

U4N1

Unpacking for Newbies - Kharneth

solution par mastermatt29

1.Introduction.....	2
2.Unpacking.....	3
a) Analyse du loader.....	4
b) Dump.....	8
c) Correction de l'IAT.....	8
d) Rétablissement de l'exé d'origine (ou presque).....	12
3.Trouver le serial.....	17
4.Conclusion.....	20

1.Introduction

Je vous propose ici une technique de résolution de l'Unpacking for Newbies #1 de Kharneth (que je remercie d'ailleurs au passage pour l'avoir codé). Le schéma de résolution pour ce crackme sera classique, à savoir unpacking, reconstruction des imports puis ayant un dump valide, recherche du serial.

Je tiens à signaler qu'il se peut que certaines subtilités m'aient échappé, que ma méthode ne soit pas la meilleure ou même que certaines choses ne soient pas tout à fait exactes. Si tel était le cas n'hésitez pas à me prévenir, j'accueillerai sans problèmes vos remarques (mastermatt29@gmail.com).

Dans le read-me accompagnant le crackme, Kharneth prévoit que l'analyse doit se dérouler suivant 4 étapes :

- dump fonctionnel du programme, tous tools autorisés (facultatif)
- analyse du loader à l'exception des fonctions de décompressions (à savoir l'ApLib dont le fonctionnement est détaillé par BeatriX dans son analyse du packer FSG)
- dump fonctionnel avec uniquement un débogueur, un dumper et un éditeur héra (reconstruction manuelle de l'IAT)
- supprimer le loader et récupérer l'exé le plus proche possible de l'original

Aussi autant que possible j'essaierai de m'y conformer.

Les outils que j'utiliserai :

- OllyDbg + plugin OllyDump (debugger)
- Hexplorer (éditeur hexadécimal)
- LordPE (éditeur PE)

Sauf indications contraires, tout au long de ce tutorial les nombres seront exprimés en base hexadécimale.

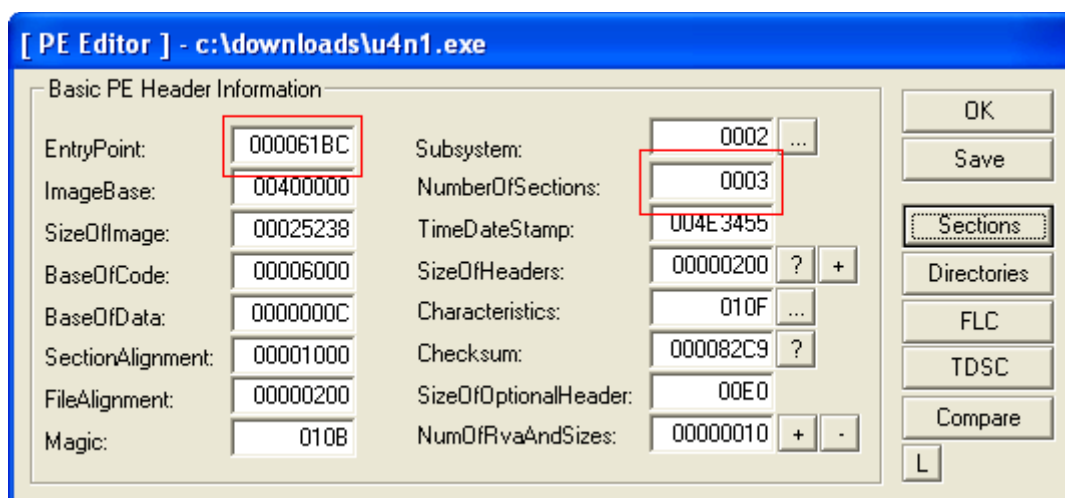
Et enfin LA doc nécessaire : pecoff.doc qui contient les spécifications de Microsoft sur le format Portable Executable (PE).

Allez c'est parti !

2.Unpacking

a) Analyse du loader

La première chose que nous allons faire est d'aller glaner quelques infos sur l'exé grâce à son PE. Le PE d'un programme est un en-tête présent dans chaque programme donnant des informations à Windows, genre par exemple là où débute le programme (Entry Point), l'emplacement des imports... On lance LordPE et on charge notre exé encore packé dans l'éditeur de PE. Bon regardons ce que nous avons :



Les informations intéressantes à retenir sont qu'il y a 3 sections et que l'Entry Point est en 61BC. Notons aussi que comme dans la majorité des cas l'ImageBase est à 400000, c'est à dire que toutes les adresses seront exprimées par rapport à cette adresse (par exemple l'Entry Point se trouvera en 4061BC). Bon maintenant allons voir les sections :

[Section Table]					
Name	VOffset	VSize	ROffset	RSize	Flags
.U4N0	00001000	00005000	00000000	00000000	E00000E0
.U4N1	00006000	00001C49	00000200	00001E00	E00000E0
.rsrc	00008000	0001D238	00002000	0001D238	40000040

Alors là on tombe sur quelque chose de vraiment intéressant : la section .U4N0 dont la taille sur le disque dur est nulle (RawSize), mais qui prend 5000 (VirtualSize) octets en mémoire ! L'EntryPoint

se trouvant dans la section .U4N1 on peut alors supposer que le code du loader se trouve dans cette section, et qu'il décompresse le code original dans la section .U4N0 où ce code sera ensuite exécuté. La section .rsrc représente les ressources du programme (curseur, icône...). On ne s'y intéressera qu'après avoir dépacké le programme.

Bon il est temps de lancer notre programme dans OllyDbg. On se retrouve comme prévu en 4061BC :

004061BC	55	PUSH EBP	<- Entry Point
004061BD	8BEC	MOV EBP,ESP	
004061BF	83EC 10	SUB ESP,10	
004061C2	60	PUSHAD	<- Sauvegarde de tous les registres
004061C3	6A 00	PUSH 0	pModule = NULL
004061C5	FF15 00604000	CALL DWORD PTR DS:[<&Kernel32.GetModuleHandleA]	GetModuleHandleA
004061C8	8945 FC	MOV DWORD PTR SS:[EBP-4],EAX	

La encore on trouve quelque chose de très intéressant : en 4061C2 on a un PUSHAD. Cette instruction sert à sauvegarder tous les registres sur la pile. Il faut savoir que l'on rencontre souvent cette instruction avant un loader (comme dans le packer UPX). Il est ensuite très facile de repérer le POPAD suivant correspondant (signifiant généralement la fin du loader) et d'unpacker la cible.

Bon là n'est pas la question, continuons. En 4061C5 un CALL à l'API GetModuleHandleA qui nous retourne ici l'ImageBase que l'on va stocker sur la pile grâce à MOV DWORD PTR SS:[EBP-4],EAX. Bref pas de quoi fouetter un chat.

004061CE	33C9	XOR ECX,ECX	<- Mise a zero de ECX
004061D0	EB 18	JMP SHORT U4N1.004061EA	
004061D2	8345 FC	ADD EAX,DWORD PTR SS:[EBP-4]	<- Adresse ou placer les donnees decomprimees
004061D5	50	PUSH EAX	
004061D6	8B04CD 5E6240	MOV EAX,DWORD PTR DS:[ECX*8+40625E]	<- Adresse des donnees a decompresser
004061D9	8345 FC	ADD EAX,DWORD PTR SS:[EBP-4]	<- Call de decompression
004061E0	50	PUSH EAX	
004061E1	E8 92FEFFFF	CALL U4N1.00406078	
004061E6	83C4 08	ADD ESP,8	
004061E9	41	INC ECX	
004061EA	8B04CD 5A6240	MOV EAX,DWORD PTR DS:[ECX*8+40625A]	<- La boucle commence pour la premiere fois ici
004061F1	85C0	TEST EAX,EAX	
004061F3	75 DD	JNZ SHORT U4N1.004061D2	

Les choses intéressantes viennent en 4061D2 où on trouve une boucle. On commence par placer une valeur dans EAX, puis on lui ajoute l'ImageBase (stockée en EBP-4, souvenez-vous), ce qui laisse donc penser que c'est une adresse. Ensuite on PUSH cette adresse puis on recommence l'opération avec une autre valeur dans EAX avant de lancer le CALL. Lorsque l'on débuge, on s'aperçoit que les adresses la première fois sont 401000 (tiens tiens le début de la section dont la taille est nulle :)) et 406282 (à priori des données, OllyDbg n'affiche que des CHAR). Tiens c'est sûrement la routine de décompression (que nous n'étudierons pas je le rappelle) ! Pour en être persuadé, placez vous sur le CALL, juste avant de l'exécuter.

Ensuite ouvrez la Memory Map (le bouton M).

Memory map									
Address	Size	Owner	Section	Contains	Type	Access	Initial	Mapped as	
00010000	00001000				Priv	Rw	Rw		
00020000	00001000				Priv	Rw	Rw		
00120000	00001000				Priv	Rw	Gua	Rw	
0012E000	00002000				Priv	Rw	Gua	Rw	
00130000	00003000				Map	R	R		
00140000	00003000				Priv	Rw	Rw		
00240000	00006000				Priv	Rw	Rw		
00250000	00003000				Map	Rw	Rw		
00260000	00016000				Map	R	R	\Device\HarddiskVolume1\WINDOWS\system32	
00280000	00030000				Map	R	R	\Device\HarddiskVolume1\WINDOWS\system32	
002C0000	00041000				Map	R	R	\Device\HarddiskVolume1\WINDOWS\system32	
00310000	00006000				Map	R	R	\Device\HarddiskVolume1\WINDOWS\system32	
00400000	00001000	U4N1		PE header	Imag	R	RWE		
00401000	00005000	U4N1	.U4N0		Imag	R	RWE		
00406000	00002000	U4N1	.U4N1	code, import	Imag	R	RWE		
00408000	0001E000	U4N1	.rsrc	resources	Imag	R	RWE		
7C800000	00001000	kernel32		PE header	Imag	R	RWE		
7C801000	00002000	kernel32	.text	code, import	Imag	R	RWE		
7C883000	00005000	kernel32	.data	data	Imag	R	RWE		
7C888000	00076000	kernel32	.rsrc	resources	Imag	R	RWE		
7C8FE000	00006000	kernel32	.reloc	relocations	Imag	R	RWE		
7C910000	00001000	ntdll		PE header	Imag	R	RWE		
7C911000	0007B000	ntdll	.text	code, export	Imag	R	RWE		
7C980000	00005000	ntdll	.data	data	Imag	R	RWE		
7C991000	00033000	ntdll	.rsrc	resources	Imag	R	RWE		
7C9C4000	00003000	ntdll	.reloc	relocations	Imag	R	RWE		
7F6F0000	00007000				Map	R E	R E		
7FFB0000	00024000				Map	R	R		
7FFDE000	00001000			data block	Priv	Rw	Rw		
7FFDF000	00001000				Priv	Rw	Rw		
7FFE0000	00001000				Priv	R	R		

Ensuite click-droit sur la section .U4N0 et là Dump. Une nouvelle fenêtre montrant le contenu de la section s'affiche. Positionnez les fenêtres CPU et Dump de manière à les voir toutes les 2. Ensuite un petit coup de F8 pour passer le CALL et paf ! La section s'est remplie comme par magie ! De la même manière on refait la boucle pour les décompressions suivantes (les données seront placées en 404000 et 405000). Ainsi il semblerai bien que l'exe original ait eu 3 sections... Enfin nous verrons cela tout à l'heure !

```

004061F5 | . 8B1D 56624000 MOV EBX,DWORD PTR DS:[406256]
004061F8 | . 035D FC ADD EBX,DWORD PTR SS:[EBP-4]
004061FE | ✓ EB 44 JMP SHORT U4N1.00406244
00406200 | > 0345 FC ADD EAX,DWORD PTR SS:[EBP-4]
00406203 | . 50 PUSH EAX
00406204 | . FF15 04604000 CALL DWORD PTR DS:[<&Kernel32.LoadLibraryA]
0040620A | . 8945 F8 MOV DWORD PTR SS:[EBP-8],EAX
0040620D | . 8B03 MOV EAX,DWORD PTR DS:[EBX]
0040620F | . 85C0 TEST EAX,EAX
00406211 | ✓ 75 03 JNZ SHORT U4N1.00406216
00406213 | . 8B43 10 MOV EAX,DWORD PTR DS:[EBX+10]
00406216 | > 0345 FC ADD EAX,DWORD PTR SS:[EBP-4]
00406219 | . 8BF8 MOV EDI,EAX
0040621B | . 8B73 10 MOV ESI,DWORD PTR DS:[EBX+10]
0040621E | . 0375 FC ADD ESI,DWORD PTR SS:[EBP-4]
00406221 | ✓ EB 18 JMP SHORT U4N1.0040623B
00406223 | > 0345 FC ADD EAX,DWORD PTR SS:[EBP-4]
00406226 | . 83C0 02 ADD EAX,2
00406229 | . 50 PUSH EAX
0040622A | . FF75 F8 PUSH DWORD PTR SS:[EBP-8]
0040622D | . FF15 08604000 CALL DWORD PTR DS:[<&Kernel32.GetProcAddress]
00406233 | . 8906 MOV DWORD PTR DS:[ESI],EAX
00406235 | . 83C6 04 ADD ESI,4
00406238 | . 83C7 04 ADD EDI,4
0040623B | > 8B07 MOV EAX,DWORD PTR DS:[EDI]
0040623D | . 85C0 TEST EAX,EAX
0040623F | ✓ 75 E2 JNZ SHORT U4N1.00406223
00406241 | . 83C3 14 ADD EBX,14
00406244 | > 8B43 0C MOV EAX,DWORD PTR DS:[EBX+C]
00406247 | . 85C0 TEST EAX,EAX
00406249 | ✓ 75 B5 JNZ SHORT U4N1.00406200
0040624B | . 61 POPAD
0040624C | . C9 LEAVE
0040624D | . 68 00104000 PUSH U4N1.00401000
00406252 | . C3 RETN

```

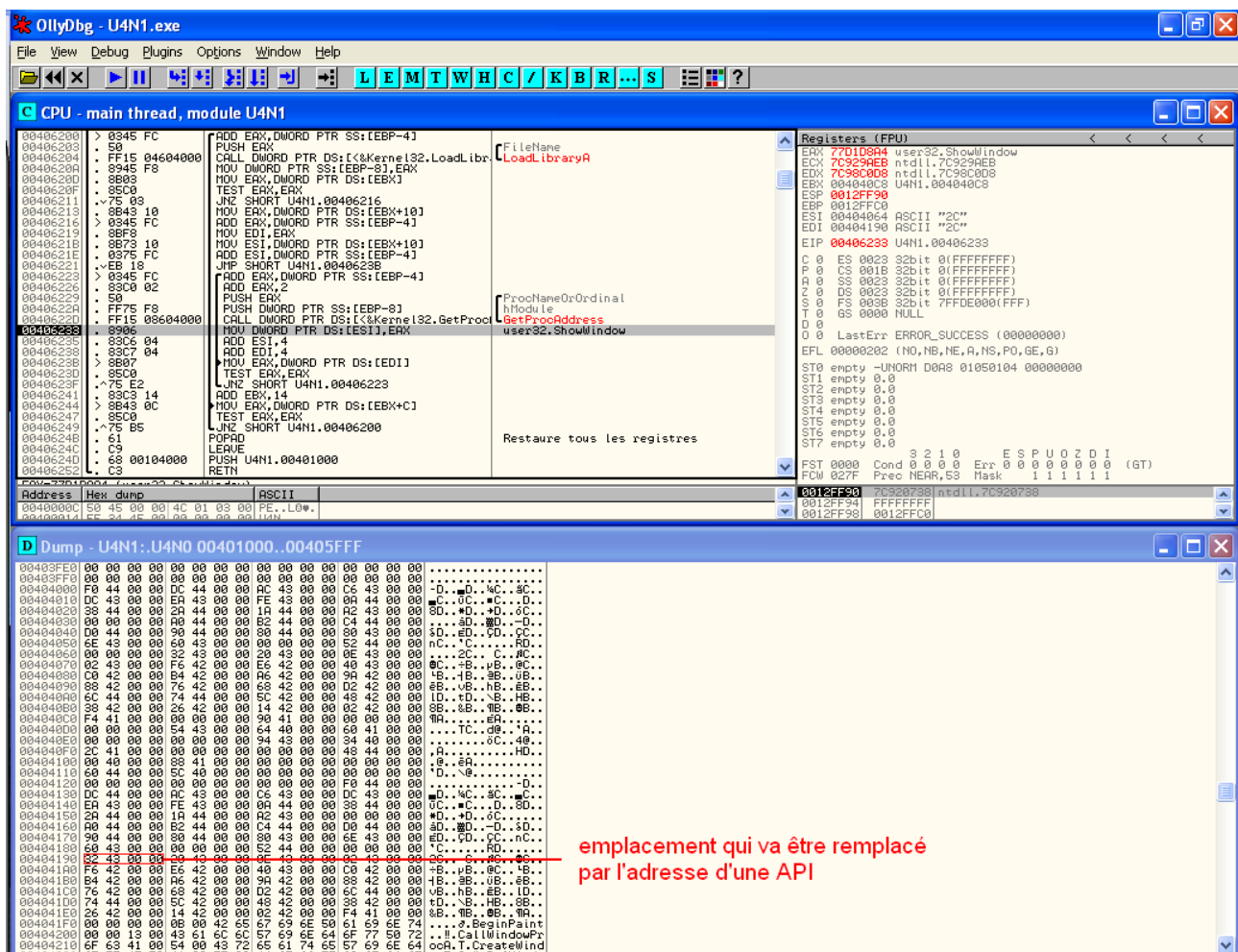
[FileName
LoadLibraryA
 ProcNameOrOrdinal
hModule
GetProcAddress
 Restaure tous les registres

On arrive maintenant à une double boucle intéressante, car faisant intervenir les API LoadLibraryA (servant à charger les DLL) et GetProcAddress (servant à chercher les adresses des API).

Note rapide sur les imports : il faut savoir qu'au lancement d'un programme, Windows recherche dans l'executable l'ImportDirectoryTable qui contient les noms des DLL à charger. Cette structure pointe entre autre vers 2 autres structures (ImportLookupTable et ImportAddressTable) contenant les noms des API à charger. Ces 2 tableaux sont initialement identiques, mais ImportAddressTable va être changé par Windows pour contenir non plus les noms des API, mais leurs adresses (nous verrons où exactement plus loin). Ici le packer émule cette fonction normalement exécutée par l'OS. Ce schéma est très souvent utilisé par les packers.

Bon analysons un peu. Le programme charge tout d'abord une DLL en mémoire grâce à LoadLibraryA. La boucle suivante récupère les adresses des API et les place à l'emplacement pointé par EDI. Débuggons ça un petit peu pour voir où EDI pointe...

On passe l'appel à LoadLibraryA (chargement de user32.dll) et on va se placer en 406233, juste après l'appel de GetProcAddressA. EDI pointe alors en 404190. Comme tout à l'heure on va voir du côté de l'adresse 404190 dans la fenêtre de Dump.



Ce qu'on remarque c'est que juste avant 404000 il y a tout un tas de 00 et que aux environs de 404200 il y a tout un tas de noms d'API.....Mais ça ressemblerait pas à une IAT ca par hasard ? Non de non mais c'est bien sur ! Le packer est justement en train de placer les adresses des API dans l'ImportAddressTable ! Donc la section .idata commence certainement en 4000 ! Gardons cette information sous le coude pour tout à l'heure...
La fin du code du loader ne représente pas grand intérêt si ce n'est les 2 dernières instructions :

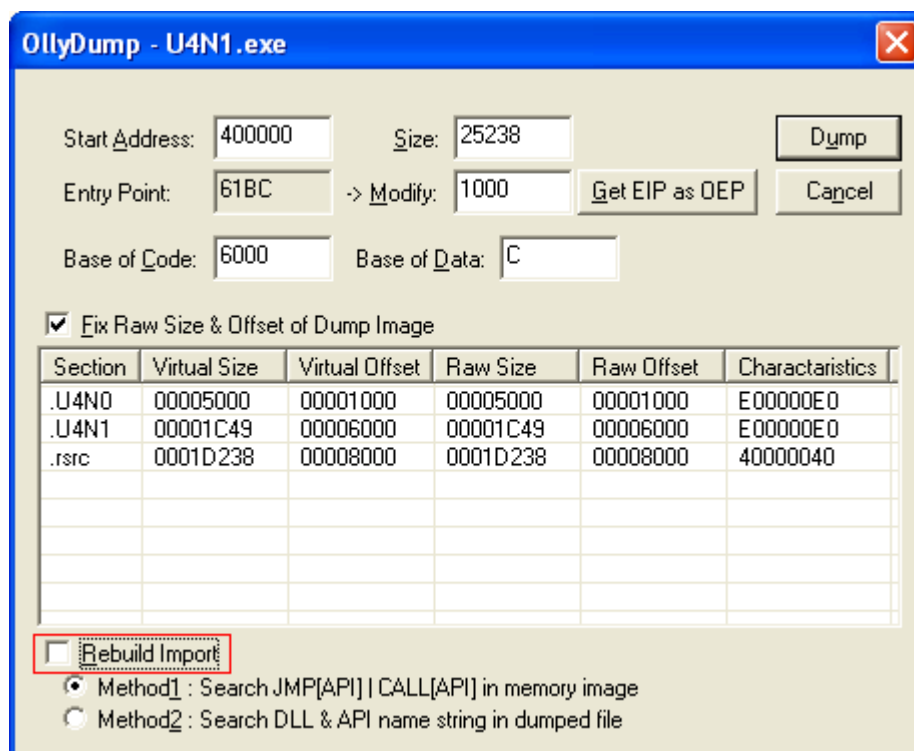
```
0040624D . 68 00104000  PUSH U4N1.00401000
00406252 . C3                RETN
```

Ainsi donc on va aller en 401000, pile poil au début du code que l'on a décompresser ! 401000 est donc l'Original Entry Point (OEP), c'est à dire l'Entry Point du programme avant qu'il ne soit packé !

b) Dump

J'utiliserai pour ma part le plugin OllyDump, mais tout autre dumper pourra faire l'affaire.

Premièrement dans Olly, on trace jusqu'à se placer en 401000 (on passe le dernier RET du loader quoi).



Donc on va dans Plugins->OllyDump->Dump debugged process. Là vous voyez cette boîte de dialogue. N'oubliez pas de décocher la case Rebuild Import (on va faire ça manuellement). Vérifiez que l'EntryPoint est bien modifié en 1000. Et pouf on dump !

c) Correction de l'IAT

A cette étape nous n'avons pas encore un exe valide. En effet le packer en se greffant sur l'exé fait charger par Windows sa table d'import et ensuite rend valide la notre par des appels de LoadLibraryA/GetProcAddress. Il faut donc que nous restaurions notre IAT pour que ce soit elle qui soit chargée par Windows au démarrage du programme, à savoir :

- restaurer notre ImportAddressTable (qui a été modifiée par le loader)

- modifier le PE header pour que notre IAT soit reconnue

Désormais on laisse tomber Olly pour ne plus prendre que notre éditeur hexadécimal (Hexplorer pour ma part).

Note : ne pas oublier que les données sont stockées en sens inverse. Par exemple 12 34 56 78 sera stocké en 78 56 43 12.

Note plus longue sur les structures des imports :

- L'ImportDirectoryTable est disons le point d'entrée. Elle se compose d'entrées contenant chacune 5 DWORD (un pointeur vers une ImportLookupTable, 2 DWORD dont on ne se soucie pas, un pointeur vers le nom d'une DLL et un pointeur vers une ImportAddressTable (même structure que l'ImportLookupTable)) plus une entrée de 5 DWORD NULL en signe de terminaison. Ce sont ces entrées qui détermineront quelles DLL seront chargées.
- L'ImportLookupTable est composée de DWORDs qui pour notre cas sont des pointeurs vers des structures NameTable plus un DWORD NULL de terminaison
- La structure NameTable est composée d'entrées contenant chacune un WORD dont on ne se soucie pas et d'une chaîne de caractère contenant le nom de l'API de la DLL associée à charger

(Euh je voulais faire un schéma pour mieux expliquer que ces phrases indigestes mais je suis vraiment trop une carpe en graphik donc j'ai laissé tomber. Désolé !)

Notons que l'ImportAddressTable est la même au lancement du programme que l'ImportLookupTable. Seulement Windows changera ses entrées pour y mettre les adresses en mémoire des API.

Voilà j'espère que vous avez compris... Je sais que c'est pas évident mais si vous avez des problèmes, n'hésitez pas à vous plonger dans la doc Microsoft, tout y est expliqué (sans doute mieux que je ne le fais).

Donc après l'ImportAddressTable (composée des adresses en 77xxxxxx) et l'ImportDirectoryTable (terminée par 5 DWORD NULL) on trouve enfin ce qu'on cherche : l'ImportLookupTable. Cette structure est identique à ce qu'était l'ImportAddressTable avant de recevoir les adresses des API. On va donc recopier les valeurs s'étendant de 412C à 41EF en 4000. Ce qui donne :

(Avant)

00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00	5@iwI'iwCpiw`iw	Au point rouge on est en 4000
35 A9 EF 77	49 92 EF 77	43 70 EF 77	00 60 EF 77	`^ñwÅaiw~niw-liw	
60 B2 F1 77	C5 61 EF 77	98 6E EF 77	2D 6C EF 77	-]iwI^iw□[iw^oiw	
97 5D EF 77	49 5E EF 77	90 5B EF 77	B2 6F EF 77	ñ°€ I^€ Ô□'	
00 00 00 00	F1 BA 80 7C	49 AA 80 7C	D4 05 92 7C	=□' e € ±Ç€)µ€	ImportLookupTable
3D 04 92 7C	65 A0 80 7C	B1 C7 80 7C	29 B5 80 7C	□,□ ¢Ê□ œ□lv	
8D 2C 81 7C	A2 CA 81 7C	00 00 00 00	9C 11 31 76	*ØÑwCδÒw□ÖÑw	
00 00 00 00	A4 D8 D1 77	43 F5 D2 77	0D D6 D1 77	.œÑw`ÚÑwšóÒwδ<Ñw	
2E 8C D1 77	60 DA D1 77	9A F3 D2 77	F6 8B D1 77	□□Òw\$□Òw>□ÒwBœÑw	ImportDirectoryTable
11 12 D2 77	24 13 D2 77	3E 0B D2 77	42 8C D1 77	δµÑw<!ÓwB□Òwæ7Òw	
F5 B5 D1 77	3C 21 D3 77	42 10 D2 77	E6 37 D2 77	Ç+Ñw□+Ñw□¶Ñw,-Ñw	
C7 86 D1 77	9D 86 D1 77	1D B6 D1 77	B8 96 D1 77	êÚÑwîÖÑw^□ÒwêèÑw	
EADA D1 77	EE D4 D1 77	5E 02 D2 77	EAE8 D1 77	□¶Ñw□A	ImportAddressTable
09 B6 D1 77	00 00 00 00	90 41 00 00	00 00 00 00	TCd@`A	
00 00 00 00	54 43 00 00	64 40 00 00	60 41 00 00	"C4@	
00 00 00 00	00 00 00 00	94 43 00 00	34 40 00 00	,AHD	
2C 41 00 00	00 00 00 00	00 00 00 00	48 44 00 00	@^A	
00 40 00 00	88 41 00 00	00 00 00 00	00 00 00 00	`D\@	
60 44 00 00	5C 40 00 00	00 00 00 00	00 00 00 00	δD	
00 00 00 00	00 00 00 00	00 00 00 00	F0 44 00 00	ÜD¬CÆCÜC	
DC 44 00 00	AC 43 00 00	C6 43 00 00	DC 43 00 00	êCþC□D8D	ImportAddressTable
EA 43 00 00	FE 43 00 00	0A 44 00 00	38 44 00 00	*D□D¢C	
2A 44 00 00	1A 44 00 00	A2 43 00 00	00 00 00 00	D^DÄDÐD	
A0 44 00 00	B2 44 00 00	C4 44 00 00	D0 44 00 00	□D€D€CnC	
90 44 00 00	80 44 00 00	80 43 00 00	6E 43 00 00	`C RD	
60 43 00 00	00 00 00 00	52 44 00 00	00 00 00 00	2C C□C□C	
32 43 00 00	20 43 00 00	0E 43 00 00	02 43 00 00	δDæB¢BÀB	
F6 42 00 00	E6 42 00 00	40 43 00 00	C0 42 00 00	'B BšB^B	
B4 42 00 00	A6 42 00 00	9A 42 00 00	88 42 00 00	vBhBòBlD	
76 42 00 00	68 42 00 00	D2 42 00 00	6C 44 00 00	tD\BHB8B	
74 44 00 00	5C 42 00 00	48 42 00 00	38 42 00 00	&B□B□BδA	
26 42 00 00	14 42 00 00	02 42 00 00	F4 41 00 00	□BeginPaint	
00 00 00 00	0B 00 42 65	67 69 6E 50	61 69 6E 74		

(Après)

00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00					
F0 44 00 00	DC 44 00 00	AC 43 00 00	C6 43 00 00		öD	ÜD	¬C	ÆC
DC 43 00 00	EA 43 00 00	FE 43 00 00	0A 44 00 00		ÛC	êC	þC	□D
38 44 00 00	2A 44 00 00	1A 44 00 00	A2 43 00 00		8D	*D	□D	¢C
00 00 00 00	A0 44 00 00	B2 44 00 00	C4 44 00 00			D	²D	ÄD
D0 44 00 00	90 44 00 00	80 44 00 00	80 43 00 00		ÐD	□D	€D	€C
6E 43 00 00	60 43 00 00	00 00 00 00	52 44 00 00		nC	`C		RD
00 00 00 00	32 43 00 00	20 43 00 00	0E 43 00 00			2C	C	□C
02 43 00 00	F6 42 00 00	E6 42 00 00	40 43 00 00		□C	öB	æB	@C
C0 42 00 00	B4 42 00 00	A6 42 00 00	9A 42 00 00		ÀB	´B	¡B	ŠB
88 42 00 00	76 42 00 00	68 42 00 00	D2 42 00 00		^B	vB	hB	ÒB
6C 44 00 00	74 44 00 00	5C 42 00 00	48 42 00 00		lD	tD	\B	HB
38 42 00 00	26 42 00 00	14 42 00 00	02 42 00 00		8B	&B	□B	□B
F4 41 00 00	00 00 00 00	90 41 00 00	00 00 00 00		ôA		□A	
00 00 00 00	54 43 00 00	64 40 00 00	60 41 00 00			TC	d@	`A
00 00 00 00	00 00 00 00	94 43 00 00	34 40 00 00				"C	4@
2C 41 00 00	00 00 00 00	00 00 00 00	48 44 00 00		,A			HD
00 40 00 00	88 41 00 00	00 00 00 00	00 00 00 00		@	^A		
60 44 00 00	5C 40 00 00	00 00 00 00	00 00 00 00		`D	\@		
00 00 00 00	00 00 00 00	00 00 00 00	F0 44 00 00					öD
DC 44 00 00	AC 43 00 00	C6 43 00 00	DC 43 00 00		ÜD	¬C	ÆC	ÛC
EA 43 00 00	FE 43 00 00	0A 44 00 00	38 44 00 00		êC	þC	□D	8D
2A 44 00 00	1A 44 00 00	A2 43 00 00	00 00 00 00		*D	□D	¢C	
A0 44 00 00	B2 44 00 00	C4 44 00 00	D0 44 00 00		D	²D	ÄD	ÐD
90 44 00 00	80 44 00 00	80 43 00 00	6E 43 00 00		□D	€D	€C	nC
60 43 00 00	00 00 00 00	52 44 00 00	00 00 00 00		`C		RD	
32 43 00 00	20 43 00 00	0E 43 00 00	02 43 00 00		2C	C	□C	□C
F6 42 00 00	E6 42 00 00	40 43 00 00	C0 42 00 00		öB	æB	@C	ÀB
B4 42 00 00	A6 42 00 00	9A 42 00 00	88 42 00 00		´B	¡B	ŠB	^B
76 42 00 00	68 42 00 00	D2 42 00 00	6C 44 00 00		vB	hB	ÒB	lD
74 44 00 00	5C 42 00 00	48 42 00 00	38 42 00 00		tD	\B	HB	8B
26 42 00 00	14 42 00 00	02 42 00 00	F4 41 00 00		&B	□B	□B	ôA
00 00 00 00	0B 00 42 65	67 69 6E 50	61 69 6E 74					□ BeginPaint

Parfait, notre première étape est accomplie ! Il ne nous reste donc plus qu'à indiquer là où est notre ImportDirectoryTable (40C8) et sa taille (50, on ne compte pas les 5 DWORD NULL de terminaison).

Si vous avez suivi la doc sur le PE que j'ai indiqué au début vous saurez qu'il faut mettre ces informations sous forme de 2 DWORD à l'offset 8C (offset d'abord puis taille). Et surtout n'oubliez pas d'inverser l'ordre des données ! (il va donc falloir rentrer les octets C8 40 00 00 50 00 00 00). C'est bon ? Ya plus qu'à sauvegarder et tester !

Formidable ça marche !

Note : si jamais nous avons dumpé le programme entre la décompression des sections et le chargement des API, il n'aurait pas été nécessaire de modifier l'ImportAddressTable, vu qu'elle était encore dans son état original !

d)Rétablissement de l'exe d'origine (ou presque)

Désormais, le loader n'étant plus exécuté, il devient donc inutile dans notre programme. Bon ben ya qu'à le supprimer alors ! Et pis tant qu'on y est autant rétablir les sections comme elles étaient à l'origine ! Je vous rappelle que lors de la décompression nous avons remarqué 3 zones : en 401000, en 404000 et en 405000. On sait déjà que première section va être la section `.code` vu qu'elle contient l'EntryPoint. La deuxième contient les imports c'est donc la section `.idata`. Regardons la dernière. Elle contient des phrases genre `MainWinClass`, c'est donc la section `.data`.

Parfois nous avons nos 3 sections du programme original ! Vous êtes sûrs qu'on oublie rien ??

Eh oui la section `resource` ! Celle-ci n'a pas été compressée et elle contient entre autre l'icone du programme et le background. Donc il faudra elle aussi la garder.

Le loader se situe lui en 6000 et prend 2000 octets (1C49 en fait plus le padding pour l'alignement des sections). Donc on sélectionne le code entre les offsets 6000 et 8000 (clic sur le premier octet, puis maintenez shift enfoncé en cliquant sur le dernier octet (celui en 7FFF, pas en 8000). Ensuite clic-droit sur la zone sélectionnée et Delete.

Résumons l'état des sections que nous avons à recréer/modifier :

- section `.code` à l'offset 1000 de taille 3000
- section `.idata` à l'offset 4000 de taille 1000
- section `.data` à l'offset 5000 de taille 1000
- section `.rsrc` à l'offset 6000, la taille importe peu vu qu'on gardera la même

Maintenant direction l'entête des sections ! Celle-ci se situe peu après le PE, aux environs de l'offset 100. L'entête est aisément identifiable car il contient le nom des sections. Bon lisons un peu notre doc. Pour chaque section il nous faut :

- le nom de la section sur 8 caractères (complété par des 00 si nécessaire)
- VirtualSize sur 4 octets
- VirtualAddress sur 4 octets
- RawSize sur 4 octets
- RawAddress sur 4 octets
- 3 DWORD dont on ne se soucie pas
- les caractéristiques sur 4 octets

Listons donc ces attributs pour nos 4 sections :

- .code
- 3000
- 1000
- 3000
- 1000
- 00000000 00000000 00000000
- E0 00 00 E0

- .idata
- 1000
- 4000
- 1000
- 4000
- 00000000 00000000 00000000
- E0 00 00 E0

- .data
- 1000
- 5000
- 1000
- 5000
- 00000000 00000000 00000000
- E0 00 00 E0

- .rsrc
- 6000
- 1D238 (c'est marqué dans l'entête original)
- 6000
- 1D238
- 00000000 00000000 00000000
- E0 00 00 E0

Bon ben ya plus qu'à rentrer ça à partir de l'offset 104 (là où débute le nom de la section .U4N0).

Seulement c'est pas tout ! Il va falloir maintenant signaler que l'on a changé le nombre de sections et que la section ressource à changé de place. D'après la doc Microsoft le champ NumberOfSection est le deuxième WORD après la signature PE, c'est ici à dire à l'offset 12. On change donc 03 en 04.

Ensuite, toujours d'après la doc la section ressource est signalée dans le PE header dans les DataDirectory en 3e position. On va donc changer le champ VirtualAddress à l'offset 94 de 8000 vers 6000.

Il ne nous reste plus qu'à changer les adresses de la section ressources qui pointent vers les ressources elle-mêmes. La doc nous dit que la section ressource est organisée de telle manière qu'il y a 3 niveaux de structure, les précédentes pointant sur les suivantes, la dernière correspondant aux données. On va appliquer ça tout de suite. Direction l'offset 6000 !

Encore une fois on sort notre doc. En premier lieu on tombe sur une structure ResourceDirectoryTable de 10 octets. On va plutôt s'intéresser à la structure qui la suit : ResourceDirectoryEntries. Celle-ci se compose d'un DWORD contenant un identifiant puis d'un autre DWORD pointant soit sur une nouvelle structure ResourceDirectoryTable soit sur un pointeur pointant sur les données ! Si jamais il s'agit d'un pointeur vers une nouvelle structure ResourceDirectoryTable l'adresse commencera par 80. Dans les 2 cas l'adresse est une RVA par rapport au début de la section.

Voici le chemin à suivre entre les différents pointeurs :

00 00 00 00	00 00 00 00	00 00 00 00	00 00 03 00		
03 00 00 00	28 00 00 80	0A 00 00 00	50 00 00 80		(€ P €
0E 00 00 00	A8 00 00 80	00 00 00 00	00 00 00 00		" €
00 00 00 00	00 00 03 00	01 00 00 00	C0 00 00 80		À €
02 00 00 00	D8 00 00 80	03 00 00 00	F0 00 00 80		ø € ð €
00 00 00 00	00 00 00 00	00 00 00 00	00 00 09 00		
9A 02 00 00	08 01 00 80	9B 02 00 00	20 01 00 80	š	€ > €
9C 02 00 00	38 01 00 80	9D 02 00 00	50 01 00 80	œ	8 € P €
FF 02 00 00	68 01 00 80	00 03 00 00	80 01 00 80	ÿ	h € € €
01 03 00 00	98 01 00 80	20 03 00 00	B0 01 00 80		~ € ° €
21 03 00 00	C8 01 00 80	00 00 00 00	00 00 00 00	!	È €
00 00 00 00	00 00 01 00	6F 00 00 00	E0 01 00 80		o à €
00 00 00 00	00 00 00 00	00 00 00 00	00 00 01 00		
09 04 00 00	F8 01 00 00	00 00 00 00	00 00 00 00	ø	
00 00 00 00	00 00 01 00	09 04 00 00	08 02 00 00		
00 00 00 00	00 00 00 00	00 00 00 00	00 00 01 00		
09 04 00 00	18 02 00 00	00 00 00 00	00 00 00 00		
00 00 00 00	00 00 01 00	09 04 00 00	28 02 00 00		(
00 00 00 00	00 00 00 00	00 00 00 00	00 00 01 00		
09 04 00 00	38 02 00 00	00 00 00 00	00 00 00 00	8	
00 00 00 00	00 00 01 00	09 04 00 00	48 02 00 00		H
00 00 00 00	00 00 00 00	00 00 00 00	00 00 01 00		
09 04 00 00	58 02 00 00	00 00 00 00	00 00 00 00	x	
00 00 00 00	00 00 01 00	09 04 00 00	68 02 00 00		h
00 00 00 00	00 00 00 00	00 00 00 00	00 00 01 00		
09 04 00 00	78 02 00 00	00 00 00 00	00 00 00 00	x	
00 00 00 00	00 00 01 00	09 04 00 00	88 02 00 00		^
00 00 00 00	00 00 00 00	00 00 00 00	00 00 01 00		
09 04 00 00	98 02 00 00	00 00 00 00	00 00 00 00	~	
00 00 00 00	00 00 01 00	09 04 00 00	A8 02 00 00		"
00 00 00 00	00 00 00 00	00 00 00 00	00 00 01 00		
09 04 00 00	B8 02 00 00	D0 82 00 00	68 03 00 00	,	Đ, h
00 00 00 00	00 00 00 00	38 86 00 00	A8 0C 00 00		8† "
00 00 00 00	00 00 00 00	E0 92 00 00	A8 1C 00 00		à' "

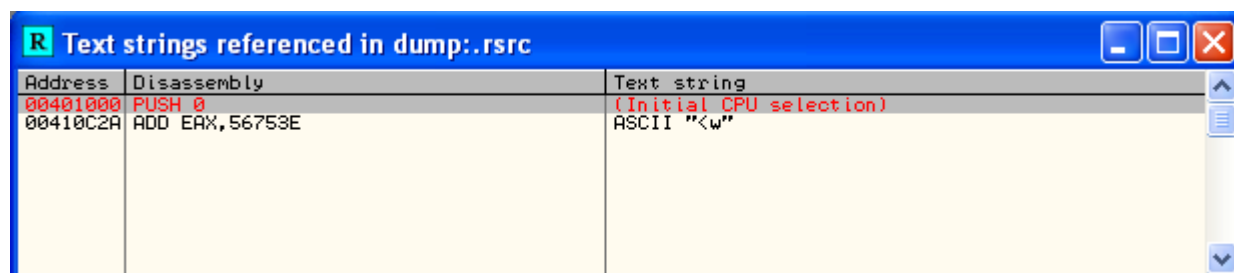
En rouge ce sont les pointeurs et en vert les structures pointées. Notez que pointeurs en rouge sont généralement associés au début de la structure en vert qui les précède. On arrive donc jusqu'à un pointeur vers 82D0 qui étaient des données. On peut à ce moment là supposer que toutes les structures pointant sur des données sont là, les unes à la suite des autres (un programme bien rangé quoi). Sachant que ces structures font 10 octets de long on va pouvoir modifier toutes les champs pointant vers des données en leur enlevant 2000 (la différence entre 8000 et 6000). On commence donc par 82D0 (dernier cadre vert) qu'on remplace par 62D0, puis 8638 par 6638, 92E0 par 72E0...jusqu'au dernier : AF88, remplacé par 8F88.

Ca y est ! Notre dump est maintenant totalement ressemblant au programme d'origine et pleinement fonctionnel ! Il ne reste maintenant plus qu'à trouver le sérial ;-)

3.Trouver le s rial

Ayant enfin fini d'unpacker le programme, on va pouvoir maintenant s'attaquer au s rial ! Rassurez-vous il n'y a rien de bien m chant !

Un petit clic-droit->Search for referenced text strings nous am ne sur cette fen tre :



Bref autant dire qu'il n'y a rien d'int ressant pour nous. On va donc poser un breakpoint sur la fonction charg e de r cup rer le serial : l'API GetWindowTextA. Il se trouve que en g n ral dans les programmes, il y a un endroit ou sont r f renc s tous sauts vers les API sous la forme de JUMP DWORD PTR [xxx]. Dans notre cas c'est en 401824 :

00401824	-FF25 C0404000	JMP DWORD PTR DS:[4040C0]	user32.BeginPaint
0040182A	-FF25 BC404000	JMP DWORD PTR DS:[4040BC]	user32.CallWindowProcA
00401830	-FF25 B8404000	JMP DWORD PTR DS:[4040B8]	user32.CreateWindowExA
00401836	-FF25 B4404000	JMP DWORD PTR DS:[4040B4]	user32.DefWindowProcA
0040183C	-FF25 B0404000	JMP DWORD PTR DS:[4040B0]	user32.DestroyWindow
00401842	-FF25 AC404000	JMP DWORD PTR DS:[4040AC]	user32.DispatchMessageA
00401848	-FF25 A8404000	JMP DWORD PTR DS:[4040A8]	user32.EndPaint
0040184E	-FF25 98404000	JMP DWORD PTR DS:[404098]	user32.GetMessageA
00401854	-FF25 94404000	JMP DWORD PTR DS:[404094]	user32.GetWindowTextA
0040185A	-FF25 90404000	JMP DWORD PTR DS:[404090]	user32.InvalidateRect
00401860	-FF25 8C404000	JMP DWORD PTR DS:[40408C]	user32.KillTimer
00401866	-FF25 88404000	JMP DWORD PTR DS:[404088]	user32.LoadCursorA
0040186C	-FF25 84404000	JMP DWORD PTR DS:[404084]	user32.LoadIconA
00401872	-FF25 80404000	JMP DWORD PTR DS:[404080]	user32.PostQuitMessage
00401878	-FF25 7C404000	JMP DWORD PTR DS:[40407C]	user32.RegisterClassExA
0040187E	-FF25 78404000	JMP DWORD PTR DS:[404078]	user32.SendMessageA
00401884	-FF25 74404000	JMP DWORD PTR DS:[404074]	user32.SetFocus
0040188A	-FF25 70404000	JMP DWORD PTR DS:[404070]	user32.SetTimer
00401890	-FF25 6C404000	JMP DWORD PTR DS:[40406C]	user32.SetWindowLongA
00401896	-FF25 68404000	JMP DWORD PTR DS:[404068]	user32.SetWindowTextA
0040189C	-FF25 64404000	JMP DWORD PTR DS:[404064]	user32.ShowWindow
004018A2	-FF25 7C404000	JMP DWORD PTR DS:[40407C]	user32.TranslateMessage
004018A8	-FF25 54404000	JMP DWORD PTR DS:[404054]	kernel32.ExitProcess
004018AE	-FF25 50404000	JMP DWORD PTR DS:[404050]	kernel32.GetCommandLineA
004018B4	-FF25 4C404000	JMP DWORD PTR DS:[40404C]	kernel32.GetModuleHandleA
004018BA	-FF25 2C404000	JMP DWORD PTR DS:[40402C]	GDI32.BitBlt
004018C0	-FF25 08404000	JMP DWORD PTR DS:[404008]	GDI32.CreateCompatibleBitmap
004018C6	-FF25 0C404000	JMP DWORD PTR DS:[40400C]	GDI32.CreateCompatibleDC
004018CC	-FF25 10404000	JMP DWORD PTR DS:[404010]	GDI32.CreateFontA
004018D2	-FF25 14404000	JMP DWORD PTR DS:[404014]	GDI32.CreateSolidBrush
004018D8	-FF25 18404000	JMP DWORD PTR DS:[404018]	GDI32.DeleteDC
004018DE	-FF25 1C404000	JMP DWORD PTR DS:[40401C]	GDI32.DeleteObject
004018E4	-FF25 28404000	JMP DWORD PTR DS:[404028]	GDI32.SelectObject
004018EA	-FF25 24404000	JMP DWORD PTR DS:[404024]	GDI32.SetBkColor
004018F0	-FF25 20404000	JMP DWORD PTR DS:[404020]	GDI32.SetTextColor
004018F6	-FF25 5C404000	JMP DWORD PTR DS:[40405C]	

On va donc poser un breakpoint (F2) sur user32.GetWindowTextA, puis lancer le programme avec F9. L  on rentre un serial bidon

(123456 pour ma part) et on clique sur OK. Le break s'effectue normalement, on clique sur le bouton Trace till return, puis encore un petit coup de F8 pour arriver à l'endroit qui nous intéresse :

00401320	6A 31	PUSH 31	
00401322	68 00504000	PUSH dump.004050D0	<- On push le buffer qui contiendra notre serial
00401327	FF35 8C504000	PUSH DWORD PTR DS:[40508C]	
0040132D	E3 22050000	CALL <JMP.&user32.GetWindowTextA>	
00401332	85C0	TEST EAX,EAX	<- On arrive la
00401334	74 36	JE SHORT dump.0040136C	<- Verifie qu'on a bien rentrer un serial
00401336	8BC8	MOV ECX,EAX	<- Met la longueur du serial dans ECX
00401338	8A81 CF504000	MOV AL,BYTE PTR DS:[ECX+4050CF]	<- Debut de la boucle
0040133E	50	PUSH EAX	
0040133F	E8 34000000	CALL dump.00401378	<- Call interessant
00401344	8881 CF504000	MOV BYTE PTR DS:[ECX+4050CF],AL	
0040134A	E0 EC	LOOPDNE SHORT dump.00401338	<- Fin de la boucle
0040134C	68 2F514000	PUSH dump.0040512F	ASCII "Pbatenghyngubaf! :)"
00401351	68 00504000	PUSH dump.004050D0	ASCII "123456"
00401356	E8 70000000	CALL dump.004013CB	<- Comparaison des 2 chaines
0040135B	85C0	TEST EAX,EAX	
0040135D	74 0D	JE SHORT dump.0040136C	

Je pense que le code est suffisamment commenté on va donc maintenant s'intéresser au CALL en 40133F :

00401378	55	PUSH EBP	
00401379	8BEC	MOV EBP,ESP	
0040137B	52	PUSH EDX	
0040137C	51	PUSH ECX	
0040137D	0FB645 08	MOVBX EAX,BYTE PTR SS:[EBP+8]	
00401381	83E8 41	SUB EAX,41	
00401384	78 39	JS SHORT dump.004013BF	<- Saute si caractere < 41
00401386	83E8 1A	SUB EAX,1A	
00401389	78 0C	JS SHORT dump.00401397	<- Saute si caractere < 41+1A
0040138B	83E8 06	SUB EAX,6	
0040138E	78 2F	JS SHORT dump.004013BF	<- Saute si caractere < 41+1A+6
00401390	83E8 1A	SUB EAX,1A	
00401393	78 16	JS SHORT dump.004013AB	<- Saute si caractere < 41+1A+6+1A
00401395	EB 28	JMP SHORT dump.004013BF	
00401397	0FB645 08	MOVBX EAX,BYTE PTR SS:[EBP+8]	<- 41 < Caractere < 41+1A
0040139B	83E8 34	SUB EAX,34	
0040139E	B9 1A000000	MOV ECX,1A	
004013A3	99	CDQ	
004013A4	F7F1	DIV ECX	
004013A6	83C2 41	ADD EDX,41	
004013A9	EB 18	JMP SHORT dump.004013C3	
004013AB	0FB645 08	MOVBX EAX,BYTE PTR SS:[EBP+8]	<- 41+1A+6 < Caractere < 41+1A+6+1A
004013AF	83E8 54	SUB EAX,54	
004013B2	B9 1A000000	MOV ECX,1A	
004013B7	99	CDQ	
004013B8	F7F1	DIV ECX	
004013BA	83C2 61	ADD EDX,61	
004013BD	EB 04	JMP SHORT dump.004013C3	
004013BF	0FB655 08	MOVBX EDX,BYTE PTR SS:[EBP+8]	<- 41+1A < Caractere < 41+1A+6 (ne fais rien)
004013C3	8BC2	MOV EAX,EDX	
004013C5	59	POP ECX	
004013C6	5A	POP EDX	
004013C7	C9	LEAVE	
004013C8	C2 0400	RETN 4	

Là encore le code est suffisamment commenté. Il suffit de connaître les quelques instructions assembleur utilisées pour savoir comment fonctionne l'algorithme transformant notre serial. L'algorithme n'étant pas simple à reverser, nous allons coder une sorte de bruteforce, ou plutôt un programme qui va effectuer les opérations en sens inverse à notre place.

Que savons-nous ?

- si le caractère d'entrée fait moins de 41, est entre 41+1A et 41+1A+6 ou est supérieur à 41+1A+6+1A, alors le caractère n'est pas modifié
- si le caractère est compris entre 41 et 41+1A alors on lui soustrait 34 et on garde le reste de sa division euclidienne par 1A auquel on rajoute 61
- si le caractère est compris entre 41+1A+6 et 41+1A+6+1A alors

on lui soustrait 54 et on garde le reste de sa division euclidienne par 1A auquel on rajoute 61

Parfait c'est tout ce qui nous faut pour coder notre programme à partir de la chaine «Pbatenghyngvbaf! :)»

J'ai mis la source dans le zip. Elle est normalement assez commentée pour que son fonctionnement soit aisément compréhensible.

On trouve finalement le serial «Congratulations! :)».

4.Conclusion

Ayé c'est fini ! J'espère que ce tuto aura plu à ceux qui l'auront lu. Toutes les remarques sont bien entendues les bienvenues étant donné que c'est mon premier tuto j'espère ne pas avoir fait trop court (ni trop long).

Enfin je tiens à remercier tous ceux qui m'ont aidé de par leurs réponses à mes questions ou par leurs écrits, mais plus particulièrement :

- BeatriX
- Deamon (dont les tutos m'ont été précieux au début)
- HolyView
- Snio (merci pour l'aide sur les ressources)
- et enfin Neitsa qui est l'une des personnes pour qui j'ai le plus d'estime. En effet c'est pour moi la personne qui se rapproche le plus d'un sage : grandes connaissances, toujours prêt à aider et surtout c'est quelqu'un qui ne s'empporte pas mais sait se montrer ferme (Neitsa si par hasard tu lisais ces lignes, va pas croire que je te lèche le cul, c'est juste l'impression que tu me donne. On dis souvent aux gens quand ils nuls, rarement quand ils sont intéressants).

23/11/05
mastermatt29