

# PULSAR'S JC#1

A LA RECHERCHE DU CODE

PERDU



BY WAGANONO

# Table des matières

1) Intro

2) Première approche

3) A la recherche du code perdu

4) Suppression du junk

*A) Phase 1*

*B) Phase 2*

5) Résolution

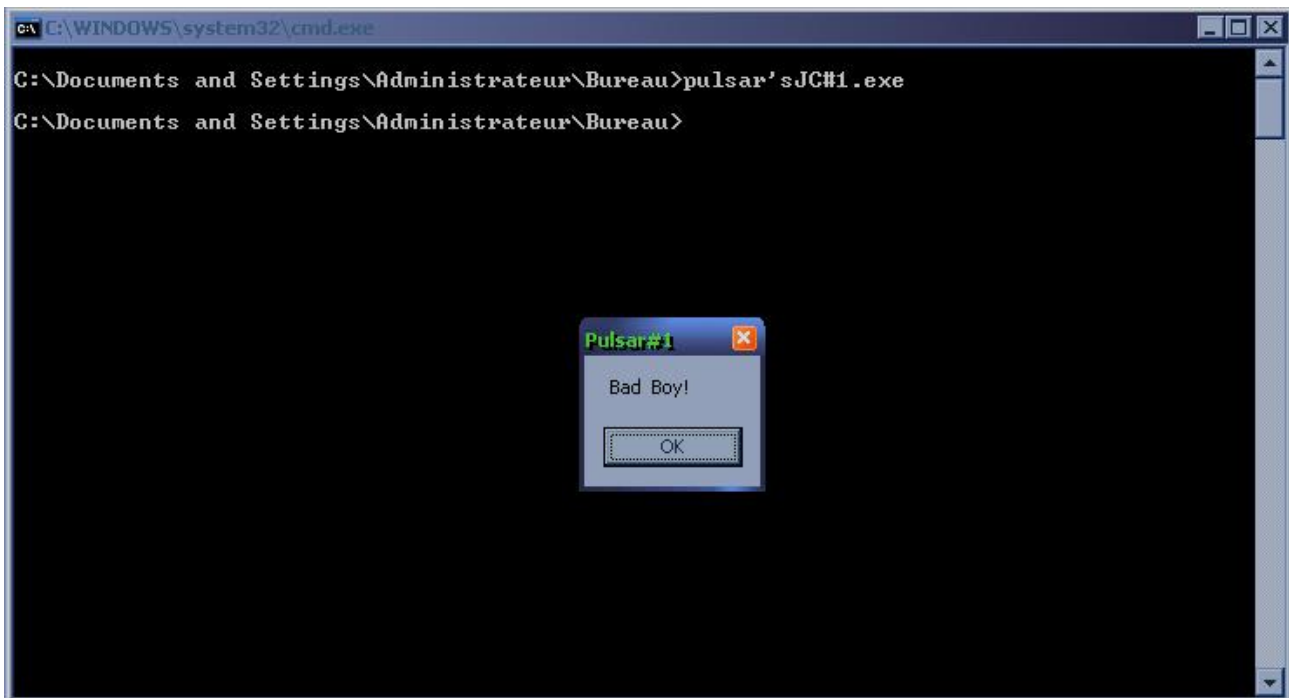
6) Outro

7) Greetz

# Intro

Le principal objectif de ce défi est la suppression du **junk code**. Nous avons affaire à une application en mode console très rudimentaire, c'est un bon début pour l'analyse.

Apparemment, il n'y a pas d'interactions avec l'utilisateur, tout doit donc sûrement se passer, soit en ligne de commande soit avec un keyfile peut être...



Dans un premier temps, nous allons essayer d'appréhender la bestiole et voir à quoi nous sommes confrontés.

# Première approche

Nous allons d'abord sortir l'artillerie classique, c'est à dire Ollydbg, et on remarque tout de suite ce méli-mélo :

00401000	\$ 68 994C4000	PUSH Pulsar's.00404C99	SE handler installation
00401005	. 64:FF35 00000	PUSH DWORD PTR FS:[0]	
0040100C	. 64:8925 00000	MOV DWORD PTR FS:[0],ESP	
00401013	. E8 9E3C0000	CALL <JMP.&kernel32.GetCommandLineA>	CGetCommandLineA
00401018	. 8BF8	MOV EDI,EAX	
0040101A	. B0 2E	MOV AL,2E	
0040101C	. F2:AE	REPNE SCAS BYTE PTR ES:[EDI]	
0040101E	. 83C7 04	ADD EDI,4	
00401021	. 8B1F	MOV EBX,DWORD PTR DS:[EDI]	
00401023	. 32CF	XOR CL,BH	
00401025	. C1CE BA	ROR ESI,0BA	Shift constant out of range 1..31
00401028	. 66:05 C607	ADD AX,7C6	
0040102C	. C1C7 86	ROL EDI,86	Shift constant out of range 1..31
0040102F	. 66:81C3 F1FE	ADD BX,0FEF1	
00401034	. 66:81EF 0BFF	SUB DI,0FF0B	
00401039	. 66:C1C6 84	ROL SI,84	Shift constant out of range 1..31
0040103D	. 02DA	ADD BL,DL	
0040103F	. 8AE8	MOV CH,AL	
00401041	. 80F6 DF	XOR DH,0DF	
00401044	. 32F7	XOR DH,BH	
00401046	. C1CD E8	ROR EBP,0E8	Shift constant out of range 1..31
00401049	. C0C0 72	ROL AL,72	Shift constant out of range 1..31
0040104C	. 66:03CB	ADD CX,BX	
0040104F	. C0C5 82	ROL CH,82	Shift constant out of range 1..31
00401052	. 8BF5	MOV ESI,EBP	
00401054	. 66:2BF3	SUB SI,BX	
00401057	. 8BE8	MOV EBP,EAX	
00401059	. B9 B3A0466A	MOV ECX,6A46A0B3	
0040105E	. C1C8 D8	ROR EAX,0D8	Shift constant out of range 1..31
00401061	. C0C5 1E	ROL CH,1E	
00401064	. 2ACD	SUB CL,CH	
00401066	. BD 91FD808B	MOV EBP,8B80FD91	
0040106B	. C1CA B8	ROR EDX,0B8	Shift constant out of range 1..31
0040106E	. C1CA 99	ROR EDX,99	Shift constant out of range 1..31
00401071	. 66:81C5 E914	ADD BP,14E9	
00401076	. C0CA EE	ROR DL,0EE	Shift constant out of range 1..31
00401079	. 66:81C7 B716	ADD DI,16B7	
0040107E	. 8BEA	MOV EBP,EDX	
00401080	. C0CA BB	ROR DL,0BB	Shift constant out of range 1..31
00401083	. 66:81F6 1F74	XOR SI,741F	
00401088	. 66:81EF 438A	SUB DI,8A43	
0040108D	. 66:C1C1 6C	ROL CX,6C	Shift constant out of range 1..31
00401091	. 80F3 78	XOR BL,78	
00401094	. C0CB 2C	ROR BL,2C	Shift constant out of range 1..31

Bon, ça commence fort et ceci jusqu'au «Good boy», ça fait quand même environ 404C5Dh – 401023h = **15418 octets de code!**

00404C17	. 81EF 418B59D6	SUB EDI,06598B41	
00404C1D	. 03C7	ADD EAX,EDI	
00404C1F	. 33CD	XOR ECX,EBP	
00404C21	. 05 36F5E3FD	ADD EAX,0DE3F536	
00404C26	. C1C9 C6	ROR ECX,0C6	Shift constant out of range 1..31
00404C29	. 66:81F1 7EED	XOR CX,0ED7E	
00404C2E	. C0C6 A2	ROL DH,0A2	Shift constant out of range 1..31
00404C31	. 2C F0	SUB AL,0F0	
00404C33	. 66:81F1 09D7	XOR CX,0D709	
00404C38	. 35 ACFF3773	XOR EAX,7337FFAC	
00404C3D	. 2ADE	SUB BL,DH	
00404C3F	. 81EB 84628AA1	SUB EBX,A18A6284	
00404C45	. C1CB 80	ROR EBX,80	Shift constant out of range 1..31
00404C48	. D0C0	ROL AL,1	
00404C4A	. 80C1 77	ADD CL,77	
00404C4D	. 66:C1CE 4F	ROR SI,4F	Shift constant out of range 1..31
00404C51	. 2AFA	SUB BH,DL	
00404C53	. 66:35 0BB1	XOR AX,0B10B	
00404C57	. 66:35 0BB1	XOR AX,0B10B	
00404C5B	. 02FA	ADD BH,DL	
00404C60	. 8B80 00300000	MOV EAX,DWORD PTR DS:[EAX+3000]	
00404C63	. 3105 09604000	XOR DWORD PTR DS:[406009],EAX	
00404C66	. 813D 09604000	CMP DWORD PTR DS:[406009],646F6F47	
00404C73	. 74 0A	JE SHORT Pulsar's.00404C7F	
00404C75	. C705 09604000	MOV DWORD PTR DS:[406009],20646142	
00404C7F	> 6A 00	PUSH 0	
00404C81	. 68 00604000	PUSH Pulsar's.00406000	
00404C86	. 68 09604000	PUSH Pulsar's.00406009	
00404C8B	. 2A 00	PUSH 0	
00404C8D	. E8 2A000000	CALL <JMP.&user32.MessageBoxA>	
00404C92	. 6A 00	PUSH 0	
00404C94	. E8 17000000	CALL <JMP.&kernel32.ExitProcess>	

```

[Style = MB_OK!MB_APPLMODAL
Title = "Pulsar#1"
Text = "Bad Boy!"
hOwner = NULL
MessageBoxA
ExitCode = 0
ExitProcess

```

On peut remarquer que le serial est passé en argument au crackme, d'où l'appel à la fonction GetCommandLineA.

On peut aussi noter l'installation d'un gestionnaire d'exception dès les premières lignes de code, donc une exception est susceptible de se produire dans tout ce flot d'instructions, en déroulant **sauvagement** le code, nous tombons sur ceci :

004028CA	. 02FD	ADD BH,CH	
004028CC	. C0C0 33	ROL AL,33	Shift constant out of range 1..31
004028CF	. C0C0 2B	ROL AL,2B	Shift constant out of range 1..31
004028D2	. 81F2 F737A471	XOR EDX,71A437F7	
004028D8	. 66:03C2	ADD AX,DX	
004028DB	. 80F7 5B	XOR BH,5B	
004028DE	. 8B80 00300000	MOV EAX,DWORD PTR DS:[EAX+3000]	
004028E4	. A3 09604000	MOV DWORD PTR DS:[406009],EAX	
004028E9	. 83C7 04	ADD EDI,4	
004028EC	. 8B0F	MOV ECX,DWORD PTR DS:[EDI]	
004028EE	. 66:81E9 08A5	SUB CX,0A508	
004028F3	. 66:C1C9 05	ROR CX,5	
004028F7	. 80EA F9	SUB DL,0F9	
004028FA	. BF 21E366C2	MOV EDI,C266E321	

C'est le seul endroit où l'exception est susceptible de se produire, toutes les autres instructions étant de simples opérations comme SHL, ROL, ADD, SUB,... (mais pas DIV!) sur les registres. On peut le vérifier en passant un serial bidon en argument au crackme.

Si l'exception est levée, le code du handler est exécuté :

00404C92	. 55	PUSH EBP	Structured exception handler
00404C9A	. 8BEC	MOV EBP,ESP	
00404C9C	. 8B45 10	MOV EAX,DWORD PTR SS:[EBP+10]	
00404C9F	. C780 B8000000	MOV DWORD PTR DS:[EAX+B8],Pulsar's.00404C69	
00404CA9	. B8 00000000	MOV EAX,0	
00404CAE	. C9	LEAVE	
00404CAF	. C3	RETN	

Le handler place le registre EIP à la fin du code, juste avant la vérification BadBoy/GoodBoy. A ce stade, la vérification est toujours fausse, car nous n'avons rien écrit en [eax+3000] étant donné que l'adresse à laquelle on tente d'accéder n'est pas mappée par le système.

Une fois ceci constaté, il ne reste plus qu'à comprendre cette histoire de junk pour en savoir plus, c'est ce que nous allons voir tout de suite...

# A la recherche du code perdu

Une des meilleurs façons d'appréhender du junk code est de **comprendre comment il a été conçu** ou comment il aurait pu être conçu. Nous devons donc passer par une phase d'observation dans un premier temps.

On pourrait supposer que le junk est construit à base de **macros** mais cela semble faux, il n'y aucun motif (de code) qui semble réapparaître à plusieurs reprises, à moins que toutes les macros soient différentes mais ça serait sans intérêts pour un crackme.

Le junk provient donc sûrement d'un «tool maison», qui construit du junk aléatoire et noie le «bon» code dans un océan de déchets que je ne peux supporter...

Après un ou deux **cappucino**, on semble reconnaître un format classique de junk consistant à cumuler des opérations inverses sur les registres, c'est à dire des opérations nulles comme:

004010B5	. 02E9	ADD CH,CL	
004010B7	. C0C7 AA	ROL BH,0AA	Shift constant out of range 1..31
004010BA	. 66:33FD	XOR DI,BP	
004010BD	. 66:33FD	XOR DI,BP	
004010C0	. C0CF AA	ROR BH,0AA	Shift constant out of range 1..31
004010C3	. 2AE9	SUB CH,CL	

Ce genre de junk n'est pas très difficile à éliminer, le problème est que des instructions valides peuvent se mêler entre ces instructions et le junk n'est alors plus «symétrique», ça se complique un peu. De plus le junk peut, ne pas être symétrique à la base, dans l'exemple ci-dessus on pourrait inverser les 2 dernières lignes par exemple et le code conserverait le même comportement.

Mais on peut venir à bout de ce genre de junk sans trop de difficultés, sortons maintenant l'artillerie lourde, **je déclare la guerre au junk!**





# Suppression du junk

## A) Phase 1

On va donc sortir IDA et coder un petit **script IDC** pour enlever le type de junk que nous avons évoqué juste avant. Afin de suivre plus en direct l'élimination du code, nous allons d'abord faire un petit script qui permet de compter le nombre d'instructions entre les offsets 401023h et 404C5Dh :

```
#define START_ADDRESS 0x401023
#define END_ADDRESS 0x404C5D

static instruction_count(){
    auto from;
    auto count;

    count = 0;
    from = START_ADDRESS;
    while(from<END_ADDRESS){
        from = from + ItemSize(from);
        count++;
    }

    return count;
}
```

Nous obtenons **4550 instructions**, bref, il y a du boulot! Nous allons donc essayer de NOPer le junk du type :

```
XXX
YYY
REAL CODE
ZZZ
ZZZ inv
XXX inv
REAL CODE
REAL CODE
YYY inv
```

Ce schéma n'est pas exhaustif mais représente bien les différentes formes que peut prendre le code. Il faudrait donc coder un script qui prenne en charge les différents cas possibles et ce n'est pas si difficile comme on pourrait le penser.

Voici un script qui s'en occupe (le code n'est pas complet mais le principe est là) :

```
static HideJunkType1() {
    auto from, from2, code1, code2;
    auto count, len;

    count = 0;
    len = 0;
    from = START_ADDRESS;

    while(from < END_ADDRESS) {
        code1 = GetDisasm(from);
        from2 = from + ItemSize(from);

        while(count < 128) {
            code2 = GetDisasm(from2);
            if(opposite(code1, code2)) {
                NOPPage(from);
                NOPPage(from2);
                len = len + 2;
                break;
            }
            count = count + 1;
            from2 = from2 + ItemSize(from2);
        }

        count = 0;
        from = from + ItemSize(from);
    }

    return len;
}
```

NOPPage = fonction qui patche une instruction en NOP

opposite = prédicat testant si deux lignes de codes s'annulent

Le principe est le suivant, pour chaque instruction, on cherche son opposé dans les 128 lignes suivantes, j'ai choisi 128 pour éviter de ralentir l'exécution du script mais on peut essayer avec d'autres valeurs, ça marche. Si on trouve un opposé alors on NOP les 2 instructions, ce n'est pas plus compliqué que ça!

Noter que la fonction renvoie le nombre d'instructions noppées, si on exécute le script on constate d'abord à quel point il est lent et que **3248 instructions** ont été **éliminées**!

C'est très bien parti, faisons un copier/coller du code dans un éditeur de texte, supprimons les NOP jusqu'au moment où l'exception est levée. Nous allons dans un premier temps étudier cette partie.

Cette première partie comporte **588 lignes de code**, il y a donc sûrement encore du junk qui reste incrusté!



## B) Phase 2

Quel type de junk a bien pu être utilisé ici? Pas des opérations inverses, nous venons de les enlever... peut être des calculs incohérents sur des **registres non utilisés par le code original**?

Nous savons qu'avant que le junk commence, les registres sont initialisés de la façon suivante :

- EBX = 1er DWORD de la chaîne passée en argument
- EDI = pointeur vers l'argument

Au moment où l'exception est levée c'est EAX qui contient l'adresse incriminée, EAX dépend sûrement d'un calcul à partir de notre serial or EBX en contient une partie! Essayons alors de construire un script éliminant toutes les opérations ne concernant pas EAX et EBX :

```
static HideJunkType2() {
    auto from, count, operands;
    count = 0;
    from = START_ADDRESS;

    while(from < 0x4028DE) {
        operands = substr(GetDisasm(from), 3, -1);
        if(not_used(operands)) {
            NOPage(from);
            count = count + 1;
        }
        from = from + ItemSize(from);
    }

    return count;
}
```

not\_used = prédicat testant si une ligne de code n'utilise pas EAX et EBX

L'exécution du script nous informe que 572 instructions ont été mises de côté. Il reste donc que **16 lignes de code**, que voici :

```
mov     ebx, [edi]           // edi = pointeur argument 1er DWORD
mov     ah, bl
mov     al, bh
sub     al, 0F5h
xor     ah, 75h
rol     eax, 9
rol     eax, 7
ror     ebx, 5
ror     ebx, 0Bh
sub     ah, ah
xor     al, al
add     al, bh
xor     al, 64h
xor     ah, bl
add     ah, 0FDh
mov     eax, [eax+3000h]     // Si eax+3000 mauvais adresse ==> exception
```

Vous pouvez recompiler ce code et constater qu'il a exactement le même comportement que dans le crackme original.

Supposons que notre première partie soit valide, c'est à dire que l'exception n'est pas levée et que (EAX+3000h) pointe vers une adresse valide. Nous poursuivons alors l'exécution logique du code.

Oui mais le code est encore junké là aussi et sûrement pas de la même façon car cette fois-ci, l'état initial des registres est le suivant :

- ECX = 2 ème DWORD de l'argument
- EBX = inchangé depuis la routine précédente
- EDI = pointeur sur l'argument

A la fin de la deuxième partie du code, (EAX+3000h) doit pointer vers une adresse valide. Pour enlever le junk, nous pouvons alors procéder de la même façon que précédemment sauf que cette fois ci nous devons conserver les opérations s'effectuant sur les registres EAX,EBX et ECX. Vous trouverez ce script en annexe, il est identique au précédent à 2 ou 3 détails près.

Une fois cette 2ème partie déjunkée de 694 instructions, il reste le code suivant :

```
mov     ecx, [edi]                // edi = pointeur argument 2nd DWORD
mov     ah, cl
mov     al, ch
ror     ecx, 5
ror     ecx, 0Bh
mov     bh, cl
sub     ah, 75h
mov     bl, ch
xor     bl, 79h
add     al, 0Fh
rol     eax, 10h
sub     bh, 1
sub     ecx, ecx
add     ecx, eax
xor     cx, cx
add     cx, bx
mov     eax, ecx
mov     eax, [eax+3000h]          // EAX+3000 = adresse valide (sinon exception)
xor     dword_406009, eax
cmp     dword_406009, 646F6F47h   // Check bad boy ?
jz      short loc_404C7F
mov     dword_406009, 20646142h   // good boy here :)
```

Certes, mais quelle doit être la valeur de EAX au final avant les accès mémoires de la partie 1 et 2?

# Résolution

Pulsar a poussé un peu plus loin le vice, remarquez le dump de la section .data du crackme :

Address	Hex dump	ASCII
00406000	50 75 6C 73 61 72 23 31 00 42 61 64 20 20 42 6F	Pulsar#1.Bad Bo
00406010	79 21 00 73 6F 6D 65 20 6D 69 73 63 20 74 65 78	yf.some misc tex
00406020	74 00 63 24 AC C3 72 65 61 6C 6C 79 20 6E 6F 74	t.c\$%treally not
00406030	68 69 6E 67 20 6F 66 20 69 6E 74 65 72 65 73 74	hing of interest
00406040	20 68 65 72 65 00 24 4B C3 A7 47 72 65 65 74 7A	here.\$K!0Greetz
00406050	3A 20 59 6F 6C 65 6A 65 64 69 2C 4B 68 61 72 6E	: Yolejedi,Kharn
00406060	65 74 68 2C 45 6C 6F 6F 6F 2C 4E 69 63 6F 2C 45	eth,Elooo,Nico,E
00406070	6C 69 63 5A 2C 2B 53 70 61 74 68 2C 4B 61 69 6E	lic2,+Spath,Kain
00406080	65 2C 4E 65 69 74 73 61 2C 47 62 69 6C 6C 6F 75	e,Neitsa,Gbillou
00406090	2C 6D 6F 72 65 20 77 69 6C 6C 20 63 6F 6D 65 20	,more will come
004060A0	6C 61 74 65 72 20 5E 5E 00 00 00 00 00 00 00	later ^^.....

Vous ne remarquez rien de bizarre?

2 DWORDs sont camouflés : 0xC3AC2463 et 0xA7C34B24.

Rappelez vous aussi, le code juste avant le check :

```
MOV EAX,DWORD PTR DS:[EAX+3000]  
XOR DWORD PTR DS:[406009],EAX  
CMP DWORD PTR DS:[406009],646F6F47 // 646f6f47 = «Good»
```

Or  $0xC3AC2463 \text{ XOR } 0xA7C34B24 == 0x646F6F47 == \text{«Good»}$ .

Donc nous devons avoir :

- $EAX + 3000h = 0x00403022$  dans la partie 1
- $EAX + 3000h = 0x00403046$  dans la partie 2

Désormais, il reste à inverser les deux petites routines et nous aurons le code qui comporte donc 8 lettres (Si avec  $i=0\dots7$ ), c'est quasiment immédiat :

## Partie 1:

$S0 = 75h \text{ XOR } 00 = 75h$   
 $S1 = F5h + 40h = 35h \text{ (mod } 100h)$   
 $S2 = 30h - FDh = 33h \text{ (mod } 100h)$   
 $S3 = 64H \text{ XOR } 22h = 46h$

## Partie 2:

$S4 = 75h + 00 = 75h$   
 $S5 = 40h - 0Fh = 31h$   
 $S6 = 30h + 1 = 31h$   
 $S7 = 46H \text{ XOR } 79h = 3Fh$

Au final, nous obtenons : **u53Fu11?**

Notez qu'il faut lancer le crackme avec l'extension .exe pour que tout fonctionne, c'est à dire «Pulsar'sJC#1.exe u53Fu11?». (bug ?)

## Outro

Ce fut vraiment un **plaisir** de résoudre ce crackme bourré de petites subtilités. On peut remarquer à quel point le junk est agaçant et nous ralentit, mais des outils de plus en plus puissant comme IDA peuvent nous faciliter la vie.

Résoudre le crackme sans l'utilisation d'un script ou autre petit utilitaire ad-hoc est une mission impossible (**99%** du code est junké), c'est ce qui fait son charme.

## Greetz

Merci à **Pulsar** pour avoir réalisé ce crackme original et très enrichissant. Il n'y a aucune raison de le laisser sans solutions, ce serait un **crime**... :)

Je tiens aussi à **remercier** tous les membres de FC, FRET, Guett@, BUBlic!, Virtualabs, haiklr, baboon, BeatriX, Kharneth, rAsm et vous.

**WOKANONO**  
**BOUL 2007**

wokanono@gmail.com  
<http://melzas.free.fr>