LA BIBLE

DU

CRACKEUR

VERSION 0.2 BETA

Avril 98

Mis en fo
Mis en fo

AVANT PROPOS :

Idée, Creation, Mise en page, Diffusion web : MOVAX1St [DSK],
Participation : Mister X, Dr Julo, Cyberbobjr [DSK]...

Ces textes sont ecrits et diffusés à but d'etudes & analyses. Toute autre utilisation vous met en violation avec le code de la propriété intellectuelle, également diffusés sur le Web. Ces textes ne sont pas ecrits pour une diffusion sur le Web, ce qui explique la mauvaise presentation du site.

Ces textes sont ecrits avec des fautes d'ortographe, et avec les techniques de parlote du web.
Ainsi « que » peut devenir « ke » etc...
Les symboles du web seront aussi utilisés, sans vous les commenter.

Ces textes sont GRATUITS et en aucun cas vous ne devez payer pour les avoir, que ce soit sur cd (livrés avec les fichiers) ou sur papier. Ils sont de libre diffusion, mais, par principe de correction, je demanderais à ce qu'il ne soit pas modifier. Vous pourrez telecharger les mises à jour de cette bible sur :
http://www.altern.org/biblecrack/bc.ZIP
Au format. DOC, lisible sous Word.
Pour des raisons que vous comprendrez aisément, les fichiers étudiés ne sont pas livrables sur ce site.

Si vous possedez des documents permettant de faire evoluer cette bible, et que vous souhaitiez me les faire parvenir, mailto : movax1@mailexcite.com . Ces docs se doivent d'etre integralement en francais, pour le bien etre des puristes francophones, dont je fais partie ;).

Pour des raisons de simplicité, de gestion... les cracks se verront annotés comme ceci :

A : Débutant
B : Débutant confirmé
C : Moyen
D : Bon
E : Réservé aux decryptage d'algos, dongles etc...

Je demanderais donc à tous les auteurs d'inscrire leurs notes en fonction de ce shéma.
Merci de votre compréhension.

Assez papoter, a l'action.

Sommaire :

ASM, une definition.

L'assembleur est un langage de programmation de tres bas niveaux. De ce fait, il est un peu plus dur que les autres a apprendre et cerner. Il retransmet au processeur des données qu'il peut comprendre, données-elle meme transformée en langage machine. L'assembleur est en quelque sorte une surcouche du langage machine. Ce dernier langage utilise des chiffres 1 & 0 pour communiquer. Imaginez-vous en tapant un courrier avec des 1 & 0 a la place du bonjour. Temps perdus et risquers de fautes.

L'hexadecimal :
.....

Ce systeme est basés sur l'utilisation des chiffres et de certaines lettres de l'alphabt (de A a F).
Nous connaissons tous le systeme decimal, base 10. L'hexadecimal lui est sur une base de 16.
Mais comment fait-on pour compter ?
Tres simplement, de meme qu'apres 10 il y a 11 dans notre systeme decimal de tous les jours,
En hexa, 9 s'ecrit 09, et 10 ? Comment le calculer ? Nous utileserons les lettres. 10 = 0A, 11 a 0B
Et ainsi de suite jusqu'à 0F, qui lui represente 15. Donc 10 en hexa = 16. C'est le principe qui est à retenir.
Nous aurons l'occasion de revoir l'hexa au cours de ces chapitres, beaucoup plus en details.

Le Binaire :
.....

Comme on l'a vu plus haut, les ordinateurs calculent en base de 2, de 0 a 1, suivant la valeur désirée.
Chaque 0 ou 1 est appelé un bit, 8bits forment un octets. Pour le nombre 1 nous avons 00000001b (le B signifie binaire), un nombre binaire se decompose ainsi :

128|64|32|16|8|4|2|1

Pour le nombre 1, la premiere case est remplis, (1=remplis). On se fait le 2 : 00000010, et pour le 3, pendant qu'on y est : 00000011. (1ere case =1, 2eme case =2, 1+2=3). Le nombre 11111111 donnera donc 255, car l'addition de 1+2+4+8+16+32+64+128 = 255. Plus il y a de bits, plus le nombre est grand. J'espere que vous avez saisi, bien que cela n'aie pas une importance majeure pour les petits cracks.

Le processeur : Registres generaux & segments.
.....

Le processeur est composé de différentes parties dont les registres, elements les plus importants de notre processeur. Il existe plusieurs types de registre, registre generaux, d'etat, de segments...
Nous avons plusieurs registres generaux, qui commencent par A, B, C, D.
Ces quatres registres sont tres important. Nous avons AX (16bits) qui se divise en deux petits registres 8 bits, AL (L=low=bas) et AH (h=high=haut). Nous avons BX (BL et BH), CX (CL et CH) et DX (DL et DH).
Nous rajoutons un E devant les registres pour obtenir du 32 bits (EAX, EBX, ECX, EDX). Notons que nous ne pouvons avoir de EAH ou ECL. Ces registres peuvent recevoir une valeur correspondante a leur capacité. AX=65535 au maximum et AL =255 au maximum. La partie haute du registre ne peut pas etre modifiés séparément.
Les registres pointeurs eux sont DI (Destination Index), SI (Source Index), BP (Base Pointer), IP (Isnstruction Pointer) et SP (Stack Pointer). Ces registres n'ont pas de partie 8 bits. Ils peuvent etre etendus a 32 bits, en rajoutant un E. les registres SI et DI sont employés pour les instructions de chaines et le registre SP est utilisé lors d'instructions de la pile. Nous verrons cela si ca s'avere necessaire.

Pour pouvoir chercher et placer des choses dans sa memoire, un ordinateur a besoin de ce qu'on appelle une adresse. Celle ci se compose de deux parties de 32 bits. La premiere est le segment et la deuxieme l'offset. Un exemple : 0A000H :00000H (adresse de la memoire video). Les registres de segments ne sont lus et ecrits que sur 16 bits. Le 386 utilise un offset de 32 bits, mais cela ne change rien vu que la modification du segment ne se fait que sur 16 bits. Il y a six segments a partir du 386 : CS (Code Segment), SS (Stack Segment), DS (Data Segment), ES (Extra Segment), FS & GS (seulement 386).

Quelques instructions :
.....

MOV : instruction qui sert à placer (et non pas a deplacer). Elle necessite deux operandes (variables) qui sont la destination & la source. Ceux ci peuvent etre des registres genereaux ou des emplacements memoire. Cependant, Les opérandes ne peuvent pas etre des toutes les deux des emplacements memoire. La destination ne peut pas non plus etre ce qu'on appelle une valeur immédiate. (Les nombres sont des valeurs immediates), donc la valeur MOV 30,AX est sans sens. Les operandes doivent etre de la meme taille, donc pas de MOV AX,AL.

Ex d'insntructions MOV :
MOV AL, BL (contenu de bl dans cl) pourrait faire si AL=5 & BL=15, nous avons AL=15, BL=15

MOV CX, ES : [DI] place dans CX le contenu 16bits de l'emplacement ES : [DI]

MOV ECX, ES : [DI] (place le contenu 32bits de ES : [DI] dans ecx)
Etc... a vous de vous exercer...

Retenons que l'instruction MOV est comme le signe =.

ADD : sert à additionner et nécéssite 2 opérandes. Une source, et une destination.
La destination prendra la valeur de sources + destination. Les regles de combinaison sont semblables a celles de MOV si ce n'est que l'utilisation des registres de segments est impossible.
Exemples d'ins :

ADD BX, CX pourrait faire si BX=12 & CX =20, nous avons BX=32 et CX=20...
C'est la meme chose pour le 32bits :
ADD EBX,ECX = prendre le contenu d'ecx pour l'additioner a ebx.
Simple non ?

JMP : Jump.. Ai-je besoin de l'expliquer ? ? oui.. bon..

L'instruction JMP sert à modifier l'ip (eip) en consequence changer d'instruction.
JMP 4000 aura pour effet de passer non pas a l'instruction 4000 mais d'aller à l'adresse memoire 4000.
Souvenons-nous qu'une instruction n'est pas égale à une instruction mémoire...

CMP, JZ, JNZ : Ha, celles la, je ne sais pas vraiment comment les decrirent.. Bien sur, je pourrais ouvrir un bouquin mais, j'ai une preference pour le travail de tete... j'vais essayer..
CMP sert à comparer une valeur, souvent pour les sauts conditionnels, ou inconditionnels.
Ecrivons un petit texte asm pour eclaircir tout ca.

MOV AX, 16
INC AX
CMP AX, 32
JZ xxxx (les x sont des adresses)

A cette partie du prog, nous avons donc le CMP AX, 32 puis un JZ.
Le cmp va tester ax et modifier en consequences les flags...
Le test se fait grace au JZ (saute si egalité), donc tant que ax n'est pas egale a 32, on est balancé à l'adresse renvoyer par le jz. Si nous avions eu un JNZ, notre programme n'aurait pas soter. (JNZ ne sote pas si valeur egale)
Cela est suffisant sur le moment pour que vous compreniez l'usage des CMP, JZ & JNZ.
Nous aurons certainement l'occasion d'etudier ca de plus pres....

NOP : Supprime l'operation..
Dans notre exemple ci-dessus, si nous remplacons le JZ par 2 NOP, le programme continue.

Il y a bien d'autres instructions, mais sur une base de départ, ces derniers suffisent.
Nous reviendrons aisément sur les instructions toutes au long de ces tutoriaux...

La table des caracteres ascii
(ASCII Characters)

Dec	Hex	Char	Code
0	00	€	NUL
1	01	€	SOH
2	02	€	STX
3	03	€	ETX
4	04	€	EOT
5	05	€	ENQ
6	06	€	ACK
7	07	€	BEL
8	08	€	BS
9	09	€	HT
10	0A	€	LF
11	0B	€	VT
12	0C	€	FF
13	0D	€	CR
14	0E	€	SO
15	0F	€	SI
16	10	€	SLE
17	11	€	CS1
18	12	€	DC2
19	13	€	DC3
20	14	€	DC4
21	15	€	NAK
22	16	€	SYN
23	17	€	ETB
24	18	€	CAN
25	19	€	EM
26	1A	€	SIB

27	1B	€	ESC
28	1C	€	FS
29	1D	€	GS
30	1E		RS
31	1F	€	US
32	20	(space)	
33	21	!	
34	22	"	
35	23	#	
36	24	$	
37	25	%	
38	26	&	
39	27	'	
40	28	(	
41	29	)	
42	2A	*	
43	2B	+	
44	2C	,	
45	2D	-	
46	2E	.	
47	2F	/	
48	30	0	
49	31	1	
50	32	2	
51	33	3	
52	34	4	
53	35	5	
54	36	6	
55	37	7	
56	38	8	
57	39	9	
58	3A	:	
59	3B	;	
60	3C	<	
61	3D	=	
62	3E	>	
63	3F	?	
64	40	@	
65	41	A	
66	42	B	
67	43	C	
68	44	D	
69	45	E	
70	46	F	
71	47	G	
72	48	H	
73	49	I	
74	4A	J	
75	4B	K	
76	4C	L	
77	4D	M	
78	4E	N	
79	4F	O	
80	50	P	
81	51	Q	
82	52	R	
83	53	S	
84	54	T	
85	55	U	

86	56	V
87	57	W
88	58	X
89	59	Y
90	5A	Z
91	5B	[
92	5C	\
93	5D	]
94	5E	^
95	5F	_
96	60	`
97	61	a
98	62	b
99	63	c
100	64	d
101	65	e
102	66	f
103	67	g
104	68	h
105	69	i
106	6A	j
107	6B	k
108	6C	l
109	6D	m
110	6E	n
111	6F	o
112	70	p
113	72	q
114	72	r
115	73	s
116	74	t
117	75	u
118	76	v
119	77	w
120	78	x
121	79	y
122	7A	z
123	7B	{
124	7C	|
125	7D	}
126	7E	~
127	7F	□
128	80	€
129	81	•
130*	82	‚
131*	83	*f*
132*	84	„
133*	85	…
134*	86	†
135*	87	‡
136*	88	ˆ
137*	89	‰
138*	8A	Š
139*	8B	‹
140*	8C	Ꞓ
141	8D	•
142	8E	Ţ
143	8F	•

144	90	•
145	91	
146	92	
147*	93	
148*	94	
149*	95	
150*	96	
151*	97	
152*	98	˜
153*	99	™
154*	9A	š
155*	9B	›
156*	9C	œ
157	9D	•
158	9E	ţ
159*	9F	Ÿ
160	A0	
161	A1	¡
162	A2	¢
163	A3	£
164	A4	¤
165	A5	¥
166	A6	¦
167	A7	§
168	A8	¨
169	A9	©
170	AA	ª
171	AB	«
172	AC	¬
173	AD	-
174	AE	®
175	AF	¯
176	B0	°
177	B1	±
178	B2	²
179	B3	³
180	B4	´
181	B5	µ
182	B6	¶
183	B7	·
184	B8	¸
185	B9	¹
186	BA	º
187	BB	»
188	BC	¼
189	BD	½
190	BE	¾
191	BF	¿
192	C0	À
193	C1	Á
194	C2	Â
195	C3	Ã
196	C4	Ä
197	C5	Å
198	C6	Æ
199	C7	Ç
200	C8	È
201	C9	É
202	CA	Ê

203	CB	Ë
204	CC	Ì
205	CD	Í
206	CE	Î
207	CF	Ï
208	D0	Ð
209	D1	Ñ
210	D2	Ò
211	D3	Ó
212	D4	Ô
213	D5	Õ
214	D6	Ö
215	D7	×
216	D8	Ø
217	D9	Ù
218	DA	Ú
219	DB	Û
220	DC	Ü
221	DD	Ý
222	DE	Þ
223	DF	ß
224	E0	à
225	E1	á
226	E2	â
227	E3	ã
228	E4	ä
229	E5	å
230	E6	æ
231	E7	ç
232	E8	è
233	E9	é
234	EA	ê
235	EB	ë
236	EC	ì
237	ED	í
238	EE	î
239	EF	ï
240	F0	ð
241	F1	ñ
242	F2	ò
243	F3	ó
244	F4	ô
245	F5	õ
246	F6	ö
247	F7	÷
248	F8	ø
249	F9	ù
250	FA	ú
251	FB	û
252	FC	ü
253	FD	ý
254	FE	þ
255	FF	ÿ

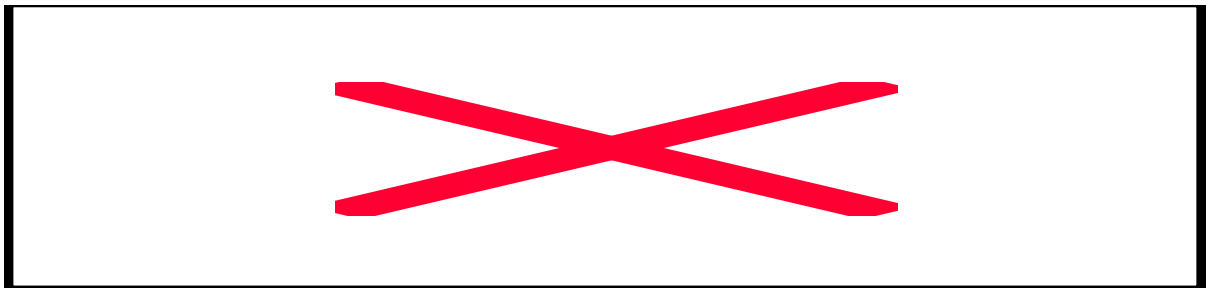
*Windows seulement. Voilà qui pourrait vous dépanner, si vous ne l'aviez pas !

Voilà qui complete ma petite intro.
Passont à notre premier crack... tout de suite.. ;))

EASY SOFT CD MENU GENERATOR VERS. 2.16 Crack de niveaux A
Par MOVAX1St [DSK_CREW]

Materiel nécéssaires :
Un editeur HEXADECIMAL, style Hex Works Shop, Hiew etc....

Ce programme à pour but de vous aider à préparer vos compiles sur cd...
Compression, présentation etc...
En sa version SHAREWARE, il a cependant un inconvénient majeur...
Des que vous générez un joli menu, UREGISTRERET VERSION apparaît dans tous les sens...
Sur chacune des fenetres générés, 3 en l'occurrence.
Notre but est donc de supprimer ce stupide message de ce stupide logiciel...
Nous l'installons et avons ces fichiers executables :
Lavmenu.exe, Menu.exe, et 2 autres qui ne nous interessent pas.
Si nous lancons le Menu.exe, une erreur apparaît : ERROR : File not Found [2] Creating menu.tps\ !farver etc..
C'est n'est pas celui la.. On lance Lavmenu.exe, et la tout se passe bien. On fait un peu n'importe quoi, le but etant de generer les fichiers... je ne m'attarde pas sur les commandes de ce log, on est pas la pour ca.
Nous generons notre fichier et le lancons. Voilà notre fameux UREGISTRERET VERSION partout..
(Avec un nom comme ca, on ne peut pas s'empecher de le cracker ;)).
On quitte et on regarde les fichiers générés.. autorun.inf, qui lui meme fait appel au fichier menu.exe present aussi dans ce repertoire... on en conclue donc qu'il colle son Menu.exe ici dans ce reperoire. Ca doit etre ca.
Prenez votre editeur hexadecimal, et ouvrez le fichier Menu.exe du rep d'install du log..
Faites Find, Recherchez UREGIS (c'est pas la peine de mettre ca au complet..)



PAF ! ! ! On le tient. Il est-la.. modifiez ca par des espaces (20 en ascii), continuez la recherche...
L'operation se fera 3 fois, une fois tout remplacés par des espaces, enregistrez, generez un fichier. BAK (Habitude a prendre, ca sert si on abime un file) et relancez le Lavmenu.exe. Ce dernier nous dit tjrs u.. machin, mais le fichier menu lui, ne laisse plus apparaître UREGIS..... nulle part.. Cool worrks ! Vous venez de faire votre premier crack ;))) Felicitations ;)
Bien sur, il y a bien d'autres methodes pour y arriver, mais celle ci me semble la plus simple, fo avouez que.....
Ceci dis, les programmeurs de ce log auraient pus avoir la bonté de cryptée ca ! Ca nous aurions fait un cours de niveaux B..
Amitiés à tous(tes)
MOVAX1St [DSK]

Crack de XARA3D 2 Par MOVAX1St [DSK] Cours de niveaux A+

Matos nécéssaires : (utilisés)
Wdasm85 ou 89
Soft-ice 3.20
Editeur Hexa

Nous voici donc sur un autre travail, un peu plus compliqué que le dernier , du moins il semble....
Voyons ca de plus près : ce logiciel sert à générer des images gif, jpg animés. (sympas d'ailleurs)
En version SHAREWARE, il nous permets de rien foutre ! ! ! ils se foutent de nous !
En effet, des que le logiciel est lancés, il vous annonce qu'il est la que pour 15 jours, sinon, fo vous enregistrer.
De plus, il nous colle un fond avec xara demo ecrit partout , donc, inutilisable !

La solution pourrait d'etre de virer le time-limit, puis de virer le fond xara demo.. mais bon, y'a au lancement un bouton UNLOCK, ce qui veut dire qu'on peut tout dévérouiller. ;))))
(personnelement, j'utilise une technique autre que celle que je vais vous présenter pour enregistrer ce genre de log, mais, pour un bon debut, j'ai préféré vous exposer cette solution.)
Dessasemblons le logiciel, avec WDASM. 2 minutes et c'est fait. Relancons le X3D.exe.
Cliquons sur purchase, il nous demande un code d'enregistrement, entrons n'importe quoi. UNLOCK.
La, le log nous réponds : YOU ENTERED AN INVALID UNLOCK CODE, THE.....
Notons ca... retournons sous WDASM. Recherchons (Search puis Find text) YOU ENTERED, le log nous envoie a ca :

```
 * Possible Reference to String Resource ID=03005: "You entered an invalid
unlock code.
The program has not been"
```

Ha ! un pas en avant.... on sait deja que ca apparaît en clair dans le texte. Nous savons egalement que notre seance de test est deja passé car il nous dis que c'est invalide.. nous devons donc remonter le code source.
Remontons doucement, en regardant ce qui se fait.. JE OU JNE 0040BE3D... cet instruction apparaît bien souvent.. 0040BD0F, a cet adresse on remarque une serie d'instruction qui laisse penser que le log fait un calcul à cet endroit, de plus a la fin de calcul, nous avons droit au JE 0040BE3D, qui nous dis au revoir....
Remontons au plus haut possible, juste avant l'instruction jne 0040BE3D. Posons un bpx sur le CALL en 40BC34. (Bpx 40BC34 sous SOFT-ICE)
Bingo, le log s'arrete... L'astuce consiste maintenant a lui faire croire qu'on est enregistré..
Le passage sur le JNE 40BE3D nous renvoie apres cette serie de calcul si le sot se fait..
Donc, nous pourrions tres bien modifier le sot par 2 NOP (a+entree sous softice, puis nop, nop)
Mais cela n'a que tres peu d'interet.. en effet, il nous faudrait mettre des nop sur tous les JE ou JNE
Se referencant a cet adresse.. le plus simple consiste a repérer le dernier de ces sots :
0040BD80 0F85B7000000 jne 0040BE3D
Puis, a placer un sot sur le call d'en dessous.
0040BD86 E8A4360700 call 0047F42F
plus loin nous voyons :

```
* Possible StringData Ref from Data Obj ->"Key"
```

Etc'est a partir de la qu'il nous la sort cette key, clé de validation ;)))
Donc, on prends le premier des sauts en : :0040BC4D 0F85EA010000 jne 0040BE3D
Qu'on remplace par un JMP a l'adresse du call ... ca nous donne : 0040BC4D JMP 0040BD86
Je ne vous ai pas mis les codes hexa volontairements ;)) a vous de les chercher 8=((
L'editeur HEXA vous aidera a réaliser ce crack (modifications du log)
C'est pas du clefs en main... Les instructions du dessous vous eviteront d'avoir à décompiler le log...
Puis, il fo bien remplir cette bible ;)
Si vous voulez le refaire vous mêmes après avoir suivi l'exemple, supprimez la key dans la base de registre...
Nous aurions pu essayer de trouver le reg attendu, mais, cela ne sert a rien sur ce genre de crack...

```
:0040BC25 52                    push edx
:0040BC26 6A06                  push 00000006
:0040BC28 E8911B0600            call 0046D7BE
:0040BC2D 8D8C24A8000000        lea ecx, dword ptr [esp+000000A8]
:0040BC34 E81B080600            call 0046C454
:0040BC39 83F801                cmp eax, 00000001
:0040BC3C 0F8509020000          jne 0040BE4B
:0040BC42 8B842440010000        mov eax, dword ptr [esp+00000140]
:0040BC49 8378F807              cmp dword ptr [eax-08], 00000007
:0040BC4D 0F85EA010000          jne 0040BE3D<= Premier Sot
:0040BC53 0FBE10                movsx edx, byte ptr [eax]
:0040BC56 52                    push edx
:0040BC57 E864640400            call 004520C0
:0040BC5C 83C404                add esp, 00000004
:0040BC5F 85C0                  test eax, eax
:0040BC61 0F84D6010000          je 0040BE3D
:0040BC67 8B842440010000        mov eax, dword ptr [esp+00000140]
:0040BC6E 0FBE4801              movsx ecx, byte ptr [eax+01]
:0040BC72 51                    push ecx
```

```
:0040BC73 E848640400            call 004520C0
:0040BC78 83C404                add esp, 00000004
:0040BC7B 85C0                  test eax, eax
:0040BC7D 0F84BA010000          je 0040BE3D
:0040BC83 8B942440010000        mov edx, dword ptr [esp+00000140]
:0040BC8A 0FBE4202              movsx eax, byte ptr [edx+02]
:0040BC8E 50                    push eax
:0040BC8F E82C640400            call 004520C0
:0040BC94 83C404                add esp, 00000004
:0040BC97 85C0                  test eax, eax
:0040BC99 0F849E010000          je 0040BE3D
:0040BC9F 8B8C2440010000        mov ecx, dword ptr [esp+00000140]
:0040BCA6 0FBE5103              movsx edx, byte ptr [ecx+03]
:0040BCAA 52                    push edx
:0040BCAB E810640400            call 004520C0
:0040BCB0 83C404                add esp, 00000004
:0040BCB3 85C0                  test eax, eax
:0040BCB5 0F8482010000          je 0040BE3D
:0040BCBB 8B842440010000        mov eax, dword ptr [esp+00000140]
:0040BCC2 0FBE4804              movsx ecx, byte ptr [eax+04]
:0040BCC6 51                    push ecx
:0040BCC7 E8F4630400            call 004520C0
:0040BCCC 83C404                add esp, 00000004
:0040BCCF 85C0                  test eax, eax
:0040BCD1 0F8466010000          je 0040BE3D
:0040BCD7 8B942440010000        mov edx, dword ptr [esp+00000140]
:0040BCDE 0FBE4205              movsx eax, byte ptr [edx+05]
:0040BCE2 50                    push eax
:0040BCE3 E8D8630400            call 004520C0
:0040BCE8 83C404                add esp, 00000004
:0040BCEB 85C0                  test eax, eax
:0040BCED 0F844A010000          je 0040BE3D
:0040BCF3 8B8C2440010000        mov ecx, dword ptr [esp+00000140]
:0040BCFA 0FBE5106              movsx edx, byte ptr [ecx+06]
:0040BCFE 52                    push edx
:0040BCFF E8BC630400            call 004520C0
:0040BD04 83C404                add esp, 00000004
:0040BD07 85C0                  test eax, eax
:0040BD09 0F842E010000          je 0040BE3D
:0040BD0F 8B842440010000        mov eax, dword ptr [esp+00000140]
:0040BD16 0FBE4804              movsx ecx, byte ptr [eax+04]
:0040BD1A 0FBE5006              movsx edx, byte ptr [eax+06]
:0040BD1E 8D0C49                lea ecx, dword ptr [ecx+2*ecx]
:0040BD21 8D0CCA                lea ecx, dword ptr [edx+8*ecx]
:0040BD24 8D1449                lea edx, dword ptr [ecx+2*ecx]
:0040BD27 0FBE4802              movsx ecx, byte ptr [eax+02]
:0040BD2B 8D0CD1                lea ecx, dword ptr [ecx+8*edx]
:0040BD2E 8D1449                lea edx, dword ptr [ecx+2*ecx]
:0040BD31 0FBE4805              movsx ecx, byte ptr [eax+05]
:0040BD35 8D0CD1                lea ecx, dword ptr [ecx+8*edx]
:0040BD38 8D1449                lea edx, dword ptr [ecx+2*ecx]
:0040BD3B 0FBE08                movsx ecx, byte ptr [eax]
:0040BD3E 8D0CD1                lea ecx, dword ptr [ecx+8*edx]
:0040BD41 8D1449                lea edx, dword ptr [ecx+2*ecx]
:0040BD44 0FBE4801              movsx ecx, byte ptr [eax+01]
:0040BD48 0FBE4003              movsx eax, byte ptr [eax+03]
:0040BD4C 8D0CD1                lea ecx, dword ptr [ecx+8*edx]
:0040BD4F 8D1449                lea edx, dword ptr [ecx+2*ecx]
:0040BD52 8BCD                  mov ecx, ebp
:0040BD54 81E155555555          and ecx, 55555555
:0040BD5A 8D94D067216BFB        lea edx, dword ptr [eax+8*edx-0494DE99]
```

```
:0040BD61 8BC5                    mov eax, ebp
:0040BD63 D1E8                    shr eax, 1
:0040BD65 2555555555              and eax, 55555555
:0040BD6A 8D0C48                  lea ecx, dword ptr [eax+2*ecx]
:0040BD6D 8D0489                  lea eax, dword ptr [ecx+4*ecx]
:0040BD70 C1E008                  shl eax, 08
:0040BD73 2BC1                    sub eax, ecx
:0040BD75 8D04C0                  lea eax, dword ptr [eax+8*eax]
:0040BD78 8D0441                  lea eax, dword ptr [ecx+2*eax]
:0040BD7B 8D0C40                  lea ecx, dword ptr [eax+2*eax]
:0040BD7E 3BD1                    cmp edx, ecx
:0040BD80 0F85B7000000            jne 0040BE3D <= dernier SOT
:0040BD86 E8A4360700              call 0047F42F<= VALIDATION
:0040BD8B 8B4004                  mov eax, dword ptr [eax+04]
:0040BD8E 55                      push ebp

* Possible StringData Ref from Data Obj ->"Key"
                                  |
:0040BD8F 684CA44A00              push 004AA44C

* Possible StringData Ref from Data Obj ->"Install"
                                  |
:0040BD94 6844A44A00              push 004AA444
:0040BD99 8BC8                    mov ecx, eax
:0040BD9B E8A3A30600              call 00476143
:0040BDA0 C605C0114C0000          mov byte ptr [004C11C0], 00
:0040BDA7 8D8C2444010000          lea ecx, dword ptr [esp+00000144]
:0040BDAE C68424B40400000B        mov byte ptr [esp+000004B4], 0B
:0040BDB6 E87B150600              call 0046D336
:0040BDBB 8D8C2440010000          lea ecx, dword ptr [esp+00000140]
:0040BDC2 C68424B40400000A        mov byte ptr [esp+000004B4], 0A
:0040BDCA E867150600              call 0046D336
:0040BDCF 8D8C2404010000          lea ecx, dword ptr [esp+00000104]
:0040BDD6 C68424B404000009        mov byte ptr [esp+000004B4], 09
:0040BDDE E89FF40600              call 0047B282
:0040BDE3 8D8C24A8000000          lea ecx, dword ptr [esp+000000A8]
:0040BDEA C68424B404000003        mov byte ptr [esp+000004B4], 03
:0040BDF2 E8FD020600              call 0046C0F4
:0040BDF7 8D4C2458                lea ecx, dword ptr [esp+58]
:0040BDFB C68424B404000002        mov byte ptr [esp+000004B4], 02
:0040BE03 E868630500              call 00462170
:0040BE08 8D4C2458                lea ecx, dword ptr [esp+58]
:0040BE0C E86F640500              call 00462280
:0040BE11 8D4C241C                lea ecx, dword ptr [esp+1C]
:0040BE15 C68424B404000000        mov byte ptr [esp+000004B4], 00
:0040BE1D E814150600              call 0046D336
:0040BE22 8D4C2414                lea ecx, dword ptr [esp+14]
:0040BE26 C78424B4040000FFFFFFFF  mov dword ptr [esp+000004B4], FFFFFFFF
:0040BE31 E800150600              call 0046D336
:0040BE36 B001                    mov al, 01
:0040BE38 E9AA000000              jmp 0040BEE7

* Referenced by a (U)nconditional or (C)onditional Jump at Addresses:
|:0040BC4D(C), :0040BC61(C), :0040BC7D(C), :0040BC99(C), :0040BCB5(C)
|:0040BCD1(C), :0040BCED(C), :0040BD09(C), :0040BD80(C)
|
:0040BE3D 6AFF                    push FFFFFFFF
:0040BE3F 6A10                    push 00000010

* Possible Reference to String Resource ID=03005: "You entered an invalid
unlock code.
```

```
The program has not been"
                                    |
:0040BE41 68BD0B0000                push 00000BBD
:0040BE46 E8808F0600                call 00474DCB

* Referenced by a (U)nconditional or (C)onditional Jump at Address:
|:0040BC3C(C)
|
:0040BE4B 8A442413                  mov al, byte ptr [esp+13]
:0040BE4F 33DB                      xor ebx, ebx
:0040BE51 84C0                      test al, al
:0040BE53 0F94C3                    sete bl
:0040BE56 8D8C2444010000            lea ecx, dword ptr [esp+00000144]
:0040BE5D C68424B40400000E          mov byte ptr [esp+000004B4], 0E
:0040BE65 E8CC140600                call 0046D336
:0040BE6A 8D8C2440010000            lea ecx, dword ptr [esp+00000140]
:0040BE71 C68424B40400000D          mov byte ptr [esp+000004B4], 0D
:0040BE79 E8B8140600                call 0046D336
:0040BE7E 8D8C2404010000            lea ecx, dword ptr [esp+00000104]
:0040BE85 C68424B40400000C          mov byte ptr [esp+000004B4], 0C
:0040BE8D E8F0F30600                call 0047B282
:0040BE92 8D8C24A8000000            lea ecx, dword ptr [esp+000000A8]
:0040BE99 C68424B404000003          mov byte ptr [esp+000004B4], 03
:0040BEA1 E84E020600                call 0046C0F4
:0040BEA6 8D4C2458                  lea ecx, dword ptr [esp+58]
:0040BEAA C68424B404000002          mov byte ptr [esp+000004B4], 02
:0040BEB2 E8B9620500                call 00462170
:0040BEB7 8D4C2458                  lea ecx, dword ptr [esp+58]
:0040BEBB E8C0630500                call 00462280
:0040BEC0 8D4C241C                  lea ecx, dword ptr [esp+1C]
:0040BEC4 C68424B404000000          mov byte ptr [esp+000004B4], 00
:0040BECC E865140600                call 0046D336
:0040BED1 8D4C2414                  lea ecx, dword ptr [esp+14]
:0040BED5 C78424B4040000FFFFFFFF    mov dword ptr [esp+000004B4], FFFFFFFF
:0040BEE0 E851140600                call 0046D336
:0040BEE5 8AC3                      mov al, bl

* Referenced by a (U)nconditional or (C)onditional Jump at Addresses:
|:0040B3DD(U), :0040B485(U), :0040BA16(U), :0040BAB2(U), :0040BE38(U)
|
:0040BEE7 8B8C24AC040000            mov ecx, dword ptr [esp+000004AC]
:0040BEEE 5F                        pop edi
:0040BEEF 5E                        pop esi
:0040BEF0 5D                        pop ebp
:0040BEF1 64890D00000000            mov dword ptr fs:[00000000], ecx
:0040BEF8 5B                        pop ebx
:0040BEF9 81C4A8040000              add esp, 000004A8
:0040BEFF C3                        ret
```

Bon courage ;))
MOVAX1St [DSK]

A peine finis ce petit exercices que je recois quelques textes, que je vous retransmets....
A etudier.

A propos de Paint Shop Pro 5.0 FINAL RELEASE (fichier psp.exe -> 3 612 672 octets)
La version finale étant sortie, je me suis donc attelé à la tâche...

Je passe rapidement sur le crack de la partie Time Limit, c'est facile et CyberBobJr a déjà détaillé le processus...
Pour info, j'ai personnellement patché en :

585C84 : PUSH 02 (6a 02)
POP EDI (5f)
DEC EDI (4f)

Manipulation visant à fixer EDI à 01 (et à remplir tous les octets remplacés). Soit 1 jour d'utilisation pour l'éternité (enfin jusqu'à la version 5.01 qui va sortir dans 3 jours n'en doutont pas...ptet même avant la fin de ce texte).
Bon maintenant le NAG SCREEN, le morceau de choix du crack...
J'ai donc lu avec attention le tutorial d'+ORC sur le crack des différents PSP (recommandé par Ethan je crois).

Personnellement, shunter le NAG SCREEN me paraît très difficile ici, car d'importantes routines nécessaires à l'initialisation de PSP sont incluses dans la routine d'affichage du NAG... un peu en dessous d'ailleurs de l'endroit où l'on a patché pour le Time Limit (la vérif s'effectuant juste avant affichage des infos dans la boîte de dialogue). Donc shunter le NAG, c'est forcément ne pas exécuter ces routines. y'aura donc un manque quelque part...

Je me suis donc orienter vers la bonne veille méthode de simulation d'action sur le (faux) bouton START (cf version 4.14 de PSP).
Je ne vais pas détailler ici la procédure car on s'aperçoit vite qu'elle ne sert à rien !
Le NAG se ferme bien mais après impossible d'ouvrir le moinde fichier (les extensions de files dans la boite OpenFileName sont vierges), ou même d'en créer un nouveau (pas assez de mémoire me dit le prog ;)...

Bref, il y a en plus une vérif quelque part...

Réfléchissons : le NAG est ouvert, affiché et fermé comme si j'appuyais sur le bouton START. Pourquoi ça ne marche pas ?
Il se passe visiblement quelque chose entre le moment où le NAG est affiché et celui où l'utilisateur est censé le fermer par START.
Humm c'est ici que je suis parti du postulat suivant : PSP doit déclarer un TIMER pour la fenêtre du NAG, et ce TIMER va exécuter une (ou plusieurs) routines indispensables par la suite.
Le fait de forcer la fermeture du NAG par "simulation" du bouton START ne permet pas à ces fonctions d'être exécutées puisque le TIMER n'a pas eu matériellement le temps d'envoyer l'info...
(en fait ce n'est pas vraiment ce qui se passe, on le verra plus loin mais j'ai gardé ce passage pour montrer le raisonnement de départ).

Donc à partir de là, que fait-on ?

On repère le HANDLE de la fenêtre du NAG (je vous conseille WINSIGHT pour ce genre de truc).
Après soit on pose un BMSG Handle-du-Dessus WM_TIMER, et on galère...
Soit on utilise encore WINSIGHT pour étudier justemment les MESSAGES WINDOWS envoyés vers le NAG.
On sélectionne WM_TIMER comme message (non ce n'est pas un hasard ;) et on regarde ce qui se passe...
Et surprise, on a juste UN et UN SEUL WM_TIMER d'envoyer à la fenêtre NAG même si on ne la ferme pas...

Donc la routine appelée lorsque le NAG reçoit le WM_TIMER va contenir un KILLTIMER, API Windows servant à détruise un Timer.
Pourquoi est-ce important ?
Tout simplement parce que maintenant nous connaissons une API sur laquelle poser un bPX et contenue dans la routine que l'on cherche....

NOTE : Habituellement les KILLTIMER sont plutôt exécutés dans la fonction gérant la fermeture de la fenêtre. Voilà pourquoi, il n'était pas forcément évident d'avoir un KILLTIMER dans la routine exécuté par le TIMER même...(c'était pour la petit histoire).

Vous allez avoir plusieurs break dû à votre BPX KILLTIMER (Plusieurs modules de windows utilisant en permanence les Timers eux-aussi).
Mais avec un peu de pot (qu'est que la chance vient foutre là, on se le demande), vous attérirez en 586240.
Allez pour une fois, je détaille (un peu) la routine :

```
:00586240 8B442404              mov eax, dword ptr [esp+04]
:00586244 83EC18                sub esp, 00000018
:00586247 83F807                cmp eax, 00000007
:0058624A 56                    push esi
:0058624B 57                    push edi
:0058624C 8BF1                  mov esi, ecx
:0058624E 0F8518010000          jne 0058636C <-- celui là ne nous intéresse pas, on s'en tape...
:00586254 A178306300            mov eax, dword ptr [00633078]
:00586259 8B5620                mov edx, dword ptr [esi+20]
:0058625C 8B4874                mov ecx, dword ptr [eax+74]
:0058625F 51                    push ecx
:00586260 52                    push edx  <- HANDLE de la fenêtre de NAG SCREEN
:00586261 FF1528CB6300          Call USER32.KillTimer  <-- notre fameux KILLTIMER
:00586267 A170306300            mov eax, dword ptr [00633070]   <- première vérif
:0058626C 85C0                  test eax, eax
:0058626E 0F85DF000000          jne 00586353   <-- c'est le mauvais chemin ça...
```

* Possible StringData Ref from Data Obj ->"5000" <-- ici début d'une jolie vérif qui va regarder si on est bien arriver là

```
:00586274 68A0EC6200            push 0062ECA0     <-- après 5 secondes et pas avant...
:00586279 C7057030630001000000  mov dword ptr [00633070], 00000001
:00586283 FF15ECC76300          Call MSVCRT.atoi
:00586289 8B0D7C306300          mov ecx, dword ptr [0063307C]
:0058628F 83C404                add esp, 00000004
:00586292 8D3C08                lea edi, dword ptr [eax+ecx]
:00586295 FF15A8B56300          Call GetTickCount
:0058629B 3BC7                  cmp eax, edi
:0058629D 0F82B0000000          jb 00586353                 <-- sinon bye bye
:005862A3 8B5620                mov edx, dword ptr [esi+20]
:005862A6 52                    push edx
:005862A7 FF15B4C96300          Call IsWindowVisible <-- Haha...ici on vérifie si la fenêtre du NAG est toujours visible
:005862AD 85C0                  test eax, eax
:005862AF 0F849E000000          je 00586353                 <-- sinon bye bye
:005862B5 8D442408              lea eax, dword ptr [esp+08]
:005862B9 C74424109B010000      mov [esp+10], 0000019B
:005862C1 50                    push eax
:005862C2 C744241859010000      mov [esp+18], 00000159
```

Après on a la gestion du cursor par rapport à la fenêtre (bouton START avec Sablier et cie)

```
:005862CA C744241CF4010000      mov [esp+1C], 000001F4
:005862D2 C744242074010000      mov [esp+20], 00000174
:005862DA FF15F4C96300          Call GetCursorPos
:005862E0 8B5620                mov edx, dword ptr [esi+20]
:005862E3 8D4C2408              lea ecx, dword ptr [esp+08]
:005862E7 51                    push ecx
:005862E8 52                    push edx
:005862E9 FF153CCB6300          Call USER32.ScreenToClient
:005862EF A170306300            mov eax, dword ptr [00633070]
:005862F4 85C0                  test eax, eax
:005862F6 753A                  jne 00586332
:005862F8 8B44240C              mov eax, dword ptr [esp+0C]
:005862FC 8B4C2408              mov ecx, dword ptr [esp+08]
:00586300 50                    push eax
:00586301 8D542414              lea edx, dword ptr [esp+14]
:00586305 51                    push ecx
:00586306 52                    push edx
```

```
:00586307 FF1570CB6300          Call USER32.PtInRect
:0058630D 85C0                  test eax, eax
:0058630F 7421                  je 00586332
:00586311 E81E540500            Call MFC42.MFC42:NoName0884
:00586316 68027F0000            push 00007F02
:0058631B 6A00                  push 00000000
:0058631D FF1520CA6300          Call USER32.LoadCursorA
:00586323 50                    push eax
:00586324 FF1510CB6300          Call USER32.SetCursor
:0058632A 5F                    pop edi
:0058632B 5E                    pop esi
:0058632C 83C418                add esp, 00000018
:0058632F C20400                ret 0004

* Referenced by a (U)nconditional or (C)onditional Jump at Addresses:
|:005862F6(C), :0058630F(C)
|
:00586332 E8FD530500            Call MFC42.MFC42:NoName0884
:00586337 68007F0000            push 00007F00
:0058633C 6A00                  push 00000000
:0058633E FF1520CA6300          Call USER32.LoadCursorA
:00586344 50                    push eax
:00586345 FF1510CB6300          Call USER32.SetCursor
:0058634B 5F                    pop edi
:0058634C 5E                    pop esi
:0058634D 83C418                add esp, 00000018
:00586350 C20400                ret 0004


* Referenced by a (U)nconditional or (C)onditional Jump at Addresses:
|:0058626E(C), :0058629D(C), :005862AF(C)
|
:00586353 C7057030630000000000  mov dword ptr [00633070], 00000000  <-- ici donc la fameuse routine
:0058635D E8D2530500            Call MFC42.MFC42:NoName0884                 1) faisant planter PSP
:00586362 8B4004                mov eax, dword ptr [eax+04]             2) empêchant ouverture et creation
fichiers
:00586365 8BC8                  mov ecx, eax
:00586367 8B10                  mov edx, dword ptr [eax]
:00586369 FF5270                call [edx+70]

* Referenced by a (U)nconditional or (C)onditional Jump at Address:
|:0058624E(C)
|
:0058636C 5F                    pop edi
:0058636D 5E                    pop esi
:0058636E 83C418                add esp, 00000018
:00586371 C20400                ret 0004
```

Bien c'est pas tout ça mais koikon fait donc ?
Bon on sait que cette routine s'exécute, que si le NAG Screen n'est pas affiché au moment de son éxecution (entre autre), on a un saut en 586353 provoquant des erreurs par la suite...
On pourrait donc empêcher les sauts et simuler le bouton START...
Mais pourquoi se faire chier ailleurs puisqu'on a tout sous la main.
Cette routine est exécuté, profitons-en...
Rajoutons à la fin un jolie SENDMESSAGEA(HandleNagScreen, WM_CLOSE, NULL, NULL) et elle se fermera toute seule :)
En plus, on a de la chance, y'a de la place à partir de 586374

D'où rajout du code suivant :

586374 : mov edi,[esi + 20] <-- correspond au HANDLE du NAG...regardez la routine plus haut si vous me croyez pas

```
        PUSH 0    (LPARAM de SendMessageA = NULL)
        PUSH 0    (WPARAM de SendMessageA = NULL)
    PUSH dword ptr 10 (WM_CLOSE = 0x10)
        PUSH EDI  (Handle du Nag, récupéré à l'instant)
    CALL DWORD PTR [0063CA54]  (adresse récupérée dans WDASM correspondant au CALL
USER32!SENDMESSAGEA

    POP EDI
        POP ESI
    ADD ESP,18
    RET 04
```

Et on n'oublie pas de mettre en 58632A et 58634B un JMP 586374 pour terminer la routine par notre code...
Voilà maintenant, on relance, si tout a été fait correctement, plantage monstre !!! (non je rigole normalement ça passe)...
Mais bon voilà que l'on doit attendre quand même 5 secondes avant que notre NAG SCREEN se ferme tout seul...
Solution : on va réduire le délai du TIMER...
Vous avez remarqué plus haut la chaîne 5000 (5000 ms = 5 secondes)...C'est pas un hasard non plus...
ATTENTION : ne faisont pas une erreur de logique : CE N'EST PAS ICI que le programme déclare son TIMER sur 5 secondes.
Si on retrouve les 5000 ms ici, ils sont du à une vérif et non à la déclaration du TIMER (d'ailleurs vous ne voyez pas de focntion SetTimer).
Je fais cette précision car en fait, la chaîne "5000" est utilisée pour la vérif de notre routine mais aussi ailleurs pour la déclaration du TIMER...En fait PSP convertit cette chaine en nombre...
La déclaration du TIMER en lui-même se fait en 584FF8 (avec WDASM c'est facile de s'en rendre compte, grâce au référencement de String, avec SoftIce un peu moins ;). Perso, je n'ai désassembler que pour vous filer le SOURCE. J'ai retrouvé la routine de SETTIMER par une autre méthode (utilisation de SmartCheck entre autre mais ça c'est une autre histoire :).
Comme quoi, on le néglige souvent à partir d'un certain niveau (en toute modestie !! Ne me faite pas dire ce que je n'ai pas voulu dire...) mais un bon vieux DEAD Listing permet souvent de gagner du temps.

Bref concluons... on transforme le 5000 en 500 ce qui parait être une bonne valeur...(modifier la chaine dans l'exe avec un Editeur Hexa - en d'autres termes : remplacer le 30 du dernier "0" par un 00)

Juste un petit mot pour conclure : j'ai présenté ici "ma" méthode, bonne ou mauvaise, peu importe car elle permet finalement de passer en revue pas mal de chose. Et finalement, le crack d'un programme en lui-même n'a que peu d'importance, ce qui compte c'est le savoir (oui mais dans ce cas, on peut dire ça pour chaque crack et on tourne en rond...ouais...c'est vrai tiens...je vais y réfléchir finalement ;).

J'espère que tous ceux ayant utilisé une autre méthode nous la ferons partager.

A+

Mister X (04/98)

Nero Burning 3.0.4.0

Nero utilise une technique de saisie de caractère que l'on recontre assez souvent.
Il n'y a pas de bouton OK ou REGISTER permettant de signaler au prog : "vas y coco verifie mon serial !!"
En fait, le serial est vérifié au fur et à mesure de la saisie.
Ici, il commence par vérifier le nombre de chiffre saisie. Si ce nombre est ok, on va vers la routine de verif en elle-même, sinon, on boucle...Et si la verif est négative, on retourne à la boucle de saisie (et tout ça est totalement transparent pour l'utilisateur).

Ce type de saisie est immédiatement repérable ici dans la mesure où l'on voit bien que le bouton "OK" permettant de fermer la fenêtre est grisé (et ne s'active que si le serial est bon).

(Parfois, avec certains programmes, le système est plus vicieux. On a bien un bouton OK (ou REGISTER), mais la vérif du serial se fait bien aussi au fur et à mesure et quand on appuie sur OK, le programme sait déjà que c'est ok ou non. C'était juste une parenthèse ça...la preuve ->).

Bon tout ceci pour dire qu'on va pas se laisser emmerder par ce genre de truc.
On pose un BPX GetWindowTextA (même pas la peine de se faire chier avec un hmemcpy).
On a 3 break (becoze trois champs de saisi, Name, Organisation et Serial en lui-même).

On s'intéresse qu'au dernier et on trace...
Et on finit par attérir sur la fameuse vérif qui teste si oui ou non on a atteint le nombre de chiffre du serial attendu : ce chiffre c'est 16 (0x10). Il faut donc entrer 16 chiffres pour dépasser enfin la boucle de saisie.
(offset : 43E308)

Juste derrière cette verif, première grosse surprise :

On a en 43E32E un CALL 43E0D0 qui ressemble fort à une routine de verif !!!
On saute dessus ! La routine est relativement classique. Elle vérifie si chaque chiffre du serial trouve sa correspondance dans une table (après une petite soustraction).
Bref on finit par générer un serial (un des nombreux possibles) : 1300202400006115
Essayer pour voir (c'est sans danger :).
On obtient un superbe : "ceci est un numéro piraté etc..." (ou un truc dans le genre).
Visiblement, les auteurs de NERO ont inclus dans la version 3.0.x le moteur de serial des version antérieures (enfin je présume), afin de baiser les pauvres petits malins voulant utiliser leur ancien serial.
J'trouve ça marrant moi ! (il me faut pas grand chose me direz-vous...)

Bref, la vraie routine de verif est forcément derrière celle là...Donc on entre n'importe koi de 16 chiffres et on continue à tracer...
Et on tombe en 43E3A1 qui est le début de la vraie routine...

Que fait-elle ?
Déjà, elle se fout complétement de notre NAME ou de l'Organisation. Elle ne s'occupe que de tester si le serial vérifie justemment certaines conditions.

On a donc un serial de 16 chiffres : 0123456789ABCDEF

0 : doit être à 1 ou 3
1 : doit être supérieur ou égale à 3 (correspondant à la version du logiciel le 3).

Ensuite, les chiffres 1,2 et 4,5,6,7,8,9,A,B vont être utiliser pour générer 5 autres chiffres (par une méthode assez alambiquée il faut bien l'avouer).

C'est 5 chiffres on va les appeler X1 X2 X3 X4 X5
Ensuite le programme vérifie si tout simplement :

3 = X1
C = X2
D = X3
E = X4
F = X5

Ce qui donne comme serials valides (entre autres évidemment) :

1662666666669497
1406000000004114

voilà...
On peut trouver sur le site de MOVAX1St http://www.cyberjunkie.com/movax1 un générateur de Serial pour NERO.
A noter, qu'il ne marche, avec la version 3.0.4.0, que si on entre 10 chiffres commençant par 1 ou 3.

Il foire pour les autres (becoze verif d'au-dessus testant le premier chiffre du serial). Les programmeurs de NERO ont du rajouter cette vérif ultérieurement...Evidemment n'utiliser ce Generator que pour info. Essayer plutôt de trouver votre propre serial avec Softice.

Mister x

A propos de DGPlayer95

Encore un programme en VISUAL BASIC 5...
Mais contrairement à TARIFCOM particulièrement chiant à tracer (car tout se passait dans la DLL), ici on a entre les mains un programme EXE de structure classique.

A ce propos, il semblerait que l'on trouve deux grandes catégories de programme VB 5. Ceux (comme TARIFCOM) ressemblant vraiment à des programmes VB 3 ou VB 4, composés d'instructions VB codées et interprétées par la DLL Runtime. Et certains comme ici DGPlayer (mais aussi BlackWidow pour ceux qui connaissent) dont le code est réellement une suite d'instruction ASM (avec appels à des API VB issue de la DLL) et donc tout à fait traçable de manière traditionnelle. Cette différence est particulièrement caractéristique, comme si on avait avec VB 5 la possibilité soit de compiler, soit d'interpreter le code de sortie. Si quelqu'un possèdant l'info pouvait me la transmettre, comme toujours je suis preneur...

Bref ici, on a un programme à priori plus abordable que TARIFCOM...
Première chose à effectuer (si ce n'est pas déjà fait), intégrer MSVBVM50.DLL (la fameuse DLL...) dans les Exports de Softice. Je pense que vous savez comment faire.
Ici, on va utiliser une API issue de cette DLL affichant une boite de dialogue : rtcMsgBox (et oui ça ressemble étrangement à MessageBox :).
Donc BPX rtcMsgBox (si softice vous jette, c'est que vous n'avez pas exporté la bonne DLL).

Et on lance le programme...

Boite de dialogue avec la saisie Name + Serial. Entrez n'importe koi...Bouton "I Accept" et break.
Un coup de F12 pour sortir de la DLL.
Et on se retrouve dans DGPlayer95.exe. Et là tout devient simple, on regarde pourquoi on est arrivé là.
- > un joli saut conditionnel testant SI. Attention il faut remonter assez haut dans le code
(Fidèle à ma sale habitude, je ne vous donnerais pas d'adresse :).
Et il faut remonter juste encore un peu pour voir ce qui a conditionné le positionnement de ESI.
On a un CALL XXXX avec juste en dessous MOV ESI,EAX.
Ce CALL XXXX est en fait l'appel à la routine de Verif...

Voilà en gros ce qu'elle fait :

Elle fait la somme des codes HEXA des lettres du Name entré.
Elle multiplie cette somme par elle-même et divise le tout par le nombre de lettre du Name entré.
Elle ajoute ensuite à ce resultat la somme des codes Hexa des lettres de DGPlayer95.
Nous avons notre Magic Number :)

Ensuite elle va vérifier si "DGP95" est bien au début du Serial entré.
Et vérifier, pour finir, si le nombre terminant le serial est bien notre magic number (en decimal).
(à noter que la comparaison s'effectue en utilisant la FPU...N'oubliez pas : WF sous Softice)

On a donc un SERIAL du genre : DGP95XYYYYY

Avec X n'importe koi (sauf un chiffre) (il sert seulement à séparer DGP95 du magic number, on peut donc y foutre un "-" mais le programme ne le reclame pas explicitement).

Anonymous = 0x3C9 (somme hexa des lettres - 41 6e 6f 6e 79 6d 6f 75 73)
0x3c9 x 0x3c9 = 0xE53D1
0x3c9 / 9 = 0x19789

DGPlayer95 = 0x366 (somme de 44 47 50 6c 61 79 65 72 39 35)

0x19789 + 0x366 = 0x19AEF = 105199

d'où :

Name : Anonymous
Serial : DGP95-105199

Voilà...

A+

Mister X

A propos d'ACDSee...

Je pense qu'on a du, grosso-modo, utiliser la même méthode, Ethan et moi...
De mon précédent message, je donnais juste le principe car en effet, tester toutes les possibilités entre 1 et FFFFFFFF demande beaucoup trop de temps...

C'est pour ça que j'ai fractionné la recherche en fixant deux valeurs, haute et basse.

Et quand on voit la valeur retournée pour "Ethan" par le Generator de Serial, on voit que j'étais pas près de la sortir :)
Pour "Anonymous", le code a été retourné rapidement (idem pour ethan avec un espace devant...).

Bref voici le code rajouté :
(pour la version 2.22b2, je ne sais pas ce qu'il en est pour la nouvelle version 2.23, elle accepte de toutes façons les mêmes SERIAL donc y'aura juste à modifier les offsets je penses).

J'ai placé le début de code à l'endroit de l'ancien appel de verif soit en 405C6B :

-- première partie : transformation code (valeur hexa) en chaine à l'aide de la fonction WSPRINTFA

```
:00405C6B BBFFFFFFFF          mov ebx, FFFFFFFF                    <-- valeur de départ à tester
(valeur haute, on décremente)
:00405C70 BFF0FFFFFF          mov edi, FFFFFFF0                    <-- valeur d'arrivée (valeur
basse)

* Referenced by a (U)nconditional or (C)onditional Jump at Address:
|:00405CA2(C)
|
:00405C75 8D44247C            lea eax, dword ptr [esp+7C]          <-- on récupère l'offset de la chaine
SERIAL
:00405C79 53                  push ebx                             <-- on empile ebx (donc la valeur du code
à tester)

* Possible StringData Ref from Data Obj ->"%u"
                        |
:00405C7A 6898F44900          push 0049F498                        <-- on empile l'offset de la chaine
"%u"
:00405C7F 50                  push eax                             <-- on empile l'offset du buffer de sortie

* Reference To: USER32.wsprintfA, Ord:0264h
                        |
:00405C80 FF15FCFB4A00        Call dword ptr [004AFBFC]            <-- appel de WSPRINTFA
:00405C86 83C40C              add esp, 0000000C                    <-- on ajuste la pile
```

quelques commentaires : wpsrintf prend dans la forme utilisée ici trois arguments, tous passés par la pile :

- le chiffre à transformer (empilé en premier, correspondant au dernier argument donc)
- la chaine de "formatage" (ici "%u" signifiant que l'on formate le nombre HEXA en entier non signé)
- le buffer de sortie (récupérant le nombre transformer en chaine)

A noter que la fonction WSPRINTFA est déjà utilisée par le programme ACDSee, donc son utilisation en est grandement simplifiée, car elle figure dans la table des fonctions importées. On a donc juste à repérer quel CALL appel WSPRINTFA à l'aide de WDASM.
(en l'occurence : Call dword ptr [004AFBFC])

Second problème, il nous faut une chaine "%u" indispensable pour WSPRINTF. On va donc utiliser la chaine indiquant "Name and Serial don't match" ou je sais plus quoi...
On doit donc répérer l'offset de cette chaine (facile avec WDASM encore une fois) (soit 49F498). Et puis remplacer avec un éditeur de texte le début de "Name and Serial etc..." par "%u" sans oublier de rajouter un 00 de fin de chaine juste après.

Nous allons ensuite utiliser le reste de la chaine pour y glisser un "CODE NOT FOUND" ou ce que vous voulez en frenchy (j'ai pris l'habitude de toujours faire les textes en english donc vous étonnez pas pour le CODE NOT FOUND ;)...
La chaine "CODE NOT FOUND" va donc commencer en 49F49B (49F498 + 3 -> %u00). N'oubliez pas d'y rajouter un 00 à la fin aussi...
Pour résumer en 49F498, on a : % u 00 C O D E N O T F O U N D 00

On continue :

-- seconde partie : appel à la routine de vérif de ACDSee

```
:00405C89 8D44247C          lea eax, dword ptr [esp+7C]
:00405C8D 8D4C243C          lea ecx, dword ptr [esp+3C]
:00405C91 50                push eax                    <-- on empile l'offset SERIAL
:00405C92 51                push ecx                    <-- on empile l'offset NAME
:00405C93 E828F8FFFF        call 004054C0                   <-- appel à la routine de VERIF
de ACDSee
:00405C98 83C408            add esp, 00000008           <-- réajustement de la pile
:00405C9B 85C0              test eax, eax               <-- on teste si la vérif est ok
:00405C9D 750C              jne 00405CAB                    <-- si oui saut
:00405C9F 4B                dec ebx                     <-- si non, on décremente ebx
:00405CA0 3BFB              cmp edi, ebx                <-- on vérifie si on a atteint la valeur
d'arrivée
:00405CA2 7CD1              jl 00405C75                 <-- sinon on boucle


* Possible StringData Ref from Data Obj ->"No Code found"
                    |
:00405CA4 B99BF44900        mov ecx, 0049F49B               <-- si on est ici, c'est que le code
n'a pas été trouvé
:00405CA9 EB41              jmp 00405CEC                    <-- donc on fixe ecx avec l'offset
de "Code Not Found"
                                                            <-- + saut vers l'affichage

* Referenced by a (U)nconditional or (C)onditional Jump at Address:
|:00405C9D(C)
|
:00405CAB 8D4C247C          lea ecx, dword ptr [esp+7C]     <-- si on est ici, c'est que le code
a été trouvé, donc
```

```
:00405CAF EB3B              jmp 00405CEC                          <-- on fixe ecx avec l'offset de la
chaine serial
                                                                  <-- et on saute vers l'affichage
-- portion de code non utilisé
...
* Referenced by a (U)nconditional or (C)onditional Jump at Addresses:
|:00405CA9(U), :00405CAF(U)
|

* Possible StringData Ref from Data Obj ->"ACDSee 32"
                    |
:00405CEC 8B1524F24900      mov edx, dword ptr [0049F224]         <-- on se retrouve ici avec la
routine d'affichage du
:00405CF2 6A00               push 00000000                        <-- MessageBox de ACDSee
:00405CF4 52                push edx
:00405CF5 51                push ecx                          <-- seul modif on met un push ecx (qui
contient ce qu'on doit                                                          <-- afficher,
serial ou message "Code not Found")
:00405CF6 90                nop                               <-- et on nop les octets superflux
:00405CF7 90                nop
:00405CF8 90                nop
:00405CF9 90                nop
:00405CFA 56                 push esi

* Reference To: USER32.MessageBoxA, Ord:0195h
                    |
:00405CFB FF1538FA4A00      Call dword ptr [004AFA38]
```

Voilà, j'espère que vous avez tout compris (je sais, c'est pas trop clair). Sorry pour mon code ASM non optimisé
De toutes façons, mieux vaut étudier le Generator de Serial trouvé par Ed Hayes.
Je rends d'ailleurs hommage à son programmeur, mon truc à côté c'est de la merde...

A+
Mister X

Cours de Crack (#1) pour « Visual Basic 3.0 » par DrJulo

le 13/01/98.

Introduction :

Non ! Ne vous découragez pas, on peut cracker du Visual Basic (la version 3 pour le moment !). En effet, Visual Basic est un langage qui compile assez mal les programmes (ils sont lents) mais qui ne se désassemble pas. En fait peu de personnes savent et essayent de cracker des logiciels développés avec ce langage.

Pourtant ce n'est pas bien compliqué. Il vous suffi de posséder un éditeur hexadécimale et d'avoir sous les yeux le tableau des codes Visual Basic 3.

Nous allons commencer simplement en crackant des mots de passe. CompKino, logiciel de capture d'écran (sous forme de vidéo), fera l'affaire.

Compilation Visual Basic 3:

Les programmes Visual Basic n'utilisent donc pas les mêmes codes que les autres langages 32bits tel le C++.

Ce qui va faciliter la tache est que ces programmes font appel à de nombreuses routines internes (il n'y a même que ça !) ainsi qu'à de nombreuses boites de dialogue. Par chance, le texte de ces messages apparaît dans la traduction du code hexadécimale. De plus les propriétés de chaque objet (boutons, boites de texte, ...), et les routines se trouvent bien séparés par rapport au programme.

Voici répertorié dans ce tableau quelque traduction du code Visual Basic :

Teste entre un objet « Texte » et une variable string :	c31168	Si =
	c3117a	Si <>
Teste entre deux variables de type « string » :	1a00f6	Si =
	1a00ff	Si <>
Début d'une routine :	4b49	Juste après 0e5b0e
Fin d'une routine :	0e5b0e	
Objets Texte et Bouton (les propriétés concernant les boutons se trouvent juste après la chaîne 7701 qui se trouve elle même après le nom du bouton).	0900	Invisible et inactif
	0800	Inactif (grisé)
Objets à Cocher (CheckBox)	0800	Inactif
Pour tous les objets visible*	- - - -	Actif et

* pour réactiver une option vous devrez supprimer la ligne 0800 ou 0900. De ce fait, votre fichier fera 2 octets de moins. Vous pourrez les rajouter 0000 dans les chaînes de zéro qui se trouvent partout dans les programmes Visual Basic.

Je vois que ça n'a pas l'air encore clair pour tout le monde ! Voici un petit exemple qui illustre les tests entre Texte et variable.

Crack de CompKino :

Installez-vous confortablement et lancez un première fois CompKino pour voir ce qui se passe dedans !

Bien ! On nous demande un code dans un objet « Texte ». Il y aura sûrement un test entre cet objet et une variable comportant le bon code. Lorsqu'on entre un mauvais code, une barre apparaît vous disant « Invalid activation code ».

Allons voir tout ça dans le listing hexadécimale. Comme je vous l'ai dit, les chaînes de chaînes de caractères (messages) sont visibles dans le listing ASCII. Cherchons le message « Invalid activation code » en respectant la case (les majuscules).

```
000019E0 0000 0000 0000 0000 0000 0000 0000 0000
  ................
000019F0 0000 0000 0000 0000 0000 0000 0000 0000
  ................
00001A00 4B49 E537 B00E A84C 0100 4C00 094C 0BC0
  KI.7...L..L..L..
00001A10 6C2B 3000 C311 6844 B734 7E01 3549 9A38
  l+0...zD.4~.5I.8
00001A20 1000 2400 0A00 4272 6176 6F21 2121 2020   ..$...Bravo!!!
00001A30 0000 C311 ED37 B00E A84C 0100 4C00 094C
  .....7...L..L..L
00001A40 0BC0 B877 144C 5200 3549 ED37 B00E A84C
  ...w.LR.5I.7...L
00001A50 0100 4C00 094C 0BC0 9A38 0600 5E00 0000
  ..L..L...8..^...
00001A60 0000 C311 6844 B734 C000 1F49 B167 ED37
  ....zD.4...I.g.7
00001A70 B00E A84C 0100 4C00 C34F 6737 614B 8800
  ...L..L..Og7aK..
00001A80 0000 584B 1300 0000 FC4F 0100 6A37 1F49
  ..XK.....O..j7.I
00001A90 9A38 1C00 9600 1700 456E 7465 7220 796F   .8......Enter yo
00001AA0 7572 206E 616D 6520 706C 6561 7365 2100   ur name please!.
...
00001BE0 E201 1800 496E 7661 6C69 6420 6163 7469   ....Invalid acti
00001BF0 7661 7469 6F6E 2063 6F64 6521 0000 C311   vation code!....
...
00001C90 7221 0000 C311 144C 5200 3549 EC35 4B49
  r!.....LR.5I.5KI
00001CA0 EC35 4B49 D965 5E0E 5B0E 4B49 502B 4400
  .5KI.e^.[.KIP+D.
```

En remontant, nous cherchons le début de la routine et plus bas la fin pour voir le domaine d'étude. Remarquez les nombreux messages qui font penser à une certaine boite demandant un mot de passe !

C'est donc là-dedans que se trouve le problème. Ce que nous cherchons est plutôt un test entre un objet Texte et une variable.

Nous trouvons deux Codes (C31168). Si on regarde bien l'un est avant « bravo » l'autre après. En réfléchissant un peu on se dit que « bravo » intervient lorsque le mot de passe entré est juste. Donc le test s'effectue avant. C'est donc le premier que l'on remplacera par un c3117a.

Le deuxième teste est pour le deuxième objet Texte qui va tester la présence d'un nom. Si vous n'avez pas inscrit de nom, alors vous tombez sur « Enter your name please ». En remplaçant ce test par c3117a vous ne devrez plus entrer de nom sinon vous retomberiez sur la même phrase.

Relancez le soft, entrez un mot de passe bidon et un nom (si vous n'avez pas changez le deuxième test). Le logiciel est maintenant dans sa version enregistrée.

Vocabulaire :

String : variable ne contenant que du texte. Il est impossible de faire des opérations avec.

Routine : morceau de programme. Les routines servent à classer le programme pour une meilleur compréhension du programmeur.

ASCII : ce sont tous les caractères qui se trouvent sur votre clavier ainsi que beaucoup d'autre encore. Chaque caractère ASCII est codé par un code hexadécimale. Ex : 7A représente le z.
On trouve cette traduction ASCII dans la plupart des éditeurs hexadécimale (à droite).

Conclusion :

J'espère que ce cour ne vous aura pas parût trop flou. En utilisant la même méthode vous serez capable de cracker 80% des softs Visual Basic 3 nécessitant un mot de passe. Si il y a des prochains cours (si vous le demandez !), nous verrons comment réactiver des options, enlever les nagscreens, etc... Vous pouvez déjà commencer à réactiver des boutons à l'aide du tableau.

Il s'avère que les programmes Visual Basic 4 et 5 ne sont pas compilé comme le 3. Le codage est beaucoup plus délicat et compliqué. Vous devrez vous contentez pour le moment de cracker les quelques nouveau softs programmés en Visual Basic 3 !

Dr Julo

Ps : Vous pouvez m'envoyez vos remarques et commentaires sur :
drjulo@hotmail.com