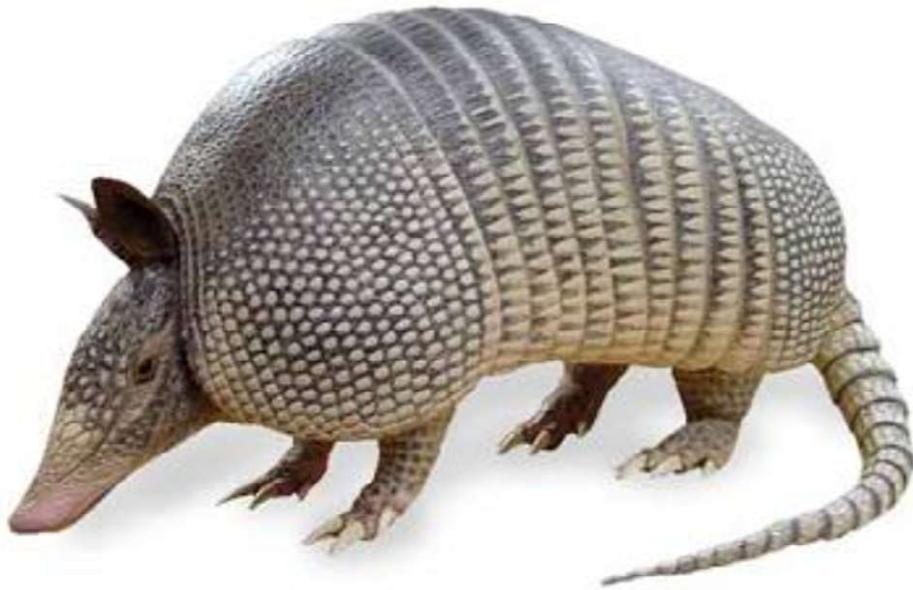


Armadillo 9 x64

Manual Unpacking



Cible: EZ CD Audio Converter 3.1.0 x64 (www.poikosoft.com)

Protection: Armadillo x64 (Standard protection)

Tools requis:

- x64 dbg (<http://x64dbg.com/>)

Pièces jointes:

- *armaccess64.dll* - *La fausse armaccess64.dll à placer dans le dossier du programme pour votre dumped.*
- *ezcd_dump_SCY.exe* - *Le dumped avec sa table d'import reconstruite.*
- *ezcd.xml* - *Le fichier Scylla final avec les modifications faites*

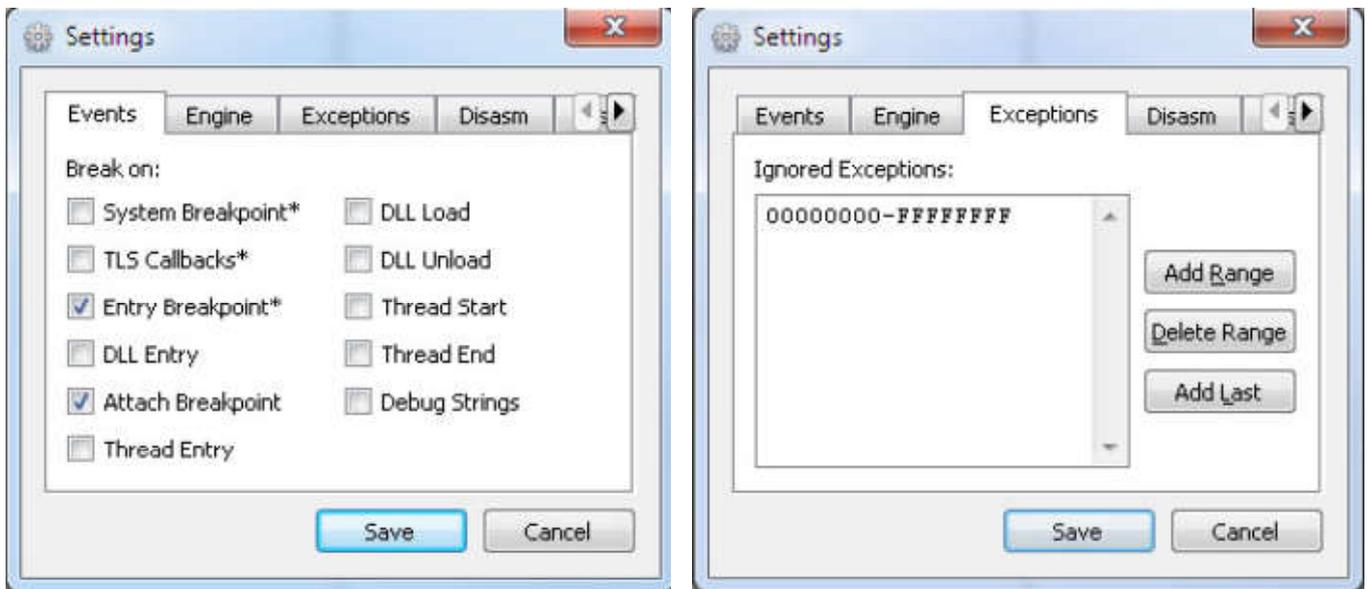
Bonjour,

je vais vous montrer comment unpacker de l'armadillo x64. Pour ceux qui savent le faire en x86, vous verrez qu'il n'y a que très peu de différences. C'est exactement le même processus. A quelques exceptions près puisque en x86, vous risquez de tomber sur des Secured Sections, du Code Splicing, de l'Import Elimination, du Debug Blocker, du CopyMem II ou pire, des Nanomites.

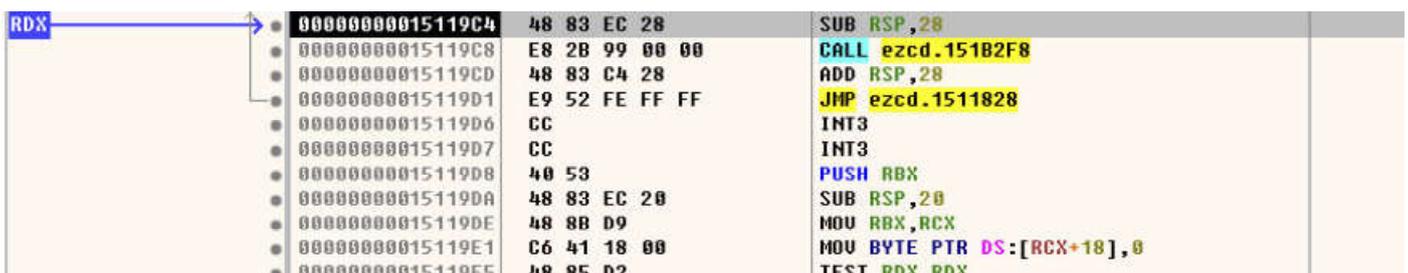
En x64 vous ne trouverez jamais ces protections et vous serez toujours confronté à de la protection Standard. La seule qui est active ici, ce sont les Secured Sections. Des portions de code qui sont manquantes si l'unpack est fait alors que le programme est encore en période d'essais. Pour avoir le code absent, il faut avoir une licence valide du programme cible.

Je vais vous montrer comment unpacker armadillo x64 mais pas à cracker EZCDDA. Une fois votre dumped fait, il y a encore du travail sur l'exécutable et ce n'est pas le sujet ici. Je fournis une armaccess64.dll mais pas le code source. Je n'ai pas cherché mais il doit bien en exister sur le net. Ou bien codez là vous même (oué je sais, je ne suis pas sympa).

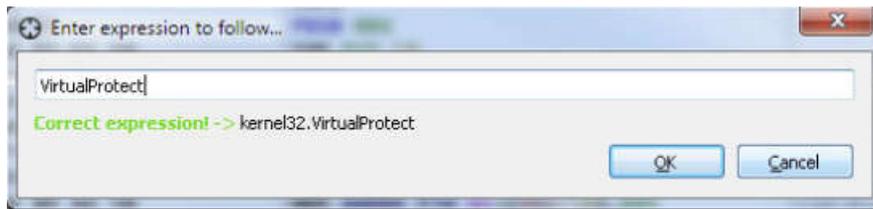
Ceci étant dit, passons aux vif du sujet. Installez x64dbg et allez dans le menu **Option** puis **preferences** et configurez-le comme ceci:



Ouvrez l'exécutable **ezcd.exe** et on arrive sur l'Entry Point.

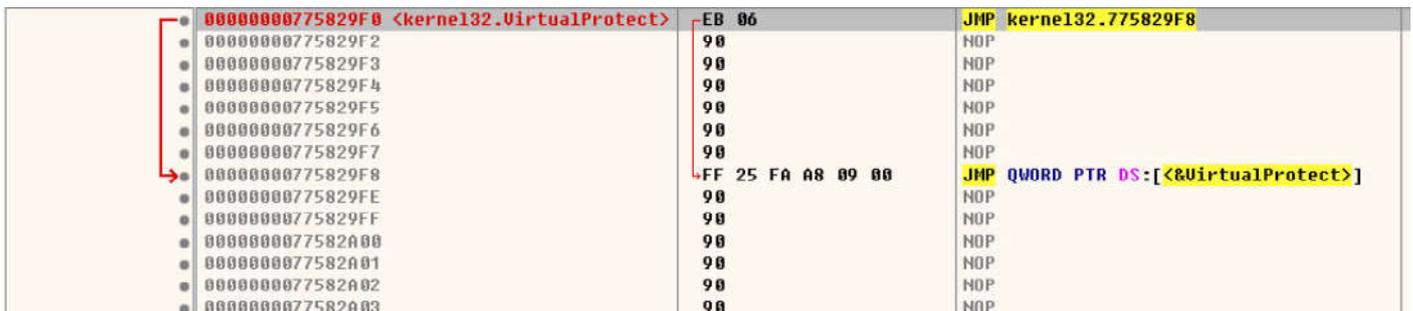


Faites un clic-droit sur le contenu de la fenêtre du code et sélectionnez **Go to -> Expression**
Tapez alors **VirtualProtect**.

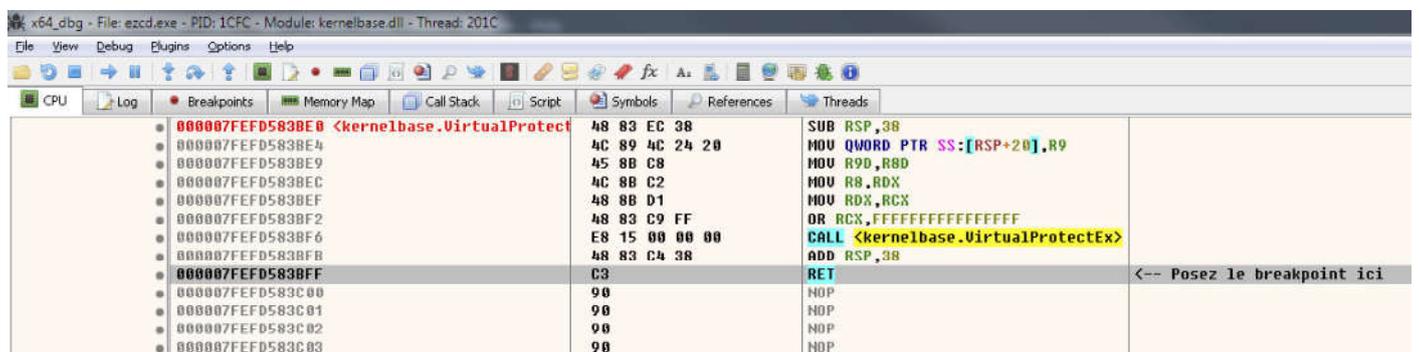


C'est l'API qui va nous servir à breaker au bon endroit. Juste avant qu'armadillo résolve les API pour notre table d'imports. Le but ici étant d'empêcher Armadillo de détourner les API système dans la table d'imports d'ezcdda.

Cliquez sur OK et on se retrouve sur le début de l'API. Contrairement au code x86, ici on ne tombe pas directement sur son code mais sur deux JMP consécutifs.



Tapez deux fois sur la touche **Entrée** et on se retrouve enfin sur l'API.



On peut alors placer notre breakpoint sur le RET final de l'API (touche F2).

Pourquoi ici et pas directement sur le premier JMP ? Armadillo détecte les breakpoints sur les premiers octets des API. J'ai choisi RET final parceque ça nous évitera de tracer le code de l'API. Un simple appuis sur la touche F7 ou F8 suffira à sortir et de nous retrouver directement dans le code du programme.

Une fois le breakpoint posé, nous pouvons lancer le programme avec **Shift+F9**.

A partir de maintenant, nous allons breaker plusieurs fois sur **VirtualProtect**. J'ai compté qu'il fallait appuyer 20 fois sur **Shift+F9** pour breaker sur le VirtualProtect qui nous interesse.

Mais il y a une méthode de suivit plus simple. Regardez le contenu de la pile. Après quelques appuis sur **Shift+F9**, vous verrez apparaitre des nom de sections. D'abord **.text** puis **.rdata**, **.data**, **.pdata**, **.tls**, **.rsrc** et enfin **.reloc** pour la dernière.

Concrètement, voici ce que cela donne.

```

0000000012EF38 0000000014C8F25 return to ezcd.0000000014C8F25 from ezcd.0000000015752F8
0000000012EF40 0000000038700000
0000000012EF48 0000000001A978C
0000000012EF50 000000003A55000
0000000012EF58 0000000100001108
0000000012EF60 00000000012EFB8
0000000012EF68 00000000012F008
0000000012EF70 00000000386D840
0000000012EF78 0000000001E8000
0000000012EF80 000000003870000
0000000012EF88 0000000036A0370
0000000012EF90 0000000180000000
0000000012EF98 0000000036A0168
0000000012EFA0 0000000036A0150 "PE"
0000000012EFA8 0000000036A0258 ".text"
0000000012EFB0 0000000036A0154
  
```

Et voici ce que cela donne sur l'avant dernier break. On voit bien le **.reloc** et à ce moment là, il ne nous reste qu'un seul **Shift+F9** à faire.

```

0000000012EF38 0000000014C8F25 return to ezcd.0000000014C8F25 from ezcd.0000000015752F8
0000000012EF40 0000000038800000
0000000012EF48 0000000001A978C
0000000012EF50 000000003A95000
0000000012EF58 0000000100001108
0000000012EF60 00000000012EFB8
0000000012EF68 00000000012F008
0000000012EF70 0000000038AD840
0000000012EF78 0000000001E8000
0000000012EF80 000000003880000
0000000012EF88 0000000036E0370
0000000012EF90 0000000180000000
0000000012EF98 0000000036E0168
0000000012EFA0 0000000036E0150 "PE"
0000000012EFA8 0000000036E0348 ".reloc"
0000000012EFB0 0000000036E0154
  
```

Ici vous faites votre dernier **Shift+F9** et vous pouvez d'ailleurs constater que le temps est plus long pour breaker sur le RET final de **VirtualProtect**. Cela aussi est un indice. Avec l'habitude (et c'est valable sur x86) vous n'aurez plus besoin ni de compter ni de regarder la pile. Le temps plus long entre deux break vous indiquera que c'est le dernier.

Vous retirez le breakpoint sur le RET final de **VirtualProtect**. Il ne nous sert plus à rien et vous faites un **F8**. On se retrouve dans le code d'armadillo.

Contrairement aux version x86, ici il n'y pas de layer de chiffrement de la routine de résolution d'imports et on a directement la routine en clair.

```

RIP 000000003931094 8B 84 24 F0 29 00 00 MOV EAX,DWORD PTR SS:[RSP+29F0]
000000003931098 8B 8C 24 00 2A 00 00 MOV ECX,DWORD PTR SS:[RSP+2A00]
00000000393109A 48 8B 94 24 E8 26 00 00 MOV RDX,QWORD PTR SS:[RSP+26E8] [rsp+26E8]:MZP
00000000393109C 48 03 D1 ADD RDX,RCX
00000000393109E 48 8B CA MOV RCX,RDX
0000000039310A0 44 8B C0 MOV R8D,EAX
0000000039310A2 48 8B 94 24 F8 29 00 00 MOV RDX,QWORD PTR SS:[RSP+29F8]
0000000039310A4 E8 80 B2 07 00 CALL 39AC340
0000000039310A6 8B 84 24 F0 29 00 00 MOV EAX,DWORD PTR SS:[RSP+29F0]
0000000039310A8 8B 8C 24 00 2A 00 00 MOV ECX,DWORD PTR SS:[RSP+2A00]
0000000039310AA 48 8B 94 24 E8 26 00 00 MOV RDX,QWORD PTR SS:[RSP+26E8] [rsp+26E8]:MZP
0000000039310AC 48 03 D1 ADD RDX,RCX
0000000039310AE 48 8B CA MOV RCX,RDX
0000000039310B0 4C 8D 8C 24 DC 29 00 00 LEA R9,QWORD PTR SS:[RSP+29DC]
0000000039310B2 44 8B 84 24 DC 29 00 00 MOV R8D,DWORD PTR SS:[RSP+29DC]
0000000039310B4 8B D0 MOV EDX,EAX
0000000039310B6 FF 15 BC 20 0E 00 CALL QWORD PTR DS:[<VirtualProtect]
0000000039310B8 48 8B 84 24 F8 29 00 00 MOV RAX,QWORD PTR SS:[RSP+29F8]
  
```

Ici on a deux boucles imbriquées et au centre de celles-ci, la routine qui va décider si oui ou non une API devra être détournée. La première boucle permet de charger chacune des DLL comme ADVAPI32.DLL, KERNEL32.DLL etc... Et à l'intérieur de cette boucle, une autre est chargée de résoudre chaque import pour chacune des DLL

Ici l'objectif est multiple.

- 1) Patcher le Magic Jump afin d'empêcher Armadillo de détourner certaines API.
- 2) Trouver l'adresse de l'Import Table
- 3) Empêcher la détection par GetTickCount
- 4) Eviter que le programme plante quand il cherchera les API d'ARMACCESS64.DLL
- 5) Trouver la fin des boucles

Les objectifs 2 et 3 sont facilement trouvable grâce à l'appel à GetTickCount car ils sont côte-à-côte. L'objectif 1 (magic jump) est lui aussi facilement trouvable via un opcode. Un **MOV [REGISTRE], 100** Les objectif 4 et 5 sont liés et trouvables eux aussi via une série d'opdoce facilement identifiables.

D'abord, commençons par trouver un appel à GetTickCount. Descendez dans le listing. Il est assez loin mais ne vous inquiétez pas, il y en a que deux. Donc quand vous l'avez, vous savez que vous êtes au bon endroit. Voilà à quoi ça ressemble.

000000003931E78	C7 00 03 00 00 00	MOV DWORD PTR DS:[RAX],3	
000000003931E7E	FF 15 5C 18 0E 00	CALL QWORD PTR DS:[<&GetLastError>]	
000000003931E84	89 44 24 20	MOV DWORD PTR SS:[RSP+20],EAX	
000000003931E88	4C 8D 8C 24 D0 34 00 0E	LEA R9,QWORD PTR SS:[RSP+34D0]	
000000003931E90	4C 8B 84 24 80 28 00 0E	MOV R8,QWORD PTR SS:[RSP+2880]	
000000003931E98	48 8D 15 99 B5 0E 00	LEA RDX,QWORD PTR DS:[3A1D438]	3A1D438:"File \"%s\", fonction \"%s\" (error %d)"
000000003931E9F	48 8B 84 24 60 59 00 0E	MOV RAX,QWORD PTR SS:[RSP+5960]	
000000003931EA7	48 8B 48 08	MOV RCX,QWORD PTR DS:[RAX+8]	
000000003931EAB	E8 2C B2 07 00	CALL 39AD0DC	
000000003931EB0	33 C0	XOR EAX,EAX	
000000003931EB2	E9 FC 10 00 00	JMP 3932FB3	
000000003931EB7	48 8B 84 24 00 27 00 0E	MOV RAX,QWORD PTR SS:[RSP+2700]	
000000003931EBF	48 39 84 24 28 27 00 0E	CMP QWORD PTR SS:[RSP+2728],RAX	
000000003931EC7	73 27	JNB 3931EF0	<-- LE JNB aide à savoir si on est au bon endroit
000000003931EC9	48 8B 84 24 28 27 00 0E	MOV RAX,QWORD PTR SS:[RSP+2728]	
000000003931ED1	48 8B 8C 24 C0 34 00 0E	MOV RCX,QWORD PTR SS:[RSP+34C0]	
000000003931ED9	48 89 08	MOV QWORD PTR DS:[RAX],RCX	<- Inscrit l'adresse de l'API dans la table d'imports
000000003931EDC	48 8B 84 24 28 27 00 0E	MOV RAX,QWORD PTR SS:[RSP+2728]	
000000003931EE4	48 83 C0 08	ADD RAX,8	
000000003931EE8	48 8B 84 24 28 27 00 0E	MOV QWORD PTR SS:[RSP+2728],RAX	
000000003931EF0	E9 7D FB FF FF	JMP 3931A72	
000000003931EF5	FF 15 BD 17 0E 00	CALL QWORD PTR DS:[<&GetTickCount>]	<-- Le GetTickCount à trouver
000000003931EFB	2B 84 24 50 2A 00 00	SUB EAX,DWORD PTR SS:[RSP+2A50]	
000000003931F02	8B 8C 24 40 2A 00 00	MOV ECX,DWORD PTR SS:[RSP+2A40]	
000000003931F09	6B C9 32	IMUL ECX,ECX,32	
000000003931F0C	81 C1 D0 07 00 00	ADD ECX,7D0	
000000003931F12	3B C1	CMP EAX,ECX	
000000003931F14	76 08	JBE 3931F1E	
000000003931F16	C6 84 24 10 28 00 00 01	MOV BYTE PTR SS:[RSP+2810],1	<- Détection par le GetTickCount à patcher en 0
000000003931F1E	48 83 BC 24 F0 27 00 0E	CMP QWORD PTR SS:[RSP+27F0],0	
000000003931F27	0F 85 B1 00 00 00	JNZ 3931FDE	

Avec les commentaires que j'ai placés, vous voyez toute de suite où je veux en venir. En dessous du **GetTickCount**, vous voyez quelques opérations sur la valeur que l'API retourne et un test. Ce dernier va positionner une variable sur 1 si le test échoue. On va faire simple, patcher le 1 en 0.

Au lieu d'avoir un

```
MOV BYTE PTR SS:[RSP+2810],1
```

nous aurons un

```
MOV BYTE PTR SS:[RSP+2810],0
```

Rappelez-vous où vous avez patché car il faudra rétablir le code à la fin. Faites un gri-gri dans la zone de commentaire du debugger pour repérer facilement l'endroit ou notez l'adresse quelque part. Rassurez-vous, il n'y a que deux patch à faire en tout et pour tout.

Objectif 3 rempli.

Maintenant, au dessus du GetTickCount vous voyez l'instruction **MOV QWORD PTR DS:[RAX],RCX** Comme il est écrit dans les commentaires sur la photo, cette instruction va placer l'adresse de l'API contenue par RCX dans l'import table qui se trouvera à l'adresse de RAX. **Placez un breakpoint sur cette instruction.** Cela va vous aider à trouver la première API de l'import table tout à l'heure quand vous allez utiliser Scylla. Cela nous permettra aussi de savoir où Armadillo va placer ses API dans l'import table.

Objectif 2 accomplis.

A partir d'ici, remontez le code (une page environ) pour trouver un **MOV [REGISTRE], 100**

```
000000003931CA5 48 8D 94 24 F0 3C 00 00 LEA RDX,QWORD PTR SS:[RSP+3CF0]
000000003931CAD 48 8D 8C 24 D0 34 00 00 LEA RCX,QWORD PTR SS:[RSP+34D0]
000000003931CB5 E8 0E C3 08 00 CALL 39BDFC8
000000003931CBA 85 C0 TEST EAX,EAX
000000003931CBC 75 16 JNZ 3931CD4 <-- Magic Jump
000000003931CBE 48 8B 84 24 E0 3C 00 00 MOV RAX,QWORD PTR SS:[RSP+3CE0]
000000003931CC6 48 8B 40 10 MOV RAX,QWORD PTR DS:[RAX+10]
000000003931CCA 48 89 84 24 C0 34 00 00 MOV QWORD PTR SS:[RSP+34C0],RAX
000000003931CD2 EB 02 JMP 3931CD6
000000003931CD4 EB 8E JMP 3931C64
000000003931CD6 8B 84 24 40 2A 00 00 MOV EAX,DWORD PTR SS:[RSP+2A40]
000000003931CDD FF C0 INC EAX
000000003931CDF 89 84 24 40 2A 00 00 MOV DWORD PTR SS:[RSP+2A40],EAX
000000003931CE6 EB 4D JMP 3931D35
000000003931CE8 BA 00 01 00 00 MOV EDX,100 <-- La valeur 100 à trouver
000000003931CED 48 8D 8C 24 18 27 00 00 LEA RCX,QWORD PTR SS:[RSP+2718]
000000003931CF5 E8 26 71 F8 FF CALL 38B8E20
000000003931CFA 0F B6 C0 MOVZX EAX,AL
000000003931CFD 99 CDQ
000000003931CFF 80 14 00 00 00 MOV ECX,14
```

Une fois la valeur **100** trouvé, juste au dessus vous apercevez un **TEST EAX, EAX**, **EAX** suivit d'un **JNZ**. C'est tout simplement une comparaison de chaine. Armadillo va déterminer, suivant le nom de l'API, si cette dernière doit être détournée au pas. **De ce fait et pour éviter cela, on patch le JNZ en JMP.**

Au lieu d'avoir:

```
CALL 39BDFC8
TEST EAX,EAX
JNZ 3931CD4
```

nous patchons en:

```
CALL 39BDFC8
TEST EAX,EAX
JMP 3931CD4
```

Et comme tout à l'heure, on fait un gri-gri dans les commentaire du debugger ou on note l'adresse car il faudra rétablir le code.

Objectif 1 terminé.

Nous avons fait pas mal de chemin jusqu'à présent mais tout n'est pas fini. Il nous faut maintenant trouver la fin de boucle. C'est à dire l'endroit où Armadillo aura fini de construire la table d'import et continuera son petit bonhomme de chemin pour rejoindre l'OEP.

Mais avant cela, il nous faut trouver l'endroit où Armadillo passe d'une DLL à la suivante afin de ne pas rater **ARMACCESS64.DLL**. Vous verrez que les deux emplacements sont liés et très proches. C'est pour cela que je fait les deux en même temps.

A cet instant, vous-vous trouvez juste après la sortie de **VirtualProtect**. Si ce n'est pas le cas, faite un clic-droit dans la fenêtre du listing et Go to -> Origin

A partir de là, cherchez la séquence d'opcode:

```
XOR EAX,EAX
CMP EAX,1
JE 38D1FE3
```

Vous en trouvez une première. Ce n'est pas celle là. C'est la **deuxième** à partir de la sortie de **VirtualProtect**.

Elle est facilement discernable car elle est suivie à quelques opcode d'un:

```
TEST EAX,EAX
JNZ 3921468
JMP 3921FE3
```

Voici une photo vous montrant à quoi ça ressemble. C'est assez distinctif et avec l'habitude, vous pourrez les repérer rapidement.

0000000039213D4	FF C0	INC EAX	
0000000039213D6	89 84 24 04 56 00 00	MOV DWORD PTR SS:[RSP+5604],EAX	
0000000039213D8	8B 84 24 B4 56 00 00	MOV EAX,DWORD PTR SS:[RSP+56B4]	
0000000039213E4	89 84 24 0C 28 00 00	MOV DWORD PTR SS:[RSP+280C],EAX	
0000000039213EB	48 8B 84 24 C8 25 00 00	MOV RAX,QWORD PTR SS:[RSP+25C8]	[rsp+25C8]:HP
0000000039213F3	48 89 84 24 C0 25 00 00	MOV QWORD PTR SS:[RSP+25C0],RAX	
0000000039213FB	48 C7 84 24 D0 25 00 00	MOV QWORD PTR SS:[RSP+25D0],0	
000000003921407	48 C7 84 24 28 27 00 00	MOV QWORD PTR SS:[RSP+2728],0	
000000003921413	48 C7 84 24 00 27 00 00	MOV QWORD PTR SS:[RSP+2700],FFFFFFFF	
00000000392141F	33 C0	XOR EAX,EAX	
000000003921421	83 F8 01	CMPEAX,1	<-- Le CMP EAX, 1 à trouver
000000003921424	0F 84 B9 0B 00 00	JE 3921FE3	
00000000392142A	48 8B 84 24 C0 25 00 00	MOV RAX,QWORD PTR SS:[RSP+25C0]	<-- RAX contiendra le nom de la DLL dont ARMACCESS64
000000003921432	48 89 84 24 80 2B 00 00	MOV QWORD PTR SS:[RSP+2B80],RAX	<-- Mettre un breakpoint ici
00000000392143A	33 D2	XOR EDX,EDX	
00000000392143C	48 8B 8C 24 C0 25 00 00	MOV RCX,QWORD PTR SS:[RSP+25C0]	
000000003921444	E8 DF C6 07 00	CALL 399DB28	
000000003921449	48 FF C0	INC RAX	
00000000392144C	48 89 84 24 C0 25 00 00	MOV QWORD PTR SS:[RSP+25C0],RAX	
000000003921454	48 8B 84 24 80 2B 00 00	MOV RAX,QWORD PTR SS:[RSP+2B80]	
00000000392145C	0F BE 00	MOVSX EAX,BYTE PTR DS:[RAX]	
00000000392145F	85 C0	TEST EAX,EAX	
000000003921461	75 05	JNZ 3921468	
000000003921463	E9 7B 0B 00 00	JMP 3921FE3	<-- Jump final. Mettre un breakpoint ici
000000003921468	48 8B 84 24 C0 25 00 00	MOV RAX,QWORD PTR SS:[RSP+25C0]	
000000003921478	8B 00	MOV EAX,DWORD PTR DS:[RAX]	

Placez un breakpoint comme indiqué sur la photo pour pouvoir vérifier à chaque DLL qu'il ne s'agit pas d'ARMACCESS64. Effectivement, si sous OllyDbg on pouvait placer un breakpoint conditionnel sur la valeur de RAX, avec x64dbg ce n'est pas possible. Ce type de breakpoint n'est pas encore développé. C'est juste ennuyeux mais pas insurmontable. Il va nous falloir faire des **Shift+F9** pour chaque DLL et être très vigilant quand on arrivera sur ARMACCESS64.

Objectifs 4 et 5 remplis

Placez un autre breakpoint sur le JMP final. Sinon, Armadillo, quand il aura fini, va nous échapper et tout sera à recommencer.

Vous avez tout préparé ? Patché le GetTickCount, patché le JNE du magic jump, placé un breakpoint pour récupérer l'adresse de l'import table, un autre sur le jmp final et un dernier sur le test de fin des DLL ?

Alors si c'est bon, on peut se lancer. Faites un **Shift+F9** et on atterrit sur le BP des DLL. D'ailleurs on peut, comme prévu, apercevoir ADVAPI32 dans le registre RAX.

On continue en faisant un autre **Shift+F9** et cette fois on rejoint l'endroit où Armadillo sauvegarde l'adresse de l'API dans l'Import table. A ce propos, on note que RCX contient bien l'adresse de l'API. A ce stade, vous pouvez faire un Follow in dump du registre RAX.

Désactivez (ne l'effacez pas) le breakpoint car il va nous gêner. Breaker sur chaque dll c'est déjà assez fatiguant en soit alors sur chaque API c'est carrément l'enfer. Désactivé car il nous servira plus tard.

Faites **Shift+F9** et on se retrouve encore sur le breakpoint des DLL mais cette fois-ci, vous pouvez constater que c'est KERNEL32 qui est pointé par le registre RAX.

Continuez comme ceci et faites des **Shift+F9** jusqu'à que **RAX affiche ARMACCESS64.DLL**

A ce stade, il va nous falloir **réactiver le breakpoint** où Armadillo sauve l'adresse de l'API dans l'import table et rétablir le patch sur le magic jump. **Restaurer le JMP en JNE**.

Pourquoi rétablir le magic jump ? *Si vous ne le faites pas, Armadillo ne trouvera pas ses propres API. Il vous affichera une MessageBox et tout sera à recommencer depuis le début. Ce n'est pas ce que vous voulez, n'est-ce pas ?*

Pourquoi rétablir le breakpoint ? *Afin de connaître les adresses où sont stocké les API d'Armadillo dans l'import table mais surtout, savoir de quelles API il s'agit. Avoir leur nom et leur emplacement. Cela nous servira à les intégrer dans le fichier tree au format xml de Scylla.*

Vous avez **restauré le JMP en JNE** et rétabli le breakpoint. Faites **Shift+F9** et vous arrivez de nouveau là:

```

0000000003921EC9  48 8B 84 24 28 27 00 00 MOV RAX,QWORD PTR SS:[RSP+2728]
0000000003921ED1  48 8B 8C 24 C0 34 00 00 MOV RCX,QWORD PTR SS:[RSP+34C0]
> 0000000003921ED9  48 89 08                MOV QWORD PTR DS:[RAX],RCX
0000000003921EDC  48 8B 84 24 28 27 00 00 MOV RAX,QWORD PTR SS:[RSP+2728]
0000000003921EE4  48 83 C0 08            ADD RAX,8
0000000003921EE8  48 89 84 24 28 27 00 00 MOV QWORD PTR SS:[RSP+2728],RAX
-> 0000000003921EF0  E9 7D FB FF FF        JMP 3921A72
0000000003921EF5  FF 15 BD 17 0E 00     CALL QWORD PTR DS:[<&GetTickCount>]
0000000003921EFB  2B 84 24 50 2A 00 00  SUB EAX,DWORD PTR SS:[RSP+2A50]
0000000003921F02  8B 8C 24 40 2A 00 00  MOV ECX,DWORD PTR SS:[RSP+2A40]
0000000003921F08  48 C0 32                IMUL ECX,ECX,32

```

Regardez dans la pile et vous constatez qu'elle contient le nom de l'API Armadillo. Ici **ExpireCurrentKey**

```

000000000128840  0000000003A0A2B0
000000000128848  00000000012C530  "ExpireCurrentKey"
000000000128850  000000000000010
000000000128858  000000000000000
000000000128860  000000000000000
000000000128868  000000000000000
000000000128870  000000000000000
000000000128878  000000000000000
000000000128880  000000000000000

```

Vous ouvrez Notepad et notez deux choses. Le nom de l'API et l'adresse de son emplacement dans la table d'imports. Registre RAX. Ici j'ai ceci:

ExpireCurrentKey = 0000000014BA2BC

Procédez comme cela pour chacune des API d'Armadillo. C'est long, c'est chiant mais c'est nécessaire.

Petite précision toutefois. Sur la version x64 d'ezcdda, Armadillo utilise les API suivantes:

```

SECUREBEGIN
SECUREEND
SECUREBEGIN_A
SECUREBEGIN_C
SECUREBEGIN_D
SECUREEND_A
SECUREEND_C
SECUREEND_D

```

Oubliez-les. Bien qu'elles soient placées dans l'import table il n'y a aucun appel dans ezcdda. D'ailleurs, au final elles ne pointent que vers un RET. Autant dire qu'elles ne font rien. De mon côté, parce que j'aime les choses «finies», j'ai quand même inclus ces API dans la table d'imports. Faites comme vous voulez. La dll Armaccess64.dll que je fournis les supporte si vous décidez de les inclure.

Résolvez toutes les API arma pour finalement finir sur le breakpoint du test des DLL. Vous constatarez que le breakpoint final n'est pas loin. Tracez avec **F8** jusqu'à lui.

Rétablissez le code sur le trick de GetTickCount (remettez le 1 à la place du 0) et effacez tous les breakpoints. Armadillo ayant tendance à vérifier son code pour détecter des changements.

Faire un **Go to -> Expression** et tapez **CreateThread**. Faites OK, placez un breakpoint sur le RET final de l'API et un **Shift+F9** pour rejoindre l'emplacement. Une fois là, **F8** pour retourner dans le code d'Armadillo. Tracez encore avec **F8** pour sortir de cette routine encore et vous devriez vous retrouver là.

0000000003928A52	48 8D 0D F7 AD 12 00	LEA RCX,QWORD PTR DS:[3A53850]
0000000003928A59	E8 12 1A F8 FF	CALL 38AA470
0000000003928A5E	0F B6 C0	MOVZX EAX,AL
0000000003928A61	85 C0	TEST EAX,EAX
0000000003928A63	75 0A	JNZ 3928A6F
0000000003928A65	C7 44 24 5C 01 00 00 00	MOV DWORD PTR SS:[RSP+5C],1
0000000003928A6D	EB 08	JMP 3928A77
0000000003928A6F	C7 44 24 5C 00 00 00 00	MOV DWORD PTR SS:[RSP+5C],0
0000000003928A77	0F B6 44 24 5C	MOVZX EAX,BYTE PTR SS:[RSP+5C]
0000000003928A7C	85 C0	TEST EAX,EAX

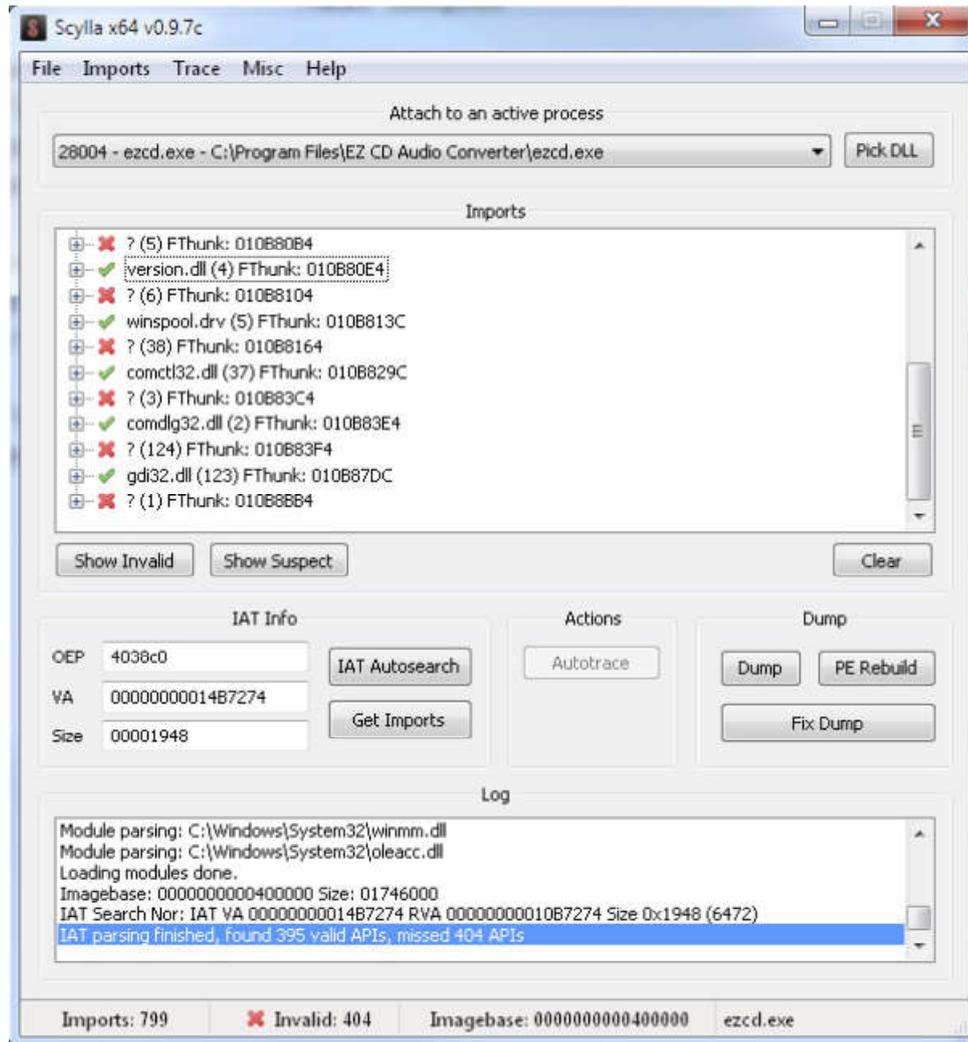
Nous ne sommes plus très loin de l'OEP. Tracez avec **F8** ou bien descendez la fenêtre pour apercevoir un **CALL [REGISTRE]**. Ici c'est un **CALL RAX**. **Attention, il y en a deux**. Celui qui nous intéresse est le dernier. Celui qui se trouve juste avant le **RET**. Placez-y un breakpoint, faites **Shift+F9**. Retirez le breakpoint et faites **F7**. Vous êtes sur l'OEP.

Nous sommes sur l'OEP, nous avons une import table propre. Il ne nous reste plus qu'à dumper notre process et reconstruire l'IAT. Pour cela, x64dbg contient le plugin Scylla. Un Imprec-like pour x64. Donc direction le menu plugins et on active Scylla.

On le configure comme ceci.



On entre l'adresse de l'EOP (et non la RVA comme sur Imprec). Soit **4038C0** puis on clique sur le bouton **IAT Autosearch** puis **Get Imports** et on obtient une bouillie qui ressemble à ça.



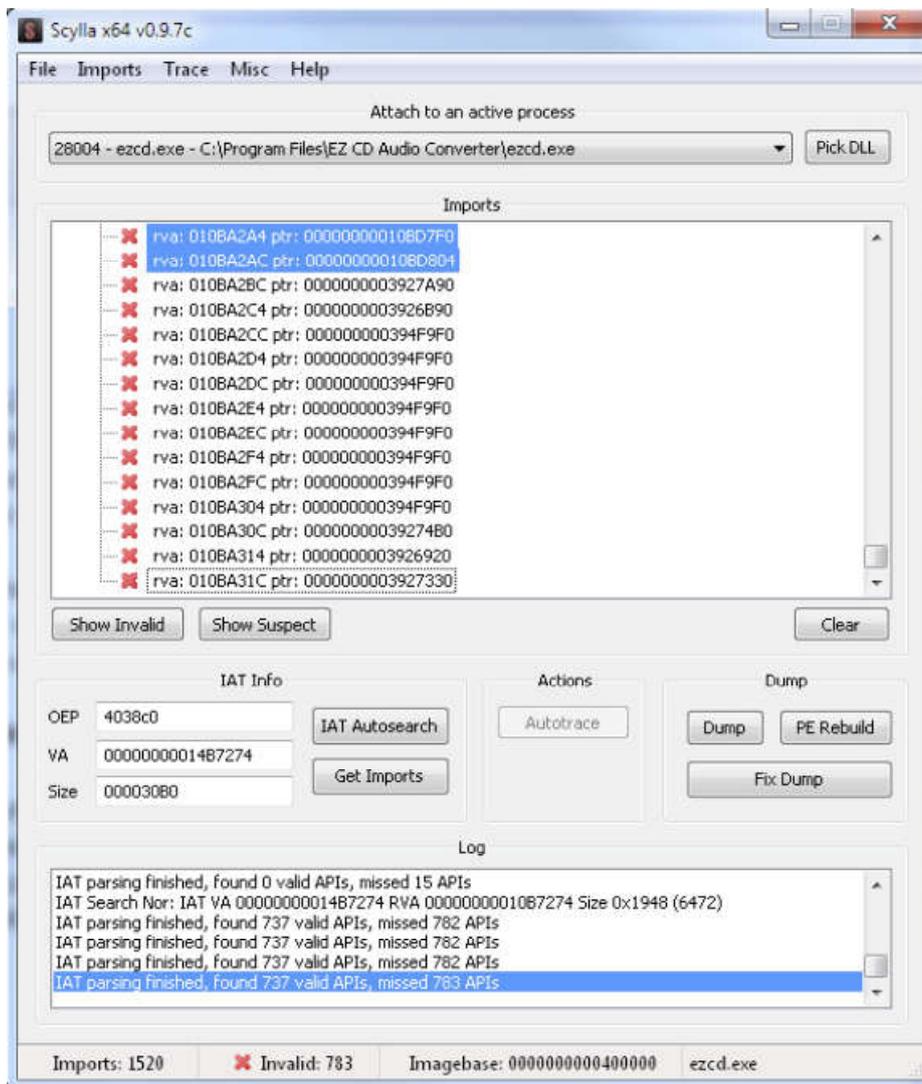
Déjà on peu voir que, comme prévu, l'IAT commence par ADVAPI32 comme nous l'avons vu tout à l'heure dans Armadillo. Scylla semble avoir fait du bon boulot à première vu mais ça n'est pas le cas. Si vous regardez la fin de l'IAT comme démontré sur la photo, on constate que l'IAT s'arrête avec GDI32. Or, tout à l'heure nous avons breaké sur chaque DLL importé et la dernière avant ARMACCESS64 était WS2_32.

Si Scylla nous a trouvé le début de l'IAT, il s'est trompé sur sa taille. Vous connaissez l'adresse dans l'import table des API Armadillo et vous savez que c'est la dernière DLL qui à été importé. Donc faites un simple calcul des adresses et vous avez la taille réelle de l'IAT.

Ici j'ai: $(0000000010BA31C + 8) - 0000000014B7274 = 30B0$

Vous pouvez vous aider de la fenêtre de dump au format «Adress» de x64dbg pour facilement repérer les adresses. Malheureusement, il n'y a pas le nom des API à coté des adresses contrairement à OllyDbg. Certainement un bug de x64dbg.

On modifie Scylla en conséquence et on appuis sur le bouton **Clear** puis **Get Imports**. Cette fois-ci c'est un peu mieux mais on a encore beaucoup de petites croix rouge. Cliquez sur le bouton **Show invalid** puis allez tout en bas et avec la touche **Control** enfoncée, **désélectionnez** à l'aide de la souris les entrées relatives aux API Armadillo. C'est à dire de 00000000010BA2BC à 00000000010BA31C. Une petite photo pour vous montrer à quoi ça doit essembler.



Maintenant, nous allons dans le menu **Imports** et on clique que **Cut Selected**. Toutes les petites croix rouge ont disparus sauf celle concernant nos API Armadillo. Elles n'ont pas de nom encore. Vous pouvez cliquer sur **Dump** (ou plus tard, ça n'a pas d'importance) et sauver votre fichier en **ezcd_dump.exe**

Toujours dans le menu **Imports**, sauvez votre IAT en **ezcd.xml** avec **Save tree** et éditez le fichier texte avec notepad. Il va falloir procéder à quelque ajouts et modifications. Il va falloir déclarer armaccess64.dll ainsi que toutes les API que vous avez noté tout à l'heure lors de l'unpack. A l'aide des adresses vous serez sûr de les placer au bon endroit.

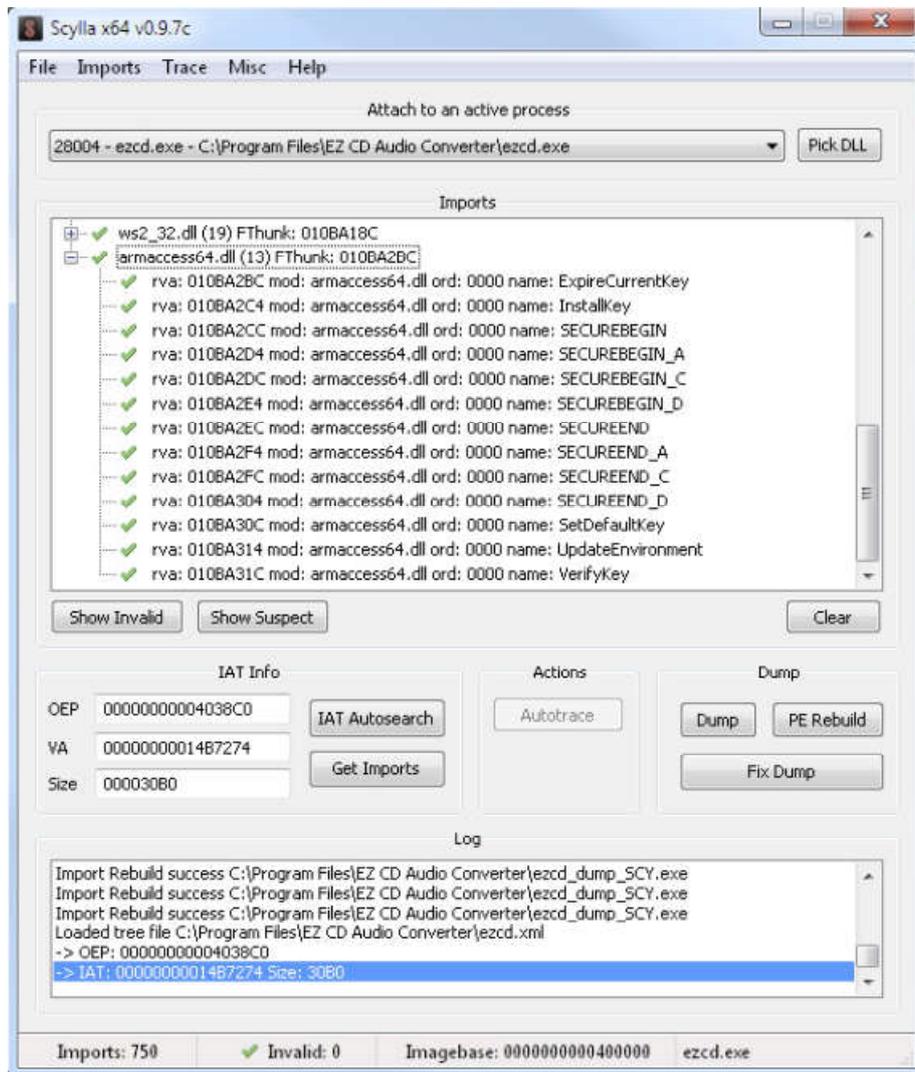
Voici à quoi ressemble la zone à modifier (situé à la fin du fichier)

```
<module filename="" first_thunk_rva="00000000010BA2BC">
  <import_invalid iat_rva="00000000010BA2BC" address_va="0000000003927A90" />
  <import_invalid iat_rva="00000000010BA2C4" address_va="0000000003926B90" />
  <import_invalid iat_rva="00000000010BA2CC" address_va="000000000394F9F0" />
  <import_invalid iat_rva="00000000010BA2D4" address_va="000000000394F9F0" />
  <import_invalid iat_rva="00000000010BA2DC" address_va="000000000394F9F0" />
  <import_invalid iat_rva="00000000010BA2E4" address_va="000000000394F9F0" />
  <import_invalid iat_rva="00000000010BA2EC" address_va="000000000394F9F0" />
  <import_invalid iat_rva="00000000010BA2F4" address_va="000000000394F9F0" />
  <import_invalid iat_rva="00000000010BA2FC" address_va="000000000394F9F0" />
  <import_invalid iat_rva="00000000010BA304" address_va="000000000394F9F0" />
  <import_invalid iat_rva="00000000010BA30C" address_va="00000000039274B0" />
  <import_invalid iat_rva="00000000010BA314" address_va="0000000003926920" />
  <import_invalid iat_rva="00000000010BA31C" address_va="0000000003927330" />
</module>
```

Et voici à quoi elle doit ressembler

```
<module filename="armaccess64.dll" first_thunk_rva="0000000010BA2BC">
<import_valid name="ExpireCurrentKey" suspect="0" iat_rva="0000000010BA2BC" />
<import_valid name="InstallKey" suspect="0" iat_rva="0000000010BA2C4" />
<import_valid name="SECUREBEGIN" suspect="0" iat_rva="0000000010BA2CC" />
<import_valid name="SECUREBEGIN_A" suspect="0" iat_rva="0000000010BA2D4" />
<import_valid name="SECUREBEGIN_C" suspect="0" iat_rva="0000000010BA2DC" />
<import_valid name="SECUREBEGIN_D" suspect="0" iat_rva="0000000010BA2E4" />
<import_valid name="SECUREEND" suspect="0" iat_rva="0000000010BA2EC" />
<import_valid name="SECUREEND_A" suspect="0" iat_rva="0000000010BA2F4" />
<import_valid name="SECUREEND_C" suspect="0" iat_rva="0000000010BA2FC" />
<import_valid name="SECUREEND_D" suspect="0" iat_rva="0000000010BA304" />
<import_valid name="SetDefaultKey" suspect="0" iat_rva="0000000010BA30C" />
<import_valid name="UpdateEnvironment" suspect="0" iat_rva="0000000010BA314" />
<import_valid name="VerifyKey" suspect="0" iat_rva="0000000010BA31C" />
</module>
```

Retournez dans Scylla et cette fois, encore dans le menu **Imports**, vous faites **Load tree**. Vous constatez que tout à l'air correct maintenant comme le montre la photo suivante.



Il ne nous reste plus qu'à cliquer sur le bouton **Fix Dump**, sélectionnez votre ezcd_dump.exe et de valider. Scylla va sauver votre fichier reconstruit avec l'ajout de _SCY dans son nom.

Nous avons un dumped valide qui se charge bien (N'oubliez pas de copier la dll armaccess64.dll dans le dossier d'ezcdda). Il affiche toujours la box d'enregistrement mais c'est normal. Les Secured Sections sont celle de la version non enregistrée. Je vous rassure, les Secured Sections sont assez rare. Le plus difficile ici était de trouver une cible protégée par Armadillo x64.

Conclusion:

Au final, unpacker de l'Armadillo x64 est assez trivial. Pour celui qui à déjà fait ça en x86 il ne sera pas déboussolé. La procédure est la même, les fonctions sont les même. Les sauts, les calls, les jumps sont identiques à une version x86. La seule chose qui change, ce sont les registres. Cela peut dérouter au départ mais on s'y habitue très vite. Une chose qui manquait jusqu'à présent, c'était les outils. Comme x64dbg qui est un parfait clône d'Ollydbg mais qui est certainement plus adapté qu'IDA/Windbg (à mon sens). Scylla est également un merveilleux outil et sans lui, rien ne serait possible.

Remerciements:

- à toute l'équipe de la FFF pour me supporter depuis toutes ces années;
- à AddS pour m'avoir rappelé que x64_dbg existait et que j'avais complètement oublié depuis;
- à doodle pour m'avoir aidé pour armaccess64.dll;
- à vous, pour prendre le temps de me lire;

Evidemment, un grand merci à Aguila pour avoir créé Scylla. Ainsi qu'aux auteurs et contributeurs de x64_dbg.

PS:

Je voulais également dans ce tutorial, inclure toute la procédure de nettoyage d'un dumped. Restaurer les véritables noms des sections, supprimer les sections Armadillo obsolètes, rétablir certaines données dans le PE Header mais cela est trop long et sincèrement inutile ici.