupérée
004010CB	. 8BD0	MOV EDX,EAX	; on affecte à edx la valeur de eax
004010CD	. 33C0	XOR EAX,EAX	; mise à 0 de eax
004010CF	> BA 10000000	MOV EDX,10	; edx = 0x10
004010D4	. 49	DEC ECX	; on décrémente le compteur
004010D5	. 8A81 28624000	MOV AL,BYTE PTR DS:[ECX+406228]	; on prend le nieme caractère du Buffer (par la fin)
004010DB	. 3C 30	CMP AL,30	; on compare au caractère 0
004010DD	. 0F8E 9A000000	JLE def_mar.0040117D	; on quitte, si c'est inférieur ou égal
004010E3	. 3C 39	CMP AL,39	; on compare au caractère 9
004010E5	. 0F8F 92000000	JG def_mar.0040117D	; on quitte, si c'est supérieur
004010EB	. 2C 30	SUB AL,30	; on lui soustrait la valeur ascii du caractère 0
004010ED	. F7E2	MUL EDX	; on multiplie par 0x10
004010EF	. 8881 A8624000	MOV BYTE PTR DS:[ECX+4062A8],AL	; on sauve dans un autre Buffer
004010F5	. 84C9	TEST CL,CL	; il reste des caractères à traiter ?
004010F7	. ^75 D6	JNZ SHORT def_mar.004010CF	; si oui, on boucle
004010F9	. 33C0	XOR EAX,EAX	; on efface la valeur de eax
004010FB	. 66:A1 A8624000	MOV AX,WORD PTR DS:[4062A8]	
00401101	. 33C9	XOR ECX,ECX	; on efface la valeur de ecx
00401103	. 66:8B0D AA6240	MOV CX,WORD PTR DS:[4062AA]	
0040110A	. 3BC1	CMP EAX,ECX	; on compare eax à ecx
0040110C	. ^7C 6F	JL SHORT def_mar.0040117D	; on quitte si eax est inférieur à ecx
0040110E	. 66:3BC1	CMP AX,CX	; on compare ax à cx
00401111	. ^79 6A	JNS SHORT def_mar.0040117D	; on quitte si la soustraction n'est pas signée
00401113	. ^EB 00	JMP SHORT def_mar.00401115	; on continue :)
00401115=def_mar.00401115			
Address	Hex dump	ASCII	
00406228	35 39 32 31 00 00 00 00 00 00 00 00 00 00 00 00	5921.....	0012F984 0012FA14
00406238	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	0012F988 004012C7
00406248	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	0012F98C 00000000
00406258	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	0012F990 0012F9AC
00406268	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	0012F994 0040151F
00406278	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	0012F998 000B07C4
00406288	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	0012F99C 7C91EE18
00406298	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	0012F9A0 7C920570
004062A8	50 90 20 10 00 00 00 00 00 00 00 00 00 00 00 00	PO	0012F9A4 FFFFFFFF
004062B8	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	0012F9A8 7C92056D
004062C8	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	0012F9AC 0012F9D8
			0012F9B0 77D18709

Pour l'instant, le crackme ne se préoccupe pas du champ "nom", donc j'ai laissé le nom "mars".

1ère condition : les caractères du serial doivent être numériques (0 exclus).

Pour le serial, prenons l'exemple de 5921.

Suite à la soustraction et la multiplication, le Buffer en 004062A8 se remplit des valeurs hexadécimales : 50 90 20 10.

Note : Toutes les valeurs, qui suivront, seront toujours hexadécimales...

Par l'instruction en 004010FB, 9050 est affecté à ax.

Par l'instruction en 004010FB, 1020 est affecté à cx.

Le saut (conditionnel jl), qui suit le "cmp eax, ecx", ne doit pas être effectué. Ceci implique que eax doit toujours être supérieur à ecx.

Ce test permet de travailler avec des nombres signés. Mais ceci n'a pas d'influence. En effet, le bit de signe de eax / ecx est nul (suite à la remise à zéro de ces registres), ce qui n'est pas forcément le cas pour ax / cx...

Par contre, ceci a de l'influence pour le 2ème test. Le saut conditionnel (jns), suivant le "cmp ax, cx", ne doit pas être effectué. Ceci implique que (cx-ax) doit être signé (négatif). En binaire, (cx-ax) doit être égal ou supérieur à 10000000 (soit 0x8000). En hexadécimal, ax >= cx + 8000 ou cx <= ax - 8000.

2ème condition : le 2ème caractère du serial doit être 9 et le 4ème doit être 1. Le 1er caractère entré doit être plus grand ou égal au 3ème.

On peut passer à la suite. On s'aperçoit qu'une partie du code suivant est crypté...

En voyant cela, je m'étais dit : hum, ça sent le bruteforce, mais finalement non :p.

00401115	> 33C1	XOR EAX,ECX	; on effectue un xor (ou exclusif) entre eax et ecx
00401117	. B9 00100000	MOV ECX,1000	; on affecte 1000 à ecx (diviseur)
0040111C	. F7F1	DIV ECX	; on divise eax par ecx (le quotient est stocké dans al)
0040111E	. 33C9	XOR ECX,ECX	; on efface ecx
00401120	> 3081 2C114000	XOR BYTE PTR DS:[ECX+40112C],AL	; on décrypte le code suivant (la clé étant al)
00401126	. 41	INC ECX	; on incrémente ecx
00401127	. 83F9 51	CMP ECX,51	; décryptage terminé
0040112A	. ^75 F4	JNZ SHORT def_i_mar.00401120	; on boucle tant que ce n'est pas terminé
0040112C	. 88	DB 88	
0040112D	. 3D	DB 3D	CHAR '='
0040112E	. 1C	DB 1C	
0040112F	. 19	DB 19	
00401130	. 48	DB 48	CHAR 'H'
00401131	. 08	DB 08	

Toujours avec mon exemple de 5921 comme serial. Le "xor eax, ecx" nous donne eax = 8070. On le divise par ecx = 1000, on obtient alors al = 08 (le reste, égal à 70 est stocké dans dl). On tombe toujours sur la bonne clé (tant que l'on respecte les 2 conditions précédentes), puisque 90x0 XOR 10y0 (x>=y) donne toujours 80z0, qui divisé par 1000, donne toujours 8 (al=08).

Le code décrypté est alors le suivant :

0040112C	. 8035 14114000	XOR BYTE PTR DS:[401114],1E	; on modifie en 00401113, le saut inconditionnel
00401133	> 8925 00604000	MOV DWORD PTR DS:[406000],ESP	
00401139	. 892D 04604000	MOV DWORD PTR DS:[406004],EBP	
0040113F	. 68 20604000	PUSH def_i_mar.00406020	[pTopLevelFilter = def_i_mar.00406020
00401144	. E8 E5080000	CALL <JMP.&kernel32.SetUnhandledExceptionFilter>	SetUnhandledExceptionFilter
00401149	. CC	INT3	
0040114A	. 68 97010000	PUSH 197	[ControlID = 197 (407.)
0040114F	. FF75 08	PUSH DWORD PTR SS:[EBP+8]	hWnd
00401152	. E8 3B080000	CALL <JMP.&user32.GetDlgItem>	GetDlgItem
00401157	. 6A 01	PUSH 1	[Enable = TRUE
00401159	. 50	PUSH EAX	hWnd
0040115A	. E8 15080000	CALL <JMP.&user32.EnableWindow>	EnableWindow
0040115F	. E8 3A080000	CALL <JMP.&user32.GetForegroundWindow>	GetForegroundWindow
00401164	. 8BF0	MOV ESI,EAX	
00401166	. 51	PUSH ECX	hWnd
00401167	. E8 38080000	CALL <JMP.&user32.GetMenu>	GetMenu
0040116C	. 50	PUSH EAX	[lpParam
0040116D	. 68 61090000	PUSH 961	wParam = 961
00401172	. 68 11010000	PUSH 111	Message = WM_COMMAND
00401177	. 56	PUSH ESI	hWnd
00401178	. E8 51080000	CALL <JMP.&user32.PostMessageA>	PostMessageA
0040117D	> 33C0	XOR EAX,EAX	
0040117F	. 33C9	XOR ECX,ECX	
00401181	. 33D2	XOR EDX,EDX	
00401183	. C705 28624000	MOV DWORD PTR DS:[406228],0	
0040118D	. C705 A8624000	MOV DWORD PTR DS:[4062A8],0	
00401197	. 5F	POP EDI	
00401198	. 5E	POP ESI	
00401199	. 5B	POP EBX	
0040119A	. C9	LEAVE	
0040119E	. C2 0400	RETN 4	

L'instruction en 0040112C modifie en 00401113, le saut inconditionnel, afin d'éviter un décryptage supplémentaire, ce qui nous redonnerait le code d'origine (et dont l'exécution aboutirait à un crash certain). On voit alors notre 2ème anti-debugger (l'utilisation de l'API SetUnhandledExceptionFilter combinée à une exception via un int 03).

SetUnhandledExceptionFilter Function

Enables an application to supersede the top-level exception handler of each thread and process.

After calling this function, if an exception occurs in a process that is not being debugged, and the exception makes it to the unhandled exception filter, that filter will call the exception filter function specified by the *lpTopLevelExceptionFilter* parameter.

LPTOP_LEVEL_EXCEPTION_FILTER WINAPI SetUnhandledExceptionFilter(__in

Parameters

lpTopLevelExceptionFilter

A pointer to a top-level exception filter function that will be called whenever the [UnhandledExceptionFilter](#) function gets control, and the process is not being debugged. A value of NULL for this parameter specifies default handling within **UnhandledExceptionFilter**.

Return Value

The **SetUnhandledExceptionFilter** function returns the address of the previous exception filter established with the function. A NULL return value means that there is no current top-level exception handler.

Après avoir appelé cette API, si une exception se produit dans un processus, qui n'est pas debuggé, la fonction spécifiée par le *pTopLevelExceptionFilter* sera appelée. Si le processus est debuggé, le programme s'arrête... Il existe de nombreux plugins afin de se prémunir de cela (Hide Debugger, Olly Advanced, etc...). On peut alors mettre un bp en 00406020, pour voir, ce qui est exécuté :

00406020	8B25 00604000	MOV ESP,DWORD PTR DS:[406000]
00406026	8B2D 04604000	MOV EBP,DWORD PTR DS:[406004]
0040602C	803D 2C624000 35	CMP BYTE PTR DS:[40622C],35
00406033	-0F84 11B1FFFF	JE defi_mar.0040114A
00406039	-E9 21B1FFFF	JMP defi_mar.0040115F

L'instruction en 0040602C est une condition supplémentaire...

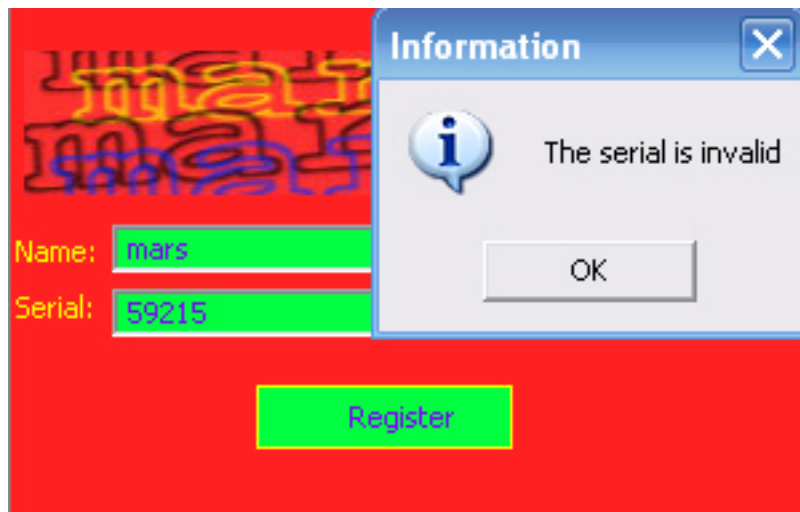
3ème condition : Le serial doit comporter un 5ème caractère. La seule possibilité acceptée est le caractère "5".

Une fois ces conditions validées, le bouton "Register" est alors validé, via ces instructions (en retour de la précédente routine) :

0040193B	\$ 55	PUSH EBP	
0040193C	. 8BEC	MOV EBP,ESP	
0040193E	. FF75 0C	PUSH DWORD PTR SS:[EBP+C]	ControlID
00401941	. FF75 08	PUSH DWORD PTR SS:[EBP+8]	hWnd
00401944	. E8 49000000	CALL <JMP.&user32.GetDlgItem>	GetDlgItem
00401949	. FF75 10	PUSH DWORD PTR SS:[EBP+10]	Enable
0040194C	. 50	PUSH EAX	hWnd
0040194D	. E8 22000000	CALL <JMP.&user32.EnableWindow>	EnableWindow
00401952	. C9	LEAVE	
00401953	. C2 0C00	RETN 0C	

2ème étape : vérification du couple name / serial :

On appuie alors sur le bouton "Register", une MessageBox nous informe que le serial est invalide, puis le programme se ferme.



Cette fois-ci, on se tourne du côté du code faisant référence à la SDR "The serial is invalid".

004011C2	> 6A 60	PUSH 60	Count = 60 (96.)
004011C4	. 68 48624000	PUSH defi_mar.00406248	Buffer = defi_mar.00406248
004011C9	. 68 95010000	PUSH 195	ControlID = 195 (405.)
004011CE	. FF75 08	PUSH DWORD PTR SS:[EBP+8]	hWnd
004011D1	. E8 C2070000	CALL <JMP.&user32.GetDlgItemTextA>	GetDlgItemTextA
004011D6	. 7B 77	JMP SHORT defi_mar.0040124F	
004011D8	> 6A 20	PUSH 20	Count = 20 (32.)
004011DA	. 68 2A624000	PUSH defi_mar.0040622A	Buffer = defi_mar.0040622A
004011DF	. 68 94010000	PUSH 194	ControlID = 194 (404.)
004011E4	. FF75 08	PUSH DWORD PTR SS:[EBP+8]	hWnd
004011E7	. E8 AC070000	CALL <JMP.&user32.GetDlgItemTextA>	GetDlgItemTextA
004011EC	. 803D 48624000	CMP BYTE PTR DS:[406248],0	
004011F3	. 7D 40	JGE SHORT defi_mar.00401235	
004011F5	. 68 08604000	PUSH defi_mar.00406008	SE handler installation
004011FA	. 64:FF35 000000	PUSH DWORD PTR FS:[0]	
00401201	. 64:8925 000000	MOV DWORD PTR FS:[0],ESP	
00401208	. 33C9	XOR ECX,ECX	
0040120A	. 33D2	XOR EDX,EDX	
0040120C	. 66:8B0D 2F6240	MOV CX,WORD PTR DS:[40622F]	
00401213	. 66:8B15 4A6240	MOV DX,WORD PTR DS:[40624A]	
0040121A	. 66:03CA	ADD CX,DX	
0040121D	. 81C1 25250000	ADD ECX,2525	
00401223	. B8 00000010	MOV EAX,10000000	
00401228	. F6F1	DIV CL	
0040122A	. 90	NOP	
0040122E	. 64:8F05 000000	POP DWORD PTR FS:[0]	
00401232	. 83C4 04	ADD ESP,4	
00401235	> 6A 40	PUSH 40	Style = MB_OK MB_ICONASTERISK MB_APPLMODAL
00401237	. 68 B4604000	PUSH defi_mar.004060B4	Title = "Information"
0040123C	. 68 84604000	PUSH defi_mar.00406084	Text = "The serial is invalid"
00401241	. FF75 08	PUSH DWORD PTR SS:[EBP+8]	hOwner
00401244	. E8 79070000	CALL <JMP.&user32.MessageBoxA>	MessageBoxA
00401249	. E8 B6070000	CALL <JMP.&kernel32.ExitProcess>	ExitProcess
0040124E	. 90	NOP	
0040124F	> A0 48624000	MOV AL,BYTE PTR DS:[406248]	
00401254	. 83F8 7A	CMP EAX,7A	
00401257	. 7F DC	JG SHORT defi_mar.00401235	
00401259	. E9 7AFFFFFF	JMP defi_mar.004011D8	

On met donc en bp dans les parages.

Toujours avec le couple mars/59215, on clique sur "Register" et l'on tombe sur une condition supplémentaire...

0040124F	> AO 48624000	MOV AL,BYTE PTR DS:[406248]
00401254	. 83F8 7A	CMP EAX,7A
00401257	. ^7F DC	JG SHORT defi_mar.00401235
00401259	. ^E9 7AFFFFFF	JMP defi_mar.004011D8

Par contre, en continuant de tracer, on obtient encore une condition, en apparence contradictoire o_O :

[illegible][illegible]

Nous pouvons passer à la dernière condition à remplir, afin de valider complètement le crackme.
On utilisera le couple mars/59215ab012345678901234567890123€ .
Toute suite, on voit la mise en place d'un SEH (via le classique push dword ptr fs:[0] / push offset_SE_handler).

Si une exception se produit, l'application la gère et appelle la fonction en 00406008.
Voyons ce qu'il y a en 00406008 :

00406008	6A 00	PUSH 0	Structured exception handler
0040600A	68 57604000	PUSH defi_mar.00406057	ASCII "keygenme"
0040600F	68 94604000	PUSH defi_mar.00406094	ASCII "valid"
00406014	6A 00	PUSH 0	
00406016	E8 A7B9FFFF	CALL <JMP.&user32.MessageBoxA>	
0040601B	E8 826A417C	CALL kernel32.ExitProcess	
00406020	8B25 00604000	MOV ESP,DWORD PTR DS:[406000]	
00406026	8B2D 04604000	MOV EBP,DWORD PTR DS:[406004]	
0040602C	803D 2C624000 35	CMP BYTE PTR DS:[40622C],35	
00406033	-0F84 11B1FFFF	JE defi_mar.0040114A	
00406039	-E9 21B1FFFF	JMP defi_mar.0040115F	

Ahah, il rusé mars; il a mis ce code en dehors de la section .text, pour qu' Ollydbg ne référence pas la SDR "valid" :).

Revenons aux instructions :

00401208	. 33C9	XOR ECX,ECX	; efface ecx
0040120A	. 33D2	XOR EDX,EDX	; efface edx
0040120C	. 66:8B0D 2F624000	MOV CX,WORD PTR DS:[40622F]	
00401213	. 66:8B15 4A624000	MOV DX,WORD PTR DS:[40624A]	
0040121A	. 66:03CA	ADD CX,DX	; ajoute dx à cx
0040121D	. 81C1 25250000	ADD ECX,2525	; ajoute 2525 à ecx
00401223	. B8 00000010	MOV EAX,10000000	; affecte 10000000 à eax
00401228	. F6F1	DIV CL	; effectue une division par cl

Par l'instruction en 0040120C, 6261 est affecté à cx.
Par l'instruction en 00401213, 7372 est affecté à dx.
On ajoute ensuite dx à cx, puis 2525 à ecx. On réalise ensuite une division de eax = 10000000 par cl.
Pour qu'il y ait exception, il faut une division par cl=0. Pour que cl=0, il faut remplir cx+dx+2525=xx00, c'est-à-dire (valeur ascii du 3ème caractère du nom)+(valeur ascii du 6ème caractère du serial)+25=00.
Pour mars, il faut résoudre $z + 0x69 + 0x25 = 00 \Leftrightarrow z + 0x8E$.
En prenant le "complément à deux", on obtient $z = 0x72$ (valeur correspondant à un "i").
Pour valider l'épreuve avec "mars", le 6ème caractère du serial doit être "i".
6ème condition : Les 3ème caractère du nom et 6ème caractère doivent avoir des valeurs dont la somme est nulle.

Voici un couple valide : mars/59215i012345678901234567890123€



En résumé :

- Le serial doit faire 30 caractères.
- Les 5 premiers caractères du serial doivent être numériques (0 exclus).
- Le 1er caractère du serial doit être plus grand ou égal au 3ème.
- Le 2ème caractère du serial doit être 9.
- Le 4ème caractère du serial doit être 1.
- Le 5ème caractère du serial doit être 5.
- Le dernier caractère du serial doit avoir une valeur ascii, supérieure ou égale à 0x80.
- Le 1er caractère du nom doit avoir une valeur ascii, inférieure ou égale à 0x7A ("z").
- Les 3ème caractère du nom et 6ème caractère du serial doivent avoir des valeurs dont la somme est nulle.
- Tous les autres caractères du nom et du serial peuvent être aléatoires...
- Pour activer le bouton "Register", il faut soit taper le serial à la main, soit faire un c/c du 1er bloc (les 5 premiers caractères) séparément du 2ème bloc.

Merci à mars pour ce défi très sympathique :).

uLysse_31.