

# Keygenning de WJChess 3D

Version 1.0  
22/02/2011

## Table des matières

1. Propos liminaires
  - a) Généralités
  - b) Prérequis
  - c) Outils utilisés
2. Approche générale
  - a) Détection packers & crypto
  - b) Recherche de la routine
  - c) Vérification du md5 utilisé
3. Keygenning
  - a) Analyse de la routine n°1
  - b) Analyse de la routine n°2
4. Conclusion et remerciements
5. Références et documentation
6. Historique

---

*Template by Shub-Nigurath*

## 1. Propos liminaires

### a) Généralités

Bonjour,

La cible du jour est un logiciel nommé WJChess 3D, repassé assez récemment shareware. Je vous propose dans ce tutoriel d'étudier le schéma de génération du sérial, avec pour but final la création d'un keygen. Le niveau du tutoriel n'est pas très levé, mais il demande un minimum d'expérience en reversing, et plus particulièrement en assembleur pour la compréhension de la routine, et en cryptographie basique (la protection se basant en effet sur du MD5 non modifié).

Nous effectuerons dans un premier temps un état des lieux au niveau de la protection (packing / crypto / type d'enregistrement...), puis nous chercherons à localiser la routine de calcul (ou les routines) du sérial.

Dans un second temps, nous analyserons ce que nous avons trouvé précédemment, afin d'en déduire les règles de calcul pour pouvoir les reproduire dans notre keygen. Je vous ai joint dans l'archive mon keygen, codé comme d'habitude en assembleur, syntaxe masn.

*Je rappelle que je ne peux en AUCUN CAS être tenu responsable d'un dommage survenant sur votre PC lors de la mise en pratique de ce tuto.*

*Il est également possible que des erreurs se soient glissées dans ce document. Si tel était le cas, merci de me les rapporter pour que je puisse les corriger.*

### b) Prérequis

Pour suivre ce tutoriel, il est recommandé de posséder de bonnes bases en langage assembleur, de savoir utiliser les outils basiques en RE, et d'avoir une connaissance sommaire de la cryptographie. Des liens seront proposés en fin de document pour consolider ou approfondir vos connaissances dans ces domaines. Ce tutoriel se veut assez explicite et détaillé, et donc par là le plus abordable possible, aux gens ayant peu d'expérience dans ce domaine, ne soyez donc pas rebutés par son aspect technique. Bonne lecture ;)

### c) Outils utilisés

- Peid 0.95 + Protection ID
- Keygenner Assistant
- OllyDbg 1.10 + ses plugins CommandBar et NameChanger
- WinAsm Studio
- Masm 10
- WJChess 3D : <http://wjchess.jeffprod.com/download/wjchess3d.exe>

## 2) Approche générale

### a) Détection packers & crypto

On commence comme à l'habitude par passer le soft sous Peid afin de repérer la présence d'un éventuel packer. Nous obtenons ceci (Fig. 1) :



Figure 1 : Résultat de Peid

Peid ne nous est guère utile ici, il ne détecte rien, même dans la base de signatures externes. Regardons avec Kanal la présence de signatures d'algorithmes cryptographiques connus avant de quitter Peid. Nous obtenons ceci (Fig. 2) :



Figure 2 : Résultat de Kanal

KANAL nous détecte donc la présence d'un MD5 référencé en 0041FF37. Nous nous pencherons sur cet algorithme un peu plus loin. Passons également le soft sous Protection ID, afin de voir si celui-ci détecte quelque chose. Nous obtenons ceci :

```
Scanning -> C:\Documents and Settings\All Users\Bureau\WJChess 3D.lnk
Link Resolved to -> C:\Program Files\WJChess 3D\WJChess3D.exe
File Type : 32-Bit Exe (Subsystem : Win GUI / 2), Size : 3583488 (036AE00h) Byte(s)
[File Heuristics] -> Flag : 00000000000001001101000000000000 (0x0004D000)
[!] Possible CD/DVD-Key or Serial Check -> UNREGISTERED
[!] File appears to have no protection or is using an unknown protection
- Scan Took : 0.641 Second(s)
```

Protection ID nous détecte un sérial check, et ne trouve pas de protection de type packer. Lançons maintenant le logiciel afin de voir de quoi il en retourne. Nous avons une boîte d'enregistrement au démarrage avec deux champs, nom et sérial. Notons également le message d'erreur en cas de sérial invalide. Je ne détaillerai pas plus les limitations du logiciel, car je rappelle que le but est ici de fournir un keygen valide, et donc ainsi de supprimer toutes les protections *proprement*. Chargeons le soft dans Ollly pour voir de quoi il en retourne.

## b) Recherche de la routine

La méthode que j'utilise habituellement pour poser des breakpoints sur les apis susceptibles de récupérer le sérial ne marche pas ici, l'auteur n'effectuant pas les appels directement, ou utilisant des fonctions autres que celle de l'api Windows. Plutôt que le bpx, je vais ainsi utiliser le bp dans la CommandBar, et pour être sûr de breaker je vais mettre un bp MessageBoxA (rappelez-vous la messagebox d'erreur en cas de mauvais sérial). On lance ensuite le logiciel avec F9, et on rentre un nom et un sérial (Horgh / 123456 pour moi).

Sans surprise on break dans user32.dll, et en jetant un œil à la pile nous voyons ceci (Fig. 3) :

```
0012FED0 0041FD1F CALL to MessageBoxA from WJChess3.0041FD1D
0012FED4 00000000 hOwner = NULL
0012FED8 00455A14 Text = "Clô de licence invalide. Wrong license key."
0012FEDC 004536A4 Title = "WJChess 3D"
0012FEE0 00000010 Style = MB_OK|MB_ICONHAND|MB_APPLMODAL
0012FEE4 7C920228 ntdll.7C920228
```

Figure 3 : Paramètres de la messagebox

Notre message d'erreur est donc appelé depuis l'adresse 0041FD1D. Faisons Alt+F9 pour retourner dans le code du programme. La messagebox s'affiche, cliquez sur OK. Nous arrivons ici dans le code (Fig. 4) :

```
0041FD16 . C605 44B97800 MOV BYTE PTR DS:[78B944],0
0041FD1D . FFD7 CALL EDI
0041FD1F . 5F POP EDI
0041FD20 . 5E POP ESI
=> messagebox
ntdll.7C920228
```

Figure 4 : Call appelant MessageBoxA

La façon détournée dont je parlais précédemment va consister en un appel d'un registre, ici EDI, qui contient l'adresse de l'api à appeler. Vous trouverez exactement la même configuration pour l'appel de DialogBoxParam qui va créer la boîte de dialogue d'enregistrement. Maintenant que nous sommes un peu après la vérification du code, il s'agit de trouver ou celle-ci s'effectue, et de trouver comment est calculé le véritable sérial. Remontons donc le code à la recherche du schéma traditionnel CALL / TEST / JE (et ses variantes). En haut de la routine nous remarquons ceci, qui confirme ce que nous avons expliqué (Fig. 5) :

```

0041FB86 | . 8B1D 0C334501 MOV EBX,DWORD PTR DS:[<&USER32.DialogBoxParamA>] USER32.DialogBoxParamA
0041FB8C | . 8B3D 64334501 MOV EDI,DWORD PTR DS:[<&USER32.MessageBoxA>] USER32.MessageBoxA

```

Figure 5 : Placement des adresses

Posons un bp sur le PUSH EBP en 0041FB80 qui marque le début de la routine, redémarrons avec Ctrl+F2, et recommençons l'opération. F9, et l'on rebreake sur notre PUSH EBP. A partir de là traçons avec F8. Le CALL EBX appelle DialogBoxParamA, et donc la boite d'enregistrement. Mettons un nouveau breakpoint sur le JE qui suit, et supprimons celui sur le PUSH. Entre à nouveau nom et sérial, Ok, et nous breakons au JE, qui est pris. Suivons-le pour savoir où il mène.

Nous arrivons en 0041FC7E sur un PUSH, et nous pouvons remarquer peu après deux calls correspondant aux types classiques des schémas de vérification. Il y a donc de fortes chances pour que nous ayons découvert l'endroit où est calculé le sérial valide. Examinons maintenant le md5 pour voir si nous avons affaire à un custom ou non. La phase d'approche du programme sera ainsi terminée.

### c) Vérification du md5<sup>1</sup> utilisé

Nous avons comme offset pour la détection du md5 dans Kanal l'adresse 0041FF37. Allons voir à quoi elle correspond dans Olly (Fig. 6) :

```

0041FF2F | . 0BF8          OR EDI,EBX
0041FF31 | . 037D C0      ADD EDI,DWORD PTR SS:[EBP-40]
0041FF34 | . 8D8C0F 56B7C LEA ECX,DWORD PTR DS:[EDI+ECX+E8C7B756]
0041FF3B | . C1C1 0C     ROL ECX,0C

```

Figure 6 : Constante repérée par Kanal

Kanal a donc détecté la présence d'un hash md5 grâce à une des constantes utilisées au cours des itérations. Nous retrouvons la constante présentée comme ceci dans une librairie md5 en asm :

```
FF    dtd,dta,dtb,dtc,dword ptr [edi+01*4],12,0e8c7b756h
```

Nous remarquons dans la suite du code l'utilisation de nombreuses autres constantes, nous vérifierons par contre plus tard pour celles-ci l'éventualité d'une modification. Remontons dans le code. Arrivé au PUSH EBP en 0041FEBO nous voyons ceci : Local calls from 004205E5, 004205FE. Allons au premier CALL en 004205E5, et répétez cette opération jusqu'à arriver au CALL en 0042078A. Nous apercevons ceci juste au dessus (Fig. 7) :

```

00420763 | . 8D4D 90      LEA ECX,DWORD PTR SS:[EBP-70]
00420766 | . C745 90 0123 MOV DWORD PTR SS:[EBP-70],67452301
0042076D | . C745 94 89AB MOV DWORD PTR SS:[EBP-6C],EFCDA889
00420774 | . C745 98 FEDC MOV DWORD PTR SS:[EBP-68],98BADCFE
0042077B | . C745 9C 7654 MOV DWORD PTR SS:[EBP-64],10325476
00420782 | . E8 09FEFFFF CALL WJChess3.00420590 WJChess3.00420590

```

Figure 7 : Constantes d'initialisation md5

Ce sont les constantes d'initialisation de la fonction md5. En général ces constantes sont les plus modifiées en cas de hashes customs, vérifions donc leur intégrité en les comparant avec la librairie asm :

```

mov    [esi].dtA,067452301h
mov    [esi].dtB,0efcdab89h
mov    [esi].dtC,098badcfefh
mov    [esi].dtD,010325476h

```

Nous ne remarquons aucune modification apportée à ces constantes, nous pouvons donc espérer être face à une librairie md5 non modifiée. Passons maintenant à la partie consacrée au keygenning, car nous disposons de l'endroit où est située la routine et de toutes les informations périphériques utiles.

### III) Keygenning de l'application

#### a) Analyse de la première routine

Il semblerait donc que nous étions proches, car en bas de la fenêtre nous apercevons l'appel à `MessageBoxA`. Mettons donc un bp sur ce push (Fig. 8) qui a l'air de marquer le début des choses intéressantes, et supprimons l'ancien :

```

0041FC7E > 6A 32      PUSH 32
0041FC80 . 68 98B77800 PUSH OFFSET <WJChess3.name>          ASCII "Horgh"
0041FC85 . 68 10B97800 PUSH OFFSET <WJChess3.name_buf2>
0041FC8A . E8 41B20100 CALL <WJChess3.copy_str>

```

Figure 8 : Breakpoint avant les routines

Comme vous pouvez le constater, les noms des variables poussées sur la pile / appelées ont été changés par des labels, afin de rendre la compréhension du code plus aisée. Le `PUSH 32` correspond à la taille du buffer poussé en premier paramètre (le `name_buf2` sur le screen), soit 50d ; le push suivant étant lui directement identifié comme poussant l'adresse du buffer contenant le nom. Le 3<sup>e</sup> `PUSH` est quant à lui pour le moment un buffer vide. Faisons click droit sur le `PUSH` -> *Follow in dump* -> *Immediate constant*, et gardons un œil sur le buffer. Rentrons dans le `call` avec `F7`. Traçons jusqu'au `REP MOVSDWORD PTR ES:[EDI],DWORD PTR DS:[ESI]` en `0043AF27`. Qu'apercevons-nous alors dans la fenêtre entre le CPU et le dump ?

`DS:[ESI]=[0078B798]=67726F48` ; valeur hexadécimale du nom "Horgh"  
`ES:[EDI]=[0078B910]=00000000` ; buffer où l'instruction va copier ce nom

Faisons *Follow address in dump* sur la deuxième valeur, afin de voir ce qu'elle va recevoir. On fait `F8`, et le nom est copié dans notre deuxième buffer. Le deuxième buffer poussé tout à l'heure en paramètre du `call` est donc le buffer de sortie, ou va être copié la string poussée en paramètre 2. Cette fonction ressemble fortement à `strcpy`, et j'ai l'impression que l'auteur a recodé certaines apis simples afin d'éviter la pose de breakpoint sur celles-ci, à la visibilité trop élevée. Nous arrivons ensuite sur le premier `CALL` suivi d'un saut conditionnel (Fig. 9) :

```

0041FC8A . E8 41B20100 CALL <WJChess3.copy_str>
0041FC8F . 68 CB777800 PUSH OFFSET <WJChess3.serial>
0041FC94 . E8 379B0100 CALL <WJChess3.calc_part2>
0041FC99 . 83C4 10     ADD ESP,10
0041FC9C . 84C0       TEST AL,AL
0041FC9E . 74 5F     JB SHORT <WJChess3.badboy>

```

Figure 9 : Première routine

Le buffer poussé en paramètre peut être facilement identifié, c'est en effet celui qui contient notre sérial entré. Allons jeter un coup d'œil à ce qui se passe dans ce `call`. Après un peu de tracing nous arrivons enfin à une chose intéressante (Fig. 10) :

```

00439800 . 8D71 01     LEA ESI,DWORD PTR DS:[ECX+1]
00439803 > 8A11       MOV DL,BYTE PTR DS:[ECX]
00439805 . 41         INC ECX
00439806 . 84D2       TEST DL,DL
00439808 . 75 F9     JNZ SHORT <WJChess3.len_serial>
0043980A . 2BCE     SUB ECX,ESI
0043980C . 83F9 18     CMP ECX,18
0043980F . 0F82 240100 JB <WJChess3.badboy>

```

Figure 10 : Première routine

Les routines qui vont par la suite implémenter le même procédé vont être nombreuses, en effet il s'agit d'un lstrlen recodé. On met dans DL le char du sérial, on incrémente le buffer pour passer au char suivant, on regarde si DL est égal à 0, si non c'est que nous n'avons pas atteint le null-byte de fin de chaîne, et donc nous recommençons. A la fin on soustrait l'adresse du buffer de départ à l'adresse du buffer d'arrivée, et la différence correspond au nombre de caractères contenus dans celui-ci. Nous remarquons donc que notre sérial doit faire au moins 24 caractères, car si le nombre est inférieur nous sautons sur le badboy. Recommençons donc, mais avec cette fois si un sérial de 24 caractères (je vous recommande de mettre des caractères différents, afin de mieux les repérer part la suite en cas de séparation).

Mettons pour exemple *ABCDEFGHIJKLMNO123456789*. Le saut n'est donc plus pris, et nous passons à la boucle suivante qui calcule le nombre de caractères du nom. Si le résultat est égal à 0, nous sautons sur le badboy. Continuons à tracer jusqu'au prochain CALL en 004398BC. Les paramètres passés à ce CALL sont des fragments du sérial (Fig. 11). Ils correspondent à :

- Arg3 = Partie 1 du sérial, 6 caractères
- Arg4 = Partie 2 du sérial, 5 caractères
- Arg5 = Partie 3 du sérial, 5 caractères

```

0012FE64  0012FE88  Arg1 = 0012FE88
0012FE68  00000002  Arg2 = 00000002
0012FE6C  0012FEA4  Arg3 = 0012FEA4 ASCII "ABCDEF"
0012FE70  0012FEB4  Arg4 = 0012FEB4 ASCII "HIJKL"
0012FE74  0012FEAC  Arg5 = 0012FEAC ASCII "N0123"

```

Figure 11 : Arguments passés au call

On remarque également que les lettres situées entre les parties 1 et 2, et 2 et 3 ont été supprimées. Il semblerait donc que les 7<sup>e</sup> et 13<sup>e</sup> caractères soient des tirets servant à séparer les différentes parties. Rentrions maintenant dans ce CALL avec F7. On trace, on tombe sur la boucle qui sert à calculer la taille des strings en 00439523 où le programme calcule la taille de la partie 1, puis sur le call qui remplace lstrcpy, qui va copier la partie 1. Vient ensuite encore la fonction de calcul de taille, qui calcule la taille du nom. Nous arrivons enfin à quelque chose d'intéressant avec le CALL suivant (Fig.12) :

```

00439547  . 2BC2          SUB EAX, EDX
00439549  . 50            PUSH EAX
0043954A  . 68 10B97800  PUSH OFFSET <WJChess3.copy_buf>
0043954F  . 8D4D D4      LEA ECX, DWORD PTR SS:[EBP-2C]
00439552  . E8 99B9FCFF  CALL <WJChess3.concat_str>
00439557  . 8B15 18A4750 MOV EDX, DWORD PTR DS:[75A418]

```

```

arg2 = partiserial
Arg1 = 0078B910 ASCII "Horgh"
WJChess3.00404EF0
WJChess3.004593E4

```

Figure 12 : Call de concaténation

Ce call est encore une fonction recodée : il s'agit de lstrcat. En effet, nous voyons que les chaînes passées en paramètre sont le nom et la première partie du sérial, en l'occurrence "ABCDEF". Le résultat nous apparaît dans les registres sous cette forme : "ABCDEFHorgh". C'est donc le nom qui est concaténé à la première partie du sérial. Voyons ce qui se passe par la suite. En traçant encore un petit peu nous retrouvons la boucle habituelle de calcul de taille en 00439567. Cette fois ci elle calcule la taille d'une chaîne fixe des datas, "Undo move". Cette chaîne est ensuite concaténée via la fonction que nous venons de découvrir à la string "ABCDEFHorgh", ce qui donne pour résultat "ABCDEFHorghUndo move". On arrive ensuite à un call intéressant en 00439567 (Fig. 13). En effet, c'est celui qui appelait les constantes de la fonction de hashage md5 tout à l'heure, et nous remarquons que cette fonction prend en paramètre la string qui vient d'être créée comme string à hasher.

```

0043957E  . 8D45 D4      LEA EAX, DWORD PTR SS:[EBP-2C]
00439581  . > 50          PUSH EAX
00439582  . E8 A971FEFF  CALL <WJChess3.md5>
00439587  . 83C4 04      ADD ESP, 4

```

```

arg1 = str a hasher
WJChess3.00420730

```

Figure 13 : call de la fonction de hashage md5

Voyons donc ce qui se passe après l'appel de la fonction (Je ne détaille pas la fonction md5, ne la maîtrisant pas suffisamment d'un point de vue formel et mathématique. Si vous cherchez à approfondir, allez voir sur wikipédia). Le résultat de la fonction se trouve dans EAX : "b31ca1520245605ae96479511d24da29". Vérifions maintenant si le hash est correct, afin de voir s'il n'a pas été modifié à un endroit que nous n'aurions pas remarqué. J'utilise personnellement le Keygenner Assistant d'at4re à cet effet, mais les tools sont nombreux. Inscrivez la chaîne "ABCDEFHorghUndo move", hashez la en md5, et comparez les résultats. Il s'avère que le hash est le même, et la probabilité de collision étant faible, on peut supposer que la fonction est celle d'origine, non modifiée. On passe juste après sur une autre boucle (Fig. 14) :

```

00439590 > 0FBE143E MOVSX EDX, BYTE PTR DS:[ESI+EDI]
00439594 . 52 PUSH EDX
00439595 . E8 6B3E0000 CALL <WJChess3.char_upper>
0043959A . 8B043E MOV BYTE PTR DS:[ESI+EDI], AL
0043959D . 46 INC ESI
0043959E . 83C4 04 ADD ESP, 4
004395A1 . 83FE 20 CMP ESI, 20
004395A4 . ^ 7C EA SHORT <WJChess3.continue_to_up_chars>

```

Figure 14 : Mise en majuscule du hash

Le résultat de cette boucle est "B31CA1520245605AE96479511D24DA29". Elle sert donc à mettre le hash en majuscules. Nous passons ensuite en 004395B5 sur la boucle de calcul de taille, ici du hash md5, taille qui est pushée en paramètre à la fonction lstrncpy recodée. Le CALL suivant n'a pas d'importance pour le moment, il s'agit de quelques vérifications (taille de la string, etc.). Continuons de tracer avec F8. Nous passons sur le CALL en 004395DD, dont j'avoue ne pas avoir saisi l'utilité, en tout cas dans notre optique de keygenning, nous repassons sur le call des messages d'erreur, et nous arrivons finalement sur le call en 00439637 (Fig.15) :

```

00439620 . 6A FF PUSH -1
0043962E . 53 PUSH EBX
0043962F . 50 PUSH EAX
00439630 . 8D4D B8 LEA ECX, DWORD PTR SS:[EBP-48]
00439633 . C645 FC 04 MOV BYTE PTR SS:[EBP-4], 4
00439637 . E8 C4B7FCFF CALL <WJChess3.create_real_value>
0043963C . C645 FC 02 MOV BYTE PTR SS:[EBP-4], 2

```

Figure 15 : Call où est générée la véritable partie

Rentrons dedans avec F7 afin de voir ce qui s'y trame. On trace avec F8, et après avoir passé les messages d'erreur et quelques sauts nous arrivons au CALL en 00404EBD. Rentrons également dedans. Après avoir encore tracé, nous arrivons ici (Fig. 16) :

```

0043B058 > 8A06 MOV AL, BYTE PTR DS:[ESI]
0043B05A . 8B07 MOV BYTE PTR DS:[EDI], AL
0043B05C . 8B45 08 MOV EAX, DWORD PTR SS:[EBP+8]

```

Figure 16 : Inscription de la bonne valeur

Que s'y passe-t-il ? Nous remarquons ceci pour la première ligne :

DS:[003D3D38]=42 ('B') AL=39 ('9')

Hors, en 003D3D38 nous retrouvons notre hash. L'instruction met donc dans AL la première lettre du hash md5. L'instruction suivante sauvegarde cette valeur dans la pile. Mettez un breakpoint en 0043B058 afin de voir quelles sont les valeurs récupérées. Ressortez du CALL, tracez, vous allez rebreaker sur le bp que l'on vient de mettre. Ce coup ci c'est le dernier octet du hash md5 qui est sauvegardé à une adresse de la pile. Gardons en tête les positions dans le hash et l'ordre dans lequel elles sont appelées, et continuons. Nous rebreakons encore une fois sur notre bp, ou la valeur est réinscrite à un autre endroit. Continuons ainsi de tracer en breakant sur notre bp, et notons les valeurs récupérées. Nous arrivons au final à ceci :

**B31CA1520245605AE96479511D24DA29** => les octets récupérés  
**135** **42** => la position de ces octets dans la partie calculée

Ce qui nous donne pour partie : "B9321". Continuons à tracer, encore et encore... Jusqu'à ce que nous ressortions du CALL en 004398BC dans lequel nous étions entrés au début. Reprenons l'analyse. La boucle en 0049B8D5 révérifie la taille de ma partie 2, et la taille est comparée à la taille de la partie générée au CMP EDI,ESI en 004398E0. Comme la taille est la même, nous ne sautons pas sur le badboy. Nous arrivons ensuite enfin à la vérification (Fig.17) :

```

004398F4 |> 51          PUSH ECX
004398F5 |. 804D E8    LEA ECX,DWORD PTR SS:[EBP-18]
004398F8 |. 51          PUSH ECX
004398F9 |. 50          PUSH EAX
004398FA |. E8 91E7FCFF CALL <WJChess3.cmp_part_serial>
004398FF |. 83C4 0C    ADD ESP,0C
00439902 |. 85C0      TEST EAX,EAX
00439904 |. 75 22     JNZ SHORT <WJChess3.badboy>

```

arg3 = len  
arg2 = fake part  
arg1 = true part  
WJChess3.00408090

Figure 17 : Comparaison des deux parties

Nous voyons ceci dans la pile :

```

0012FE6C 0012FE88 |Arg1 = 0012FE88 ASCII "B9321"
0012FE70 0012FEB4 |Arg2 = 0012FEB4 ASCII "HIJKL"

```

Nous remarquons donc bien ici que notre partie 2 du sérial est comparée à la partie générée précédemment (rappel : ABCDEF-HIJKL-N0123456789, pour le sérial entré à ce stade de l'analyse). Modifions les sauts suivants qui nous amènent au badboy, et continuons à tracer. Comme nous sommes passés sur le MOV AL,1, nous ne sauterons pas au badboy en 0041FC9E.

Nous venons donc de finir l'analyse de la première routine de calcul du sérial. Faisons le bilan de ce que nous avons appris jusque là.

- Le sérial aurait cette forme : xxxxxx-xxxxx-xxxxxxxxxxx (24 chars)
- Concatène la partie 1 du sérial au nom, puis à la string "Undo move", et md5 la chaîne.
- Récupère les octets aux positions 1-32-2-31-3 dans le hash md5 pour faire la partie 2.

## b) Analyse de la seconde routine

Passons maintenant à l'analyse de la 2<sup>e</sup> routine ci-dessous (Fig. 18) :

```

0041FCA0 |. 68 CCB77800 PUSH OFFSET <WJChess3.serial>
0041FCAS |. E8 B69C0100 CALL <WJChess3.calc_part4>
0041FCAA |. 83C4 04    ADD ESP,4
0041FCAD |. 84C0      TEST AL,AL
0041FCAF |. 74 4E     JZ SHORT <WJChess3.badboy>

```

Arg1 = 007887CC ASCII "ABCDEFGHJKLMN0123456789"  
WJChess3.00439960

Figure 18 : Deuxième routine

On retrouve le même schéma que tout à l'heure dans ce call. Il va tout d'abord calculer la taille du sérial, la comparer à 24 caractères, et ensuite calculer la taille du nom. Une fois ceci fait, nous nous retrouvons un call similaire à la routine 1 en 00439A72, voici ses paramètres (Fig. 19) :

```

0012FE68 0012FE8C |Arg1 = 0012FE8C
0012FE6C 00000004 |Arg2 = 00000004
0012FE70 0012FEA8 |Arg3 = 0012FEA8 ASCII "ABCDEF"
0012FE74 0012FEB8 |Arg4 = 0012FEB8 ASCII "HIJKL"
0012FE78 0012FEB0 |Arg5 = 0012FEB0 ASCII "N0123"

```

Figure 19 : Paramètres du call

- Arg3 = Partie 1 du sérial, 6 caractères
- Arg4 = Partie 2 du sérial, 5 caractères
- Arg5 = Partie 3 du sérial, 5 caractères

Revenons encore une fois dans le CALL avec F7. Encore une fois, nous tombons sur les mêmes fonctions : en 004390E5, la boucle calcule la taille de la partie 2, puis le call en 004390EE la copie dans un autre buffer. Ensuite en 00439105, il calcule la taille de la partie 3 du sérial. Nous arrivons enfin avec le CALL suivant à quelque chose d'intéressant : il s'agit du call identifié précédemment comme effectuant la même action que Istrcat (Fig. 20) :

```

00439109 . 50          PUSH EAX
0043910A . 56          PUSH ESI
0043910B . 804D 04     LEA ECX, DWORD PTR SS:[EBP-2C]
0043910E . E8 DBDFCFF CALL <WJChess3.concat_str>

```

Figure 20 : Call concaténant les parties 2 et 3

Ce call va concaténer la partie 3 du sérial du sérial à la partie 2, pour ce résultat : "HIJKLNO123". Cette nouvelle string va ensuite être passée au md5 du call suivant (Fig. 21) :

```

0043911F > 450        PUSH EAX
00439120 . E8 0B76FEFF CALL <WJChess3.md5>

```

Figure 21 : md5 de la string précédente

Le déroulement va ensuite être similaire en tout point à la routine 1. La boucle de 00439130 à 00439144 va mettre le hash en majuscules, le hash va ensuite passer à la routine de calcul de taille de 00439150 à 00439155. A partir de là commence désormais le calcul de la partie 4 du sérial, qui sera tirée du hash md5 des parties 2 et 3 concaténées. Nous n'avons pas enlevé notre breakpoint précédent en 0043B058 (Fig. 22) :

```

0043B058 > 8A06       MOV AL, BYTE PTR DS:[ESI]
0043B05A . 8B07       MOV BYTE PTR DS:[EDI], AL
0043B05C . 8B45 08     MOV EAX, DWORD PTR SS:[EBP+8]

```

Figure 22 : Inscription de la bonne valeur

En appliquant la même méthode que précédemment, c'est à dire en regardant la valeur tirée du dump, et sa position dans le hash md5, on arrive à cette conclusion :

00995F7DB587DF736E70A0EE322CB27C => les octets récupérés  
 3 125 4 => la position de ces octets dans la partie calculée

La véritable partie ainsi obtenue est donc "8772D". A la suite des calls récupérant chaque bonne valeur dans le hash, nous arrivons en 00439A92 ou il va comparer la taille des deux parties du sérial. Continuons encore une fois, jusqu'à arriver en 00439AAA (Fig. 23) :

```

00439AA1 > 8045 B4     LEA EAX, DWORD PTR SS:[EBP-4C]
00439AA4 . 51         PUSH ECX
00439AA5 . 8055 E8     LEA EDX, DWORD PTR SS:[EBP-18]
00439AA8 . 52         PUSH EDX
00439AA9 . 50         PUSH EAX
00439AAA . E8 E1E5FCFF CALL <WJChess3.cmp_part_serial>
00439AAF . 83C4 0C     ADD ESP, 0C

```

Figure 23 : Comparaison des deux parties

Ce call va donc comparer les 5 derniers caractères de notre sérial au 5 qui viennent d'être calculé. On s'aperçoit ainsi encore qu'un caractère entre la partie 3 (NO123) et la partie 4 (56789) est ignoré. Ce caractère va donc constituer l'ultime tiret, qui va lier les parties 3 et 4. Le résultat de la comparaison étant fatalement négatif, nous devons modifier les flags de façon à passer sur le MOV AL, 1. Ainsi, en 0041FCFAF nous ne prenons pas le JE vers le badboy.

Une fois passé le RETN nous nous retrouvons face à la vérification de la valeur inscrite dans EAX suite à la comparaison des vraie et fausse parties 4 du sérial. En inversant le Z-Flag, nous nous apercevons que le programme crée par la suite deux clés dans le registre pour y inscrire nos nom et sérial. Nous avons donc fini d'analyser le schéma de calcul de ce programme, récapitulons maintenant :

- Le sérial a cette forme : xxxxxx-xxxxx-xxxxx-xxxxx (24 chars)
- On concatène la partie 1 du sérial au nom, puis à la string "Undo move"
- On md5 la chaîne créée
- On met en majuscules le md5
- On récupère les octets aux positions 1-32-2-31-3 dans le hash md5 pour faire la partie 2
- On concatène la partie 2 et la partie 3
- On md5 la chaîne créée
- On met en majuscules le md5
- On récupère les octets à la position 11-12-7-30-13 dans le hash md5 pour faire la partie 4

Il faudra bien sûr générer des parties 1 & 3 aléatoires afin de pouvoir reproduire cette routine. Le keygen a été codé en ASM, *as usual*, et le fichier qui vous intéressera plus particulièrement est algo.asm. J'ai essayé de le commenter de façon à rendre ma démarche lisible.

#### 4) Conclusion et remerciements

Je tiens à remercier :

Shub-Nigurrath / ARTeam, pour le template Word

L'auteur du soft

Et surtout tous les gens que je connais & apprécie (ils se reconnaîtront)

#### 5) Références et documentation

<sup>1</sup> : <https://secure.wikimedia.org/wikipedia/fr/wiki/MD5>

Pour aller plus loin sur le keygenning de protections basées sur du md5 :

- MD5 Keygenning, *MISSING iN ByTES*, <http://tuts4you.com/download.php?view.2506>
- MD5 Keygenning (Part 1), *Encrypto*, <http://tuts4you.com/download.php?view.2235>
- MD5 Keygenning (Part 2), *Encrypto*, <http://tuts4you.com/download.php?view.2234>
- Crypto Keygen Tutorial – MD5, *Ziggy*, [http://rapidshare.com/files/36729002/ file\\_snd-ziggys\\_cryptotutorial\\_231\\_md5.zip](http://rapidshare.com/files/36729002/file_snd-ziggys_cryptotutorial_231_md5.zip)
- <http://xtx.free.fr/liens/tut/tut.html>, section Crypto / Hash

#### 6) Historique

- Version 1.0 first public release