

CrackMe 2 – Kaine

solution par mastermatt29

1.Introduction.....	2
2.Les antixxx.....	3
3.Trouver un serial.....	7
4.Conclusion.....	10

1.Introduction

Salut salut !

Je propose ici une solution pour le crackme 2 de Kaine. Je sais que ce crackme est assez vieux, mais les antixxx sont intéressants et m'ont permis de mieux découvrir le mécanisme du débogage. Aussi j'espère pouvoir faire partager ce que j'ai découvert.

Voilà, voilà si tout n'était pas clair vous pouvez me le signaler (mastermatt29@gmail.com)

Je n'utiliserai dans ce tutorial que Ollydbg (pour debugger, ca peut servir ;-)).

Prêt ? Bon ben c'est parti !

2. Les antixxx

Pour un peu de sémantique. Le terme antixxx désigne les morceaux de code destinés à détecter l'usage de certains programmes (surtout les debugger), voir empêcher leur usage (MessageBox prévenant qu'un debugger tourne jusqu'à un plantage pur et simple de Windows).

Bon chargeons notre programme dans Olly (inutile d'aller voir le PE, ya pas d'infos intéressantes). On arrive ici :

00401000	§ 68 00104000	PUSH def_i_kai.0040100D	
00401005	. C3	RETN	RET used as a jump to 0040100D
00401006	> 6A 00	PUSH 0	ExitCode = 0
00401008	. E8 FF040000	CALL <JMP.&KERNEL32.ExitProcess>	ExitProcess
0040100D	> EB 03	JMP SHORT def_i_kai.00401012	
0040100F	CD	DB CD	
00401010	. 20C7	AND BH,AL	
00401012	> 68 49304000	PUSH def_i_kai.00403048	ASCII "User32"
00401017	.. EB 01	JMP SHORT def_i_kai.0040101A	
00401019	E8	DB E8	
0040101A	> E8 F3040000	CALL <JMP.&KERNEL32.GetModuleHandleA>	GetModuleHandleA
0040101F	.. EB 01	JMP SHORT def_i_kai.00401022	
00401021	E8	DB E8	
00401022	> EB 03	JMP SHORT def_i_kai.00401027	
00401024	CD	DB CD	
00401025	. 20C7	AND BH,AL	(pas executer)
00401027	> 68 58304000	PUSH def_i_kai.00403058	ASCII "MessageBoxA"
0040102C	.. EB 03	JMP SHORT def_i_kai.00401031	
0040102E	CD	DB CD	
0040102F	. 20C7	AND BH,AL	(pas executer)
00401031	> 50	PUSH EAX	
00401032	.. EB 01	JMP SHORT def_i_kai.00401035	
00401034	E8	DB E8	
00401035	> EB 03	JMP SHORT def_i_kai.0040103A	
00401037	CD	DB CD	
00401038	. 20C7	AND BH,AL	(pas executer)
0040103A	> E8 D9040000	CALL <JMP.&KERNEL32.GetProcAddress>	GetProcAddress
0040103F	.. EB 03	JMP SHORT def_i_kai.00401044	
00401041	CD	DB CD	
00401042	. 20C7	AND BH,AL	
00401044	> EB 01	JMP SHORT def_i_kai.00401047	
00401046	E8	DB E8	
00401047	> 8038 CC	CMP BYTE PTR DS:[EAX],0CC	<- Regarde si le premier octet de
0040104A	.. EB 01	JMP SHORT def_i_kai.0040104D	MessageBoxA est un breakpoint
0040104C	E8	DB E8	
0040104D	> 0F84 AC040000	JE def_i_kai.004014FF	<- Si oui => BadBoy
00401053	.. EB 03	JMP SHORT def_i_kai.00401058	

Juste un petit truc à remarquer en 401000 : on PUSH une adresse et on fait ensuite un RET. Cette technique n'est en fait qu'un JMP dissimulé pour aller en 40100D (Olly nous le signale d'ailleurs et ça n'a guère d'intérêt si ce n'est que « ça fait + l33t ;) » dicit meat).

Il faut savoir qu'un RET prend la première adresse de la pile et y saute. Lors d'un CALL par exemple l'adresse courante (située dans le registre EIP) du programme est tout d'abord empilée puis le programme passe à l'adresse spécifiée par le CALL.

L'ExitProcess qui suit n'est là que pour parer au cas impossible où le code continuerait de s'exécuter.

Nous ne ferons pas cas des JMP qui saute 1 ou 2 octets plus loin, qui polluent le code et qui ne servent à rien si ce n'est à nous embêter.

En 40103A, nous avons un appel à l'API GetProcAddress pour récupérer l'adresse de MessageBoxA. On va ensuite comparer l'octet situé à cette adresse et vérifier que ce n'est pas CC.

Note sur la théorie des debuggers : vous êtes vous déjà demandé comment faisait Olly pour stopper un programme pile-poil là où nous avons mis un breakpoint ? En fait Olly place à l'adresse de l'instruction où il faut breaker l'instruction INT3 (qui est codée par l'octet CC). Lorsque cette instruction est rencontrée, la main est passée au debugger, charge à lui ensuite de faire ce qu'il veut.

Vous voyez où je veut en venir ? Et oui, le programme vérifie si l'on n'a pas posé de breakpoint à l'entrée de l'API MessageBoxA !

Voilà voilà on l'a pas fait, donc on peut continuer ! On trouve un CALL intéressant en 40106A. Allons donc voir le code de ce CALL :

```
004014F2 | $ 8A11 | MOV DL, BYTE PTR DS:[EAX] | <- Met un octet dans DL
004014F4 | . 3010 | XOR BYTE PTR DS:[EAX], DL | <- XOR l'octet pointer par EAX avec DL
004014F6 | . 3BC3 | CMP EAX, EBX | <- Regarde si on est arriver a la fin de la zone
004014F8 | .v74 04 | JE SHORT def_i_kai.004014FE
004014FA | . 40 | INC EAX
004014FB | . 41 | INC ECX
004014FC | .^EB F4 | JMP SHORT <def_i_kai.Cryptage>
004014FE | > C3 | RETN
```

Donc voilà rien de très compliqué on va coder par un XOR tous les octets pointés par EAX jusqu'à arriver à l'endroit pointé par EBX.

Lors de son appel en 40106A les adresses respectives de EAX et EBX sont 40100D et 401053. Tiens donc, le code qu'on vient tout juste d'exécuter ! Le crackme n'essaierait-il pas de brouiller les pistes ?

Bon continuons. On trouve le même trick sur l'API CreateFileA que celui qu'il y avait sur MessageBoxA (à savoir une détection de breakpoint sur le premier octet de la fonction). De nouveau en 4010C5 on trouve un cryptage du code tout juste exécuté.

Les choses intéressantes reprennent en 4010DB :

004010DB	> 6A 00	PUSH 0	<pre> hTemplateFile = NULL Attributes = NORMAL Mode = OPEN_EXISTING pSecurity = NULL ShareMode = FILE_SHARE_READ FILE_SHARE_WRITE Access = GENERIC_READ GENERIC_WRITE FileName = "\\.\NTALL" CreateFileA </pre>	
004010DD	. 68 80000000	PUSH 80		
004010E2	. 6A 03	PUSH 3		
004010E4	. 6A 00	PUSH 0		
004010E6	. 6A 03	PUSH 3		
004010E8	. 68 000000C0	PUSH C0000000		
004010ED	. 50	PUSH EAX		
004010EE	. E8 13040000	CALL <JMP.&KERNEL32.CreateFileA>		
004010F3	√EB 01	JMP SHORT def_i_kai.004010F6		
004010F5	. E8	DB E8		
004010F6	> EB 03	JMP SHORT def_i_kai.004010FB	<pre> <- Test si l'ouverture a echouer </pre>	
004010F8	. CD	DB CD		
004010F9	. 20C7	AND BH,AL		
004010FB	> 83F8 FF	CMP EAX,-1		
004010FE	√EB 03	JMP SHORT def_i_kai.00401103		
00401100	. CD	DB CD		
00401101	. 20C7	AND BH,AL		
00401103	> 0F85 F6030000	JNZ def_i_kai.004014FF		
00401109	√EB 01	JMP SHORT def_i_kai.0040110C		<pre> <- Si l'ouverture a reussi alors BadBoy </pre>

On cherche ici à ouvrir le driver NTALL (driver de SoftIce) grâce à l'API CreateFileA. Celle-ci retourne un handle (un identificateur en gros) si l'ouverture à réussi et -1 sinon. Le programme cherche donc le driver NTALL et si celui-ci est chargé alors on saute vers BadBoy.

Cette technique est connue sous le nom de MeltIce. Elle a été utilisée à l'origine par le SymbolLoader de SoftIce pour détecter si celui était lancé.

On retrouve cette technique successivement en 401137 (FROGSICE pour FrogIce qui cache la présence de SoftIce), puis en 401190 (SICE, driver de SoftIce sous 98) et en 4011E9 (NTICE, driver de SoftIce sous NT).

Le prochain trick est en 401239 :

00401239	. 68 87124000	PUSH def_i_kai.00401287	<pre> <- Adresse de la fonction gerant les exceptions </pre>	
0040123E	√EB 01	JMP SHORT def_i_kai.00401241		
00401240	. E8	DB E8		
00401241	> E8 D0020000	CALL <JMP.&KERNEL32.SetUnhandledExceptionFilter>		
00401246	√EB 03	JMP SHORT def_i_kai.0040124B		
00401248	. CD	DB CD		
00401249	. 20C7	AND BH,AL		
0040124B	> EB 01	JMP SHORT def_i_kai.0040124E		
0040124D	. E8	DB E8		
0040124E	> B8 04000000	MOV EAX,4		
00401253	√EB 01	JMP SHORT def_i_kai.00401256	<pre> SetUnhandledExceptionFilter </pre>	
00401255	. E8	DB E8		
00401256	> BD 4B484342	MOV EBP,4243484B		
0040125B	√EB 03	JMP SHORT def_i_kai.00401260		
0040125D	. CD	DB CD		
0040125E	. 20C7	AND BH,AL		
00401260	> CC	INT3		
00401261	√EB 01	JMP SHORT def_i_kai.00401264		
				<pre> <- Interruption </pre>

Note sur les exceptions : il faut savoir que Windows possède un système de gestion des exceptions. C'est à dire que lorsqu'un programme fait un opération non conforme (division par 0, écriture sur un emplacement interdit...), Windows essaie de corriger cette erreur en faisant appel à des parties du programme fautif qui y sont destinées : les SEH (Structured Exception Handler). Nous n'entrerons pas dans les détails, mais sachez que tout programme possède au moins le SEH de kernel32 chargé d'apporter un traitement par défaut à toute exception.

Lors d'une exception, ce SEH appelle entre autre l'API

UnhandledExceptionFilter qui fournit une fonction de gestion d'exception qui a été précédemment spécifiée par un appel de SetUnhandledExceptionFilter. Dans notre cas on trouve l'appel à SetUnhandledExceptionFilter en 401241 avec en paramètre 401287 (l'adresse de la fonction de gestion, qui bizarrement n'est pas trop loin du code actuel).

Ensuite on continue jusqu'en 401260 où le programme cherche à exécuter un INT3. Cette instruction est elle aussi une exception, ce qui signifie que normalement on devrait passer en 401287 grâce au mécanisme que nous avons expliqué précédemment. Seulement voilà lorsque le programme est debuggé l'exception est directement passée au debugger, sans passer par la fonction spécifiée par SetUnhandledExceptionFilter ! Or ici ne pas passer par 401287 revient à aller directement vers BadBoy...

Donc on va assembler le code et rentrer un JMP 401287 :

```
00401239 . 68 87124000 PUSH def_i_kai.00401287
0040123E .>EB 01 JMP SHORT def_i_kai.00401241
00401240 E8 DB E8
00401241 > E8 08020000 CALL <JMP.&KERNEL32.SetUnhandledExceptionFilter>
00401246 .>EB 03 JMP SHORT def_i_kai.0040124B
00401248 CD DB CD
00401249 . 20C7 AND BH,AL
0040124B >EB 01 JMP SHORT def_i_kai.0040124E
0040124D E8 DB E8
0040124E > B8 04000000 MOV EAX,4
00401253 .>EB 01 JMP SHORT def_i_kai.00401256
00401255 E8 DB E8
00401256 > BD 4B484342 MOV EBP,4243484B
0040125B .>EB 03 JMP SHORT def_i_kai.00401260
0040125E CD DB CD
0040125E . 20C7 AND BH,AL
00401260 >EB 25 JMP SHORT def_i_kai.00401287
00401262 90 NOP
00401263 E8 DB E8
00401264 > 68 FF144000 PUSH def_i_kai.004014FF
00401269 . C3 RETN
```

<- Adresse de la fonction gerant les exceptions

Assemble at 00401260

JMP 401287

Fill with NOP's

Assemble Cancel

<- Interruption

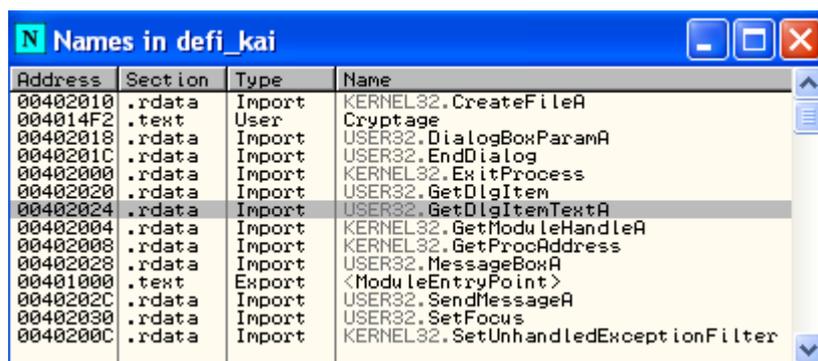
RET used as a jump to 004014FF

Voilà, étant donné qu'on a maintenant passé tous les antixxx on va pouvoir s'intéresser au serial !

3.Trouver un serial

Notre point de départ sera l'API GetDlgItemTextA qui comme chacun le sait sert à récupérer des choses comme des serial...

Donc on fait un click-droit puis Search For -> Name in current module. On a une nouvelle fenêtre qui s'affiche avec les noms des API utilisées dans le programme :



Address	Section	Type	Name
00402010	.rdata	Import	KERNEL32.CreateFileA
004014F2	.text	User	Cryptage
00402018	.rdata	Import	USER32.DialogBoxParamA
0040201C	.rdata	Import	USER32.EndDialog
00402000	.rdata	Import	KERNEL32.ExitProcess
00402020	.rdata	Import	USER32.GetDlgItem
00402024	.rdata	Import	USER32.GetDlgItemTextA
00402004	.rdata	Import	KERNEL32.GetModuleHandleA
00402008	.rdata	Import	KERNEL32.GetProcAddress
00402028	.rdata	Import	USER32.MessageBoxA
00401000	.text	Export	<ModuleEntryPoint>
0040202C	.rdata	Import	USER32.SendMessageA
00402030	.rdata	Import	USER32.SetFocus
0040200C	.rdata	Import	KERNEL32.SetUnhandledExceptionFilter

A nouveau click-droit sur GetDlgItemTextA puis View call tree. Parfait ! Nous n'avons qu'un seul appel à cette fonction et c'est en 401413. Allez zou on y va :

```
004013C8 . 68 48304000 PUSH defi_kai.00403048
004013CD . E8 40010000 CALL <JMP.&KERNEL32.GetModuleHandleA>
004013D2 . 68 70304000 PUSH defi_kai.00403070
004013D7 . 50 PUSH EAX
004013D8 . E8 3B010000 CALL <JMP.&KERNEL32.GetProcAddress>
004013DD . 8038 CC CMP BYTE PTR DS:[EAX],0CC
004013E0 .v0F84 19010000 JE defi_kai.004014FF
004013E6 . 68 48304000 PUSH defi_kai.00403048
004013EB . E8 22010000 CALL <JMP.&KERNEL32.GetModuleHandleA>
004013F0 . 68 80304000 PUSH defi_kai.00403080
004013F5 . 50 PUSH EAX
004013F6 . E8 1D010000 CALL <JMP.&KERNEL32.GetProcAddress>
004013FB . 8038 CC CMP BYTE PTR DS:[EAX],0CC
004013FE .v0F84 FB000000 JE defi_kai.004014FF
00401404 . 68 00020000 PUSH 200
00401409 . 68 BD304000 PUSH defi_kai.004030BD
0040140E . 6A 66 PUSH 66
00401410 . FF75 08 PUSH DWORD PTR SS:[EBP+8]
00401413 . E8 1E010000 CALL <JMP.&USER32.GetDlgItemTextA>
00401418 . BE BD304000 MOV ESI,defi_kai.004030BD
0040141D . E8 6D000000 CALL defi_kai.0040148F
00401422 .vEB 10 JMP SHORT defi_kai.00401434
00401424 > 66:83F8 6B CMP AX,6B
00401428 .v75 0A JNZ SHORT defi_kai.00401434
0040142A . 6A 00 PUSH 0
0040142C . FF75 08 PUSH DWORD PTR SS:[EBP+8]
0040142F . E8 F6000000 CALL <JMP.&USER32.EndDialog>
```

```
pModule = "User32"
GetModuleHandleA
ProcNameOrOrdinal = "GetDlgItemTextA"
hModule
GetProcAddress

pModule = "User32"
GetModuleHandleA
ProcNameOrOrdinal = "GetWindowTextA"
hModule
GetProcAddress

Count = 200 (512.)
Buffer = defi_kai.004030BD
ControlID = 66 (102.)
hWnd
GetDlgItemTextA

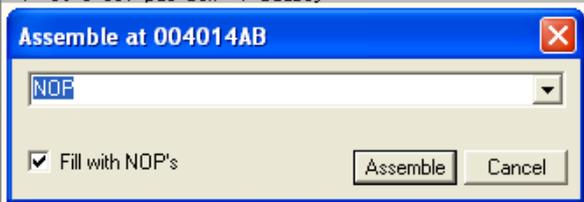
Result = 0
hWnd
EndDialog
```

Ici encore on va vérifier en 4013FB que l'on n'a pas mis de breakpoint sur le début de la fonction (sinon BadBoy). On va maintenant aller voir ce que contient le CALL de 40141D :

0040148F	50	PUSH EAX	
00401490	51	PUSH ECX	
00401491	52	PUSH EDX	
00401492	33C0	XOR EAX,EAX	<- EAX = 0
00401494	33C9	XOR ECX,ECX	<- ECX = 0
00401496	8A06	MOV AL,BYTE PTR DS:[ESI]	<- Met un caractere dans AL
00401498	8A4E 01	MOV CL,BYTE PTR DS:[ESI+1]	<- Le caractere suivant dans CL
0040149B	85C9	TEST ECX,ECX	
0040149D	74 05	JE SHORT def_i_kai.004014A4	<- Verifie qu'on a pas atteint la fin de la chaine
0040149F	F7E1	MUL ECX	<- Multiplie ECX par EAX. Le resultat est dans EAX
004014A1	46	INC ESI	
004014A2	EB F4	JMP SHORT def_i_kai.00401498	<- On boucle
004014A4	5A	POP EDI	
004014A5	59	POP EDX	
004014A6	3D 80C08958	CMP EAX,5889C080	<- On compare EAX a une constante
004014A8	75 16	JNZ SHORT def_i_kai.004014C3	<- Si c'est pas bon => BadBoy
004014AD	B8 9F144000	MOV EAX,def_i_kai.0040149F	
004014B2	BB BC144000	MOV EBX,def_i_kai.004014BC	
004014B7	BF A6144000	MOV EDI,def_i_kai.004014A6	
004014BC	E8 09000000	CALL def_i_kai.004014CA	<- Suite de la verification
004014C1	58	POP EAX	
004014C2	C3	RETN	

Rien de très compliqué dans cette boucle, il suffit juste que le produit de tous les caractères du serial soit égal à 5889C080, ou plus exactement que le produit de tous les caractères du serial congrue à 5889C080 modulo 2^{32} (c'est à dire qu'en fait il peut y avoir une sorte d'overflow sur EAX qui fait que tous les nombres du produit ne sont pas présent). Pour l'instant on va juste modifier le JNZ 4014C3 par des NOP pour pouvoir continuer à analyser le code tout en debuggant (ce qui va être nécessaire vous allez voir) :

004014A5	59	POP ECX	
004014A6	3D 80C08958	CMP EAX,5889C080	<- On compare EAX a une constante
004014AB	90	NOP	<- Si c'est pas bon => BadBoy
004014AC	90	NOP	
004014AD	B8 9F144000	MOV EAX,def_i_kai.0040149F	
004014B2	BB BC144000	MOV EBX,def_i_kai.004014BC	
004014B7	BF A6144000	MOV EDI,def_i_kai.004014A6	
004014BC	E8 09000000	CALL def_i_kai.004014CA	
004014C1	58	POP EAX	
004014C2	C3	RETN	
004014C3	58	POP EAX	
004014C4	68 65144000	PUSH def_i_kai.00401465	
004014C9	C3	RETN	
004014CA	66:C700 03C1	MOV WORD PTR DS:[EAX],0C103	
004014CF	C703 90909090	MOV DWORD PTR DS:[EBX],90909090	
004014D5	83C3 04	ADD EBX,4	
004014D8	C603 90	MOV BYTE PTR DS:[EBX],90	
004014DB	47	INC EDI	
004014DC	C707 A0030000	MOV DWORD PTR DS:[EDI],3A0	
004014E2	BE BD304000	MOV ESI,def_i_kai.004030BD	
004014E7	E8 A3FFFFFF	CALL def_i_kai.0040148F	
004014EC	68 48144000	PUSH def_i_kai.00401448	
004014F1	C3	RETN	



On continue et on rentre dans le CALL de 4014BC :

004014AD	B8 9F144000	MOV EAX,def_i_kai.0040149F	
004014B2	BB BC144000	MOV EBX,def_i_kai.004014BC	
004014B7	BF A6144000	MOV EDI,def_i_kai.004014A6	
004014BC	E8 09000000	CALL def_i_kai.004014CA	<- Suite de la verification
004014C1	58	POP EAX	
004014C2	C3	RETN	
004014C3	58	POP EAX	
004014C4	68 65144000	PUSH def_i_kai.00401465	
004014C9	C3	RETN	
004014CA	66:C700 03C1	MOV WORD PTR DS:[EAX],0C103	RET used as a jump to 00401465
004014CF	C703 90909090	MOV DWORD PTR DS:[EBX],90909090	<- Remplace l'instruction MUL ECX
004014D5	83C3 04	ADD EBX,4	<- Place des NOP en 4014BC
004014D8	C603 90	MOV BYTE PTR DS:[EBX],90	
004014DB	47	INC EDI	
004014DC	C707 A0030000	MOV DWORD PTR DS:[EDI],3A0	<- Remplace la constante dans la comparaison
004014E2	BE BD304000	MOV ESI,def_i_kai.004030BD	ASCII "123456"
004014E7	E8 A3FFFFFF	CALL def_i_kai.0040148F	<- Et zou on appel l'endroit que l'on a patcher
004014EC	68 48144000	PUSH def_i_kai.00401448	
004014F1	C3	RETN	RET used as a jump to 00401448

On s'aperçoit que les MOV avant le CALL ne sont pas innocents... En effet le programme va remplacer des morceaux de son code de comparaison de serial ! (c'est un « self-patcher process » © eedy31). Une fois que tout aura été remplacé on va executer ce « nouveau » code qui est devenu :

0040148F	50	PUSH EAX	defi_kai.0040149F
00401490	51	PUSH ECX	
00401491	52	PUSH EDX	
00401492	33C0	XOR EAX,EAX	<- EAX = 0
00401494	33C9	XOR ECX,ECX	<- ECX = 0
00401496	8A06	MOV AL,BYTE PTR DS:[ESI]	<- Met un caractere dans AL
00401498	8A4E 01	MOV CL,BYTE PTR DS:[ESI+1]	<- Le caractere suivant dans CL
0040149B	85C9	TEST ECX,ECX	
0040149D	74 05	JE SHORT defi_kai.004014A4	<- Verifie qu'on a pas atteint la fin de la chaine
0040149F	83C1	ADD EAX,ECX	<- Somme EAX et ECX
004014A1	46	INC ESI	
004014A2	EB F4	JMP SHORT defi_kai.00401498	<- On boucle
004014A4	5A	POP EDX	
004014A5	59	POP ECX	
004014A6	3D A0030000	CMF EAX,3A0	<- On compare EAX a une constante
004014A8	90	NOP	<- Notre JNZ transformer en NOP
004014AC	90	NOP	
004014AD	B8 9F144000	MOV EAX,defi_kai.0040149F	
004014B2	BB EC144000	MOV EBX,defi_kai.004014BC	
004014B7	BF A6144000	MOV EDI,defi_kai.004014A6	
004014BC	90	NOP	
004014BD	90	NOP	
004014BE	90	NOP	
004014BF	90	NOP	
004014C0	90	NOP	
004014C1	58	POP EAX	
004014C2	C3	RETN	

N'oublions pas que nous avons modifié un JNZ en NOP. Ici la routine prend 2 caractères, un dans AL et le suivant dans CL. Ensuite on fait la somme de ECX et EAX et on boucle. A la fin on doit trouver 3A0 dans EAX.

Bon nous avons toutes les indications nécessaires à la réalisation de notre bruteforcer !

Ce qu'on va faire : on prend un chaine de caractères (chacun compris entre 20 et FE) sur lesquelles on va vérifier les conditions nécessaires. Si ca ne convient pas, on augmente le premier caractère de 1 et on recommence ! Et tout cela jusqu'à ce que l'on trouve un serial qui marche. On prendra un serial d'au moins 3 caractères pour commencer et on prendra 8 comme longueur maximale.

Avec toutes ces informations j'ai réalisé un bruteforcer en C (initialement ca devait être en ASM, mais mes connaissances ne sont pas encore assez développées) dont la source (commentée) est disponible dans le zip.

Je suis désolé, mais je n'ai pas de serial valide à vous proposer et je ne suis même pas sur que mon bruteforcer marche correctement étant donné que je l'ai fait tourner un petit peu, mais malheureusement sans résultat. Si jamais quelqu'un l'aurait testé, qu'il me fasse signe ! (qu'il marche ou non). Cependant je pense que le principe reste bon.

4. Conclusion

Je n'ai pas grand chose à conclure, à part que je remercie Kaine pour son crackme initiant aux antixxx.

Je tiens aussi à remercier pour leur aide :

- - eedy31
- - Kharneth
- - meat

Puis vu la date, ***Bonne année à tous !***

31/12/05
mastermatt29^NAS