Advanced Registry Tracer 1.0

Outils

SoftIce
R!SC's Process Patcher v1.2i

Méthode

Live Approach

Avant Propos

Advanced Registry Tracer est un petit outil particulièrement utile à notre cause. Grâce à lui vous pourrez faire une " photo " de votre base de registre avant le lancer pour la première fois un programme/cible, et comparer cette photo avec une ou plusieurs autres que vous pourriez prendre par la suite . Il peut se révéler tout particulièrement intéressant lors d'un combat acharné contre une Time Limit.
Sans la demande de Psyché, je n'aurais jamais rien écrit sur ce programme. Vous allez comprendre pourquoi…

Un peu d'Histoire :

Advanced Registry Tracer (ART) v1.0a est le Phénix de Regfix 0.95.

Que fait il ?
* Undo and Redo files in *.reg and *.inf format
* Unlimited quantity of snapshots (copie of Registry) in one file
* Optimised usage of memory by limiting the number of snapshots to be loaded at a time
* Allowing to add comments to snapshots
* Improved presentation of search results

Vous ne comprenez pas l'anglais ?
Dommage…
Mais comme moi non plus…

Ce programme, en lui même facile à " modifier ", présente une petite particularité : il a été passé deux fois dans le compresseur ASPack (on retrouve deux signatures de cette protection dans l'exécutable).
ASPack se présente…
Mais je laisserai plutôt Psyché se charger de vous parler d'ASPack en détail.

HOW TO

Le Soft, à son lancement, commence par afficher un écran shareware à trois boutons :[I Agree], [Quit], et [Register]. Quel manque de courtoisie…
Il n'empêche que ces écrans Sharewares sont bien pratiques pour rechercher un Flag Utilisateur Reg/Unreg, tout comme les boites de message " Code Invalide " : ils sont faciles à trouver, et donnent un point d'entrée dans le schéma de protection.

Dans le cas de ART, SoftIce ne faisant pas le plus petit effort pour apparaître, malgré l'usage de son Symbol Loader , les choses se compliquent un peu.
Comment réussir à provoquer l'apparition de ce si bel écran noir (dont la couleur est configurable, au passage), avant que l'écran Shareware ne s'affiche ?

Il y a bien les API habituelles (MessageBox, DialogBoxParamA), mais ni l'une, ni l'autre ne perturbent ART.

Il y aurait bien la solution de rechercher avec ProcDump le point d'entrée du programme est d'y coller un petit CC/BPINT 3 pour forcer la main à Sice, mais nous allons faire plus rapide, bien que moins élégant :

CTRL-D > pour faire apparaître SoftIce manuellement
BPX GetProcAddress > une API très utilisée en début d'application
F5 > pour revenir à l'explorateur Windows
GO > traduisez : on lance ART

POP ! > Break
A partir de maintenant, le labeur commence : en traçant avec [F10], vous allez cherchez où se cache le call qui provoque l'apparition de l'écran Shareware.
Attention ! Ca va être long…
Il faut qu'ASPack décompresse le Loader, décompresse une première fois le programme d'origine, puis une seconde fois, et enfin vous serez dans l'application telle qu'elle a été écrite avant d'être Bi-ASPackée.

```
0177:00480C7F  E82CFFFFFF          CALL      00480BB0
0177:00480C84  84C0                TEST      AL,AL           > condition
0177:00480C86  750D                JNZ       00480C95        > branchement
0177:00480C88  A1DCD34B00          MOV       EAX,[004BD3DC]
0177:00480C8D  8B10                MOV       EDX,[EAX]
0177:00480C8F  FF92CC000000        CALL      [EDX+000000CC]  > nag screen
```

En 00480C8F, vous ferrez apparaître l'écran shareware. Juste au dessus, vous avez un branchement conditionnel. Posez un BPX dessus (F9), puis quittez SoftIce (F5), fermez l'application (Quit), et relancez-la (Clic).

POP !

Normal, Quand même, Hey !
Inversez le Zéro Flag (R FL Z), et relancez le Soft (F5).
Le programme démarrera sans la moindre anicroche…
Et surtout sans le nag Screen !

Histoire de vérifier deux ou trois petites choses, avancez votre horloge, et recommencez tout le process
Pas de Time Limit !

Il semblerait bien que ce Nag Sceen soit la seul " limitation " Shareware. Du coup il va être amusant de s'en débarrasser…

Le branchement en 00480C86 est sous condition de AL. Allons tracer du coté du call 00480BB0…
Pour commencer, il va falloir poser un point d'arrêt sur celui ci (F9), et une fois de plus reprendre le Process au début.

POP !
On s'y attendait…

F8 pour entrer DANS le call, puis F10 pour partir à la recherche de manipulations sur AL, et tout particulièrement en sortie de routine :

```
0177:00480BA6  8A45FF              MOV       AL,[EBP-01]
0177:00480BA9  5F                  POP       EDI
0177:00480BAA  5E                  POP       ESI
0177:00480BAB  5B                  POP       EBX
0177:00480BAC  59                  POP       ECX
0177:00480BAD  59                  POP       ECX
0177:00480BAE  5D                  POP       EBP
0177:00480BAF  C3                  RET
```

Faites un " ? al " et vous obtiendrez en retour la valeur 0.
Vous allez voir qu'à tous les coups, en mettant 1 dans AL (EAX est peut être plus tendancieux, avec sa valeur 0071FA00…), tout va bien se passer.

Et oui !
Quel court de crack, n'est ce pas…

On dispose donc de trois octets (8A45FF MOV AL,[EBP-01]) pour pouvoir nous exprimer. Qu'allons nous faire d'autant d'espaces ?

```
0177:00480BA6  33C0          XOR EAX, EAX   > force eax = 0
0177:00480BA8  40            INC EAX        > force eax = 1
0177:00480BA9  5F            POP EDI        > on continue comme si de rien n'était…
```

Et voilà !

Reste quand même à trouver une solution pour patcher le programme, et avec la double compression ASPack, les choses risquent de se corser un peu…

A chacun de faire à hauteur de ses capacités.
Pour moi : R!SC's Process Patcher v1.2i
Pour Psyché : Le VRAI patch du programme.

R!SC's Process Patcher v1.2i

R!SC's Process Patcher v1.2 : http://mercury.spaceports.com/~quel/protools/patchers.htm

C'est un utilitaire génial dont voici la description :

Its a process patcher thingy, creates a win32.exe from a simple script, which will then load a process, ans wait for it to unpack/deprotect itself, then patch the memory to fix any bugs that the autor left in the programm. (comme des nag screen, des times limit, ce genre d'erreurs…)

Bref, R!SC est un memory patcher qui utilise un script pour créer un fichier exécutable de 8 ko, et qui permettra de lancer ART sans le modifier " physiquement " (et donc sans avoir à s'inquiéter d'ASPack), mais qui interviendra quand le programme sera décompressé en mémoire.
Voici le script qu'il serait possible d'écrire :

```
T=2000:                          ; temps accordé pour trouver les bytes à modifier
F=art.exe:                       ; nom du programme à patcher
O=loader_art.exe:                ; nom du fichier exe que vous allez créer
P=480BA6/8A,45,FF/33,C0,40:      ; modifications à apporter en 0000480BA6
$                                ; ordre de fin de script
```

A l'aide de ce script, R!SC va générer un Loader qui, une fois lancé, va faire démarrer l'application / cible exactement comme si vous aviez modifié physiquement l'exécutable

Gain de place et efficacité garantie !

Fin de la rubrique " CRACK pour NEWBIES "
Que les VRAIES festivités commencent…

Christal

Advanced Registry Tracer 1.0a
ou comment faire mumuse avec du code automodifiant
par Psyché

1. Introduction

Tout d'abord je voudrais remercier Christal d'avoir bien voulu rédiger la première partie de ce tut.
Contrairement à ce qu'il semble manifester implicitement, ce n'est pas vraiment du " cracking for newbies " ce qu'il a fait … je ne suis pas sûr qu'un newbie y serait arrivé d'ailleurs. Tomber sur un soft qui ne se désassemble pas quand on débute, il n'y a rien de tel pour se décourager et finalement laisser tomber. En outre, il fallait connaître R !SC pour s'en sortir et réussir à patcher ce soft.
Bref, Christal arrête de te dénigrer ! Si je t'ai demandé de faire ce tut avec moi c'est parce que je t'accorde une valeur certaine ! ☺ Bon, excusez-nous … vous savez ce que c'est les vieux couples ;) (non je déconne, hein, n'allez pas penser que … c'est pas vrai, c'est de l'humour … mais SI, puisque je vous le dit … 'tain y sont lourds ☺).

2. ASPAck

Je ne vais pas revenir sur ASPack en général (j'ai fait un autre texte à ce sujet) mais plutôt sur la situation de ART.
Pour tout vous dire, je n'ai même pas vu qu'il s'agissait d'ASPack car le code généré en multipasse ne reprend rien de celui obtenu en monopasse. En plus, dans la compression simple, le loader se trouve dans une section nommée " .adata ", inexistante ici. Bon, mais si on dit que c'est ASPack, je veux bien, ça n'a pas beaucoup d'importance, en tout cas c'est un compresseur commercial (en opposition au compresseur " maison ") car comme je l'ai mis ailleurs (le tut sur Remote Selector, je crois) il est " général ", c-à-d qu'il utilise des adresses relatives aux sections où se trouve le code (ex. mov [EBP+ 445255], eax) et pas des adresses fixes (ex. mov [425688], eax).

Un autre détail que je voulais signaler c'est que ART n'est pas passé 2 fois au compresseur mais 3 fois … mais ça change pas grand chose non plus.

Dernière chose, si vous voulez un autre genre d'approche, référez-vous au tut de Brénuche sur AZPR (qui est fait par la même boîte que ART) qui est assez bien torché ;)

3. On attaque

Mettons-nous d'accord sur l'objectif qui sera le même que celui de Christal … modifier l'adresse 480BA6 en remplaçant 8A 45 FF par 33 C0 40.

Il faut aussi que je vous dise qu'il est fondamental , quand on se trouve en face d'un soft compressé, de l'aborder dès son Entry Point car les festivités commencent d'emblée et elles n'attendent pas d'arriver à un GetProcAdress ou tout au moins, vous risquez fort de passer à côté de l'aspect général.

J’arrive ici à un autre point essentiel dont j’ai déjà discuté plusieurs fois avec Christal. Ce qui importe c’est de comprendre “ pourquoi ” plutôt que de comprendre “ comment ”. Ceci vaut aussi bien pour un test de flag (d’où vient-il et pourquoi fait-on ça à cet endroit-là ?) que pour un soft compressé (un dongle aussi d’ailleurs mais là c’est vraiment une autre histoire ☺).

Bien … l’Entry Point (= EP) disais-je ?
Pourquoi l’Entry Point (en fait le raw offset de l’Entry Point, c-à-d l’offset de l’EP dans le fichier, sur le disque) ? Pour la raison signalée par Christal plus haut … je vais obliger le programme à s’arrêter à l’Entry Point en y plaçant une interruption 3 (INT3) qui n’est autre que l’instruction du breakpoint.
Vous avez deux manières de l’obtenir … la sportive et celle du fainéant, je vais vous les expliquer toutes les 2, à vous de choisir :

1°. *Le fainéant*

Vous vous êtes sûrement déjà tous énervés sur ces enfoirés de softs qui ne se désassemblent pas, non ? En pensant qu’il y a de l’anti Wdasm dans l’air et tout le toutim.
Mouais. A vrai dire, il y a assez peu de softs vraiment impossible à désassembler. Les puristes vous diront que IDA passe au-dessus de ces limitations, mais moyennant une petite manipulation, notre bon vieux Wdasm fera parfaitement l’affaire.

Ici revient la question … “ pourquoi ” ça ne se désassemble pas ?
Hé hé, tout bêtement parce que le programme ne mentionne pas qu’il contient du code exécutable et à partir de là, Wdasm baisse les bras (ce que ne fait pas IDA, lui avec un Entry Point il va partout).
Il nous suffira donc d’obliger le programme à déclarer sa partie exécutable ☺

La déclaration dont je parlais est liée aux attributs de la section (.CODE ou .TEXT ou n’importe quelle autre section d’ailleurs, on peut avoir du code n’importe où) et ces attributs sont contenus dans l’en-tête (header) du programme.
Pour information, voici un récapitulatif des différentes valeurs des attributs (ils sont bien sûr cumulables par simple addition).

```
// Section characteristics.

#define IMAGE_SCN_TYPE_NO_PAD               0x00000008 //   Reserved.
#define IMAGE_SCN_CNT_CODE                  0x00000020 //   Section contains code.
#define IMAGE_SCN_CNT_INITIALIZED_DATA      0x00000040 //   Section contains initialized data.
#define IMAGE_SCN_CNT_UNINITIALIZED_DATA    0x00000080 //   Section contains uninitialized
                                                           data.
#define IMAGE_SCN_LNK_OTHER                 0x00000100 //   Reserved.
#define IMAGE_SCN_LNK_INFO                  0x00000200 //   Section contains comments or some
                                                           other type of information.
#define IMAGE_SCN_LNK_REMOVE                0x00000800 //   Section contents will not become
                                                           part of image.
#define IMAGE_SCN_LNK_COMDAT                0x00001000 //   Section contents comdat.
#define IMAGE_SCN_NO_DEFER_SPEC_EXC         0x00004000 //   Reset speculative exceptions
                                                           handling bits in the TLB entries
                                                           for this section.
#define IMAGE_SCN_GPREL                     0x00008000 //   Section content can be accessed
                                                           relative to GP
#define IMAGE_SCN_MEM_FARDATA               0x00008000
#define IMAGE_SCN_MEM_PURGEABLE             0x00020000
#define IMAGE_SCN_MEM_16BIT                 0x00020000
#define IMAGE_SCN_MEM_LOCKED                0x00040000
```

```
#define IMAGE_SCN_MEM_PRELOAD                0x00080000
#define IMAGE_SCN_ALIGN_1BYTES               0x00100000
#define IMAGE_SCN_ALIGN_2BYTES               0x00200000
#define IMAGE_SCN_ALIGN_4BYTES               0x00300000
#define IMAGE_SCN_ALIGN_8BYTES               0x00400000
#define IMAGE_SCN_ALIGN_16BYTES              0x00500000 //   Default alignment if no others are
specified.
#define IMAGE_SCN_ALIGN_32BYTES              0x00600000
#define IMAGE_SCN_ALIGN_64BYTES              0x00700000
#define IMAGE_SCN_ALIGN_128BYTES             0x00800000
#define IMAGE_SCN_ALIGN_256BYTES             0x00900000
#define IMAGE_SCN_ALIGN_512BYTES             0x00A00000
#define IMAGE_SCN_ALIGN_1024BYTES            0x00B00000
#define IMAGE_SCN_ALIGN_2048BYTES            0x00C00000
#define IMAGE_SCN_ALIGN_4096BYTES            0x00D00000
#define IMAGE_SCN_ALIGN_8192BYTES            0x00E00000
#define IMAGE_SCN_LNK_NRELOC_OVFL            0x01000000 //   Section contains extended
                                                        relocations.
#define IMAGE_SCN_MEM_DISCARDABLE            0x02000000 //   Section can be discarded.
#define IMAGE_SCN_MEM_NOT_CACHED             0x04000000 //   Section is not cachable.
#define IMAGE_SCN_MEM_NOT_PAGED              0x08000000 //   Section is not pageable.
#define IMAGE_SCN_MEM_SHARED                 0x10000000 //   Section is shareable.
#define IMAGE_SCN_MEM_EXECUTE                0x20000000 //   Section is executable.
#define IMAGE_SCN_MEM_READ                   0x40000000 //   Section is readable.
#define IMAGE_SCN_MEM_WRITE                  0x80000000 //   Section is writeable.
```

Nous allons avoir besoin de ProcDump ici pour nous donner l'Entry Point en virtual offset de ART.
Lancer Pdump => PE Editor => charger ART.EXE => Entry Point : 499B9 (ceci est le virtual offset, bref c'est l'offset en mémoire et si vous voulez connaître l'adresse, vous ajoutez l'Image Base = 400000, ce qui nous donne 4499B9 comme Entry Point en mémoire).
Pourquoi fait-on ça ? Pour savoir dans quelle section se trouve l'Entry Point.
Allez maintenant dans SECTION et regardez la colonne " virtual offset ".
499B9 se trouve dans .CODE car 499B9 est > 1000 (le début de la section .CODE) et est < 86000 (le début de la section .DATA).
Voilà une bonne chose … maintenant regardons les attributs de la section .CODE
Cliquez avec le bouton de droite de la souris sur CODE (colonne NAME toujours dans ProcDump of course ☺) et choisissez EDIT.
En bas, à droite se trouvent les " Section Characteristics " (= les attributs) qui valent : C00000040 qui, si on se réfèrent à notre tableau ci-dessus veut dire : readable (40000000 => accès en lecture) + writable (80000000 => accès en écriture => c'est typique des exe compressés, les programmes habituels protègent leur code en ne permettant pas l'écriture DANS le code, contrairement à la section DATA où on peut lire et écrire) + Init Data (00000040) = C0000040.
Si vous regardez de plus près la liste, vous découvrirez deux choses intéressantes : 20000000 (exécutable) et 00000020 (contient du code).
Maintenant, si on disait que notre section doit être 20000000 + 40000000+ 80000000 + 00000020 = E00000020 ?
Changeons C0000040 en E0000020 et quittons ProcDump.
Essayez maintenant de désassembler ART.EXE avec Wdasm … vi vi ça marche ☺
Cliquons sur l'icône PEP et on arrive à l'Entry Point (4499B9), allez sur cette ligne et vous verrez tout en-bas @48DB9 (vous savez bien, le truc que vous regardez pour savoir où patcher …) et bien c'est le " raw Offset ".

Petite remarque juste pour rire … si maintenant (après notre modification) vous utilisez le Symbol Loader de SI avec ART.EXE, il va faire un break au début ☺ Cocasse, non ? On l'a juste en peu aidé en lui expliquant que la section où se trouve l'EP contient du code ;) Ceci dit ça ne fonctionne pas à tous les coups, surtout si les attributs ne sont pas C0000040. Donc, continuons l'autre technique, dite du BPINT3

2°. *Le sportif*

Reprenons : ProcDump => PE Editor => charger ART.EXE => EP = 499B9 => SECTIONS
Le virtual offset de l'EP est égal à 499B9 … mais encore ?
Que fait Windows quand il lance un programme ?
Il va chercher l'Image Base du programme (ici = 400000), il va voir le virtual offset de la première section (ici CODE avec un VO de 1000) et sa virtual size (ici 85000) et il alloue ensuite un espace depuis 401000 (400000 + 1000) jusqu'à 486000 (401000 + 85000) où il va aller placer le contenu de cette section CODE … et ainsi de suite pour les sections suivantes, jusqu'à la dernière.
Si on regarde le " Raw Offset " de CODE, Pdump nous donne 400, ça veut dire que l'octet qui sera placé en mémoire à 401000 est dans le fichier à l'offset 400.
Donc, nous ce qui nous intéresse, c'est l'endroit (raw offset) où se trouve l'EP.
Le calcul est simple : puisque le 400ème (hexa) octet du fichier sera en 401000 en mémoire … alors à quel emplacement sera l'octet trouvé à l'adresse 4499B9 ?
Si ça avait été l'octet 0 du fichier qui se retrouvait en 401000 il aurait suffit de faire 4499B9 – 401000 et on avait le raw offset … mais ici c'est le 400ème octet qui est à 401000, donc il faut faire 4499B9 – 401000 + 400 = 48DB9 CQFD ☺

Quelle qu'ait été la manière d'y arriver, nous avons notre raw offset pour l'EP : 48DB9
A l'aide d'un éditeur héxa, nous allons modifier la valeur à cet offset (ici 90) et la remplacer par un breakpoint (INT 3 = CC).

A ce propos, je vous rappelle que ce que nous venons de faire est exactement ce que fait Soft Ice mais de façon transparente à nos yeux. Quand on place un BPX xxxxxx il va mettre un CC en xxxxxx et retient la valeur qu'il y avait en xxxxxx. Ensuite, quand il breakera, il prendra soin de remettre la bonne valeur pour que nous puissions voir le listing correct.

Pour que SI réponde maintenant il faut lui indiquer que l'on souhaite qu'il s'arrête lorsqu'il rencontre une INT3. Pour ce faire on va dans SI (ctrl –D) et on tape BPINT 3.
A partir de maintenant chaque fois qu'il tombera sur l'instruction CC (INT3) il fera un break.

Une dernière chose : il ne faut surtout pas oublier de restaurer la bonne valeur à l'adresse 4499B9 (à savoir 90) sinon ça va crasher ☹

Bien sûr, dans ce cas-ci on peut utiliser le Symbol Loader de SI, mais je trouvais important de vous expliquer comment procéder pour d'autres situations moins glorieuses.

4. Le lancement d'Apollo 13

Voyons maintenant ce que cache notre ami ART.EXE …

Voici ce que vous devriez voir au début :

```
:004499B9 90                    nop
:004499BA 7500                  jne 004499BC

* Referenced by a (U)nconditional or (C)onditional Jump at Address:
|:004499BA(C)
|
:004499BC E93F660900            jmp 004E0000
:004499C1 00000000000000000000  BYTE 10 DUP(0)
:004499CB 00000000000000000000  BYTE 10 DUP(0)
:004499D5 00000000000000000000  BYTE 10 DUP(0)
:004499DF 00000000000000000000  BYTE 10 DUP(0)
:004499E9 00000000000000000000  BYTE 10 DUP(0)
:004499F3 00000000000000000000  BYTE 10 DUP(0)
:004499FD 000000                BYTE  3 DUP(0)



* Referenced by a (U)nconditional or (C)onditional Jump at Address:
|:004499BC(U)
|
:004E0000 60                    pushad
```

```
:004E0001 E800000000              call 004E0006

* Referenced by a CALL at Address:
|:004E0001
|
:004E0006 5D                      pop ebp
:004E0007 81ED3ED94300            sub ebp, 0043D93E
:004E000D B838D94300              mov eax, 0043D938
:004E0012 03C5                    add eax, ebp
:004E0014 2B850BDE4300            sub eax, dword ptr [ebp+0043DE0B]
:004E001A 898517DE4300            mov dword ptr [ebp+0043DE17], eax
:004E0020 80BD01DE430000          cmp byte ptr [ebp+0043DE01], 00
:004E0027 7515                    jne 004E003E
:004E0029 FE8501DE4300            inc byte ptr [ebp+0043DE01]
:004E002F E81D000000              call 004E0051
:004E0034 E879020000              call 004E02B2
:004E0039 E812030000              call 004E0350

* Referenced by a (U)nconditional or (C)onditional Jump at Address:
|:004E0027(C)
|
:004E003E 8B8503DE4300            mov eax, dword ptr [ebp+0043DE03]
:004E0044 038517DE4300            add eax, dword ptr [ebp+0043DE17]
:004E004A 8944241C                mov dword ptr [esp+1C], eax
:004E004E 61                      popad
:004E004F FFE0                    jmp eax
```

Donc, depuis 4499B9 le programme nous envoie en 4E0000.
Si on trace un peu on se rend vite compte que la situation se reproduit 3 fois de suite avant d'arriver à notre cible (ART) :
4499B9 (jump) envoie en 4E0000 (procédure) qui envoie en
443239 (jump) qui envoie en 4DF000 (procédure) qui envoie en
43D519 (jump) qui envoie en 4DE000 (procédure) qui envoie enfin en
485E98 (le début de ART.EXE décompressé, bref, le véritable Entry Point du programme).
En détaillant encore un peu plus, vous vous rendrez vite compte que les listings en 4E0000, 4DF000 et 4DE000 sont IDENTIQUES !
En plus, en regardant avec ProcDump les différentes sections, on constate que chacune des adresse ci-dessus correspond à une section séparée … en fait 3 sections consécutives nommées .data toutes les trois.

Nous en déduisons donc facilement que le soft est compressé en 3 passes.

Ils ont juste commis une erreur mais bien légitime … à actions similaires, code similaire.
C'est normal, un compresseur utilise un seul algorithme de compression / décompression qu'il va appliquer X fois d'affilée.
A nous de tirer parti de cette faiblesse …

En fait, le code que l'on retrouve en 4E0000, 4DF000 et 4DE000 n'est autre que ce qu'on pourrait appeler une procédure de décompression.
Comme il s'agit d'une procédure, elle aura des paramètres d'entrée et fournira un ou des résultats en sortie, ce qui donne en gros :
Procedure Unpack (source offset, data size, destination offset)
Evidemment j'ai inventé cet exemple fictif et réducteur, mais ça donne une idée du fonctionnement.
On appelle UNPACK en lui donnant l'adresse des données compressées (source offset), leur taille (data size) et l'emplacement où il devra mettre les données

décompressées (destination offset). Il envoie en fin de compte à l'adresse (une sorte d'Entry Point) d'entrée du code décompacté, etc … jusqu'au programme principal.
Remarque : je continuerai à faire référence par la suite à cette procédure UNPACK par facilité pour parler de 4E0000, 4DF000 et 4DE000

Que pouvons-nous tirer comme conclusion ici ?
Simplement que nous n'avons besoin que d'une occurrence du code UNPACK, le tout étant de lui fournir les bons paramètres d'entrée pour qu'il fournisse une sortie correcte ☺
Oui mais quels sont les bons paramètres ? Traçons pour voir ce qui se passe, sans entrer dans les calls.

```
:004E0000 60                    pushad              <- push tous les registres sur la Pile
:004E0001 E800000000            call 004E0006       <- l'intérêt de ceci est de placer
                                                    l'adresse de retour du call sur la Pile
                                                    ( = 4E0006)

* Referenced by a CALL at Address:
|:004E0001
|
:004E0006 5D                    pop ebp             <- récupère EIP (façon astucieuse de
                                                    repérer où on se trouve dans un
                                                    programme quand il faut insérer du code
                                                    n'importe où)
:004E0007 81ED3ED94300          sub ebp, 0043D93E   <- voici la seule chose qui change
                                                    d'une section à l'autre ; en fait EBP
                                                    va servir de pointeur relatif vers des
                                                    données … à partir de là, les datas
                                                    seront calculées par [EBP+xxxxxx]
:004E000D B838D94300            mov eax, 0043D938
:004E0012 03C5                  add eax, ebp
:004E0014 2B850BDE4300          sub eax, dword ptr [ebp+0043DE0B]
:004E001A 898517DE4300          mov dword ptr [ebp+0043DE17], eax <- en définitive eax
                                                                  contient toujours l'Image
                                                                  Base = 400000
```

Par chance, on tombe rapidement sur ce que l'on cherchait, à savoir la variable qui déterminera quelles données utiliser pour la décompression. Cette variable est EBP.
Peut-être êtes vous largués avec cette histoire … je comprendrais … voici une métaphore pour expliquer comment ça marche :

Imaginons une femme de chambre dans un hôtel. Elle a une série de tâches à accomplir (refaire le lit, remplacer les serviettes, vider les cendriers … limitons nous à ces 3 là) et ce, quelle que soit la chambre.
Donc ses actions sont systématisées, elles les accomplis dans le même ordre et de la même manière pour chaque chambre dans laquelle elle entre.
Elle doit pourtant savoir UNE chose : quelles sont les chambres où l'on a dormi et parmi celles-ci, lesquelles sont à présent vides (ben oui ça vous plairait à vous que quelqu'un refasse le lit dans lequel vous êtes couché ? ☺).
Il lui faudra donc une liste des chambres correspondantes à ces deux critères afin qu 'elle puisse exécuter ses tâches définies.
Mettons qu'il y ait la 103, la 215 et la 308. Il lui faudra utiliser ces références comme pointeurs vers ses activités. Voilà le principe de EBP … un pointeur. C'est clair ? Non ? Bon ben faites un effort, je vais pas passer mon temps à donner des exemples ;)

Pour en revenir à EBP, on voit qu'il aura successivement les valeurs A26C8, A16C8 et A06C8 en fonction des sections. Celui qui à remarqué qu'on soustrait 1000 d'une

fois à l'autre est observateur … mais juste ce qu'il faut, hein ? C'est pas extraordinaire ☺

Tentons l'expérience. On laisse passer la première section jusqu'en 4E004F où eax contient l'adresse de la seconde section DATA (4DF000) et forçons eax pour que le programme boucle en 4E0000 (eax = 4E0000).
On retourne en 4E0000 et on s'arrête en 4E0007 où EBP vaut de nouveau A26C8.
Changeons A26C8 en A16C8. On continue jusqu'en 4E004F et on recommence.
Cette fois-ci EBP vaudra A06C8.
Arrivé en 4E004F, eax contiendra cette fois l'Entry Point du programme décompressé (485E98) donc on laisse courir.
Bingo tout s'est passé comme si le programme était passé par les 3 sections DATA.
L'hypothèse de départ était correcte … EBP est bien le pointeur relatif vers les données à traiter.

Nous pouvons à présent travailler à l'aise sur du code en clair pour appliquer notre patch. Il faudra juste modifier un peu pour que le programme boucle 2 fois (j'ai dit " boucle " 2 fois, donc il passe 3 fois sur ce code …) sur 4E0000.

Si vous êtes familier avec le patchage de softs compressés, vous savez sûrement qu'il faut trouver un petit endroit inoccupé pour y greffer notre propre crack.
On saute alors du code du décompresseur vers ledit " petit endroit " pour y revenir quand le boulot (patch) est accompli.
Le tout étant de trouver une bonne place.

Pour ma part j'étais très confiant car si vous vous souvenez de l'Entry Point du programme (4499BC) on avait pas mal de place en-dessous du JMP (pour être précis, on a 63 octets).

J'y ai donc placé mon premier patch pour me rendre compte avec horreur que cette partie de la section CODE se voit écrasée par du code décompressé lors de la 3^{e} passe du décompresseur (et c'est en fait assez logique puisque la 3^{e} passe est équivalente à la première compression qui s'est nécessairement occupée de la compression du code principal du programme).

Shit ! Comme diraient nos confrères anglophones ☹ Il va falloir trouver autre chose.
L'ennui avec les multicompressions c'est que trouver un espace vide pour y mettre son code relève souvent de l'utopie.
C'est là que ma technique va nous venir en aide ☺
On va très vite avoir toute la place que l'on veut … dès le premier passage de UNPACK.
En effet, puisqu'on boucle sur 4E0000, les procédures en 4DF000 et 4DE000 deviennent absolument inutiles, ce qui nous arrange et nous donne de la place où l'on est sûr que rien d'autre ne viendra se mettre.
Euh ouais c'est bien beau tout ça me direz-vous, mais 4DF000 ne sera disponible qu'après le premier passage de UNPACK, donc pas question d'y mettre un seul octet avant ce passage, il serait écrasé …
Bien vu, répliquerais-je ! Y en a quand même qui suivent, c'est rassurant ☺
Nous allons donc nous servir de nos 63 octets en 4499C1 pour y mettre les données à transférer après la première décompression.

Voici la modification (je vous conseille de la lire en suivant les jumps et les calls si vous voulez comprendre – le patch est en vert) :

```
:004499B9 90                    nop
:004499BA 7500                  jne 004499BC

* Referenced by a (U)nconditional or (C)onditional Jump at Address:
|:004499BA(C)
|
:004499BC E93F660900            jmp 004E0000 <- saut vers UNPACK

* Referenced by a CALL at Address:
|:004E0044
|
:004499C1 BF00F04D00            mov edi, 004DF000
:004499C6 BEDC994400            mov esi, 004499DC
:004499CB B908000000            mov ecx, 00000008
:004499D0 F2                    repnz
:004499D1 A5                    movsd <- transfert de la zone 4499DC -> 4499FB vers 4DF000
:004499D2 C70544004E00E8B7EFFF  mov dword ptr [004E0044], FFEFB7E8 <- astuce que je
                                                                    commenterai après
:004499DC 038517DE4300          add eax, dword ptr [ebp+0043DE17] <- ligne précédemment
                                                                  écrasée (4E0044) pour y
                                                                  mettre notre call 4499C1
:004499E2 3D985E4800            cmp eax, 00485E98 <- la compression est-elle finie ?
:004499E7 7407                  je 004499F0 <- oui
:004499E9 B800004E00            mov eax, 004E0000 <- non donc on doit retourner en 4E0000

* Referenced by a (U)nconditional or (C)onditional Jump at Address:
|:004499FA(U)
|
:004499EE 50                    push eax <- on place l'adresse de retour sur la Pile
:004499EF C3                    ret <- et on y va ☺



* Referenced by a (U)nconditional or (C)onditional Jump at Address:
|:004499E7(C)
|
:004499F0 C705A60B480033C0405F  mov dword ptr [00480BA6], 5F40C033 <- patch de ART.EXE (v.
                                                                    Christal)
:004499FA EBF2                  jmp 004499EE
:004499FC 00000000              BYTE  4 DUP(0)



* Referenced by a (U)nconditional or (C)onditional Jump at Address:
|:004499BC(U)
|
:004E0000 60                    pushad
:004E0001 E800000000            call 004E0006

* Referenced by a CALL at Address:
|:004E0001
|
:004E0006 5D                    pop ebp
:004E0007 81ED3ED94300          sub ebp, 0043D93E  ! 4E000A est en rouge
:004E000D B838D94300            mov eax, 0043D938
:004E0012 03C5                  add eax, ebp
:004E0014 2B850BDE4300          sub eax, dword ptr [ebp+0043DE0B]
:004E001A 898517DE4300          mov dword ptr [ebp+0043DE17], eax
:004E0020 80050A004E0010        add byte ptr [004E000A], 10 <- ajout de 10 à 4E000A
:004E0027 EB06                  jmp 004E002F <- passe les lignes inutiles
:004E0029 FE8501DE4300          inc byte ptr [ebp+0043DE01]

* Referenced by a (U)nconditional or (C)onditional Jump at Address:
|:004E0027(U)
|
:004E002F E81D000000            call 004E0051
:004E0034 E879020000            call 004E02B2
:004E0039 E812030000            call 004E0350
:004E003E 8B8503DE4300          mov eax, dword ptr [ebp+0043DE03]
:004E0044 E87899F6FF            call 004499C1 <- vers le patch
:004E0049 90                    nop
:004E004A 8944241C              mov dword ptr [esp+1C], eax
```

```
:004E004E 61                    popad
:004E004F FFE0                  jmp eax
```

Je pense que tout ceci mérite quelques explications ☺

1. `:004E0020 add byte ptr [004E000A], 10`

Je vous ai dit que EBP était le pointeur et que celui-ci diminuait de 1000 à chaque passage dans UNPACK. Donc, en ajoutant 10 à l'adresse 4E000A, le " SUB EBP, 43D93E " deviendra " SUB EBP, 43E938 " puis " SUB EBP, 43F938 ". Vous aurez compris que de cette manière EBP aura les valeurs nécessaires en fonction du passage : A26C8, A16C8 et A06C8.

2. Après la première décompression, le programme saute vers le patch (call 4499c1).
Que fait-il ?
1° - Il transfert le code de 4499DC à 4499FB vers 4DF000

```
:004499C1 BF00F04D00            mov edi, 004DF000
:004499C6 BEDC994400            mov esi, 004499DC
:004499CB B908000000            mov ecx, 00000008
:004499D0 F2                    repnz
:004499D1 A5                    movsd
```

2° - `004499D2 mov dword ptr [004E0044], FFEFB7E8`
Place E8 B7 EF FF en 4E0044 ? Ca fait quoi ça ? Héhé ☺

Voilà ce qu'il y a avant :
```
:004E0044 E87899F6FF            call 004499C1 <- vers le patch 1
```

Voilà ce qu'il y a après :
```
:004E0044 E8b7efffFF            call 004df000 <- vers le patch 2
```

Vous voyez ? Au premier coup, le patch se trouve en 4499C1 mais quand celui-ci est passé, la partie principale a été transférée vers 4DF000 qui deviendra alors notre patch. Donc, il faut qu'à la 2ème et 3ème décompression, le programme se branche sur 4DF000 et plus sur 4499C1 puisque celui-ci sera écrasé au 3e passage.

3° -
```
:004499E2 3D985E4800            cmp eax, 00485E98
:004499E7 7407                  je 004499F0 <- oui
```

En fin de décompression eax contient le point d'entrée du code décompressé.
Ou bien tout est décompressé et eax contient l'Entry Point de ART décompressé (485E98) ou bien celui du UNPACK suivant (4DF000 et 4DE000).
Dans ce deuxième cas, nous forçons le programme à retourner en 4E0000.

```
:004499E9 B800004E00            mov eax, 004E0000
:004499EE 50                    push eax
:004499EF C3                    ret
```

Si la décompression est finie on patche le programme cible comme l'a fait Christal.
Si vous vous demandez pourquoi j'ai placé un Dword alors qu'il ne fallait remplacer que 3 octets c'est tout simplement pour gagner de la place. Faire un " mov dword ptr " à la place d'un " mov word ptr " puis un " mov byte ptr " prend quelques octets de moins. Il faut juste remettre à sa place l'octet précédant ou suivant les 3 bytes à modifier pour que l'on ait 4 bytes.
```
:004499F0 C705A60B480033C0405F  mov dword ptr [00480BA6], 5F40C033
:004499FA EBF2                  jmp 004499EE
```

Et on va vers 4499EE où on push eax (= 480BA6) et le RET branche le programme vers cette adresse … le soft est cracké ☺

4° - Vous avez peut-être remarqué que j'ai zappé allègrement la fin de UNPACK :

```
:004E0049 90                    nop
:004E004A 8944241C              mov dword ptr [esp+1C], eax
:004E004E 61                    popad
:004E004F FFE0                  jmp eax
```

En réalité, le programme boucle depuis le patch et le Push eax RET remplace la partie ci-dessus car elle fait en gros la même chose. Mov dword ptr [esp+1C], eax place eax sur la pile à l'endroit où a été pushé eax en 4E0000 par le pushad, puis le popad récupère tous les registres dans l'état où ils étaient au début de la routine SAUF eax qui vient d'être modifié directement sur la pile (brumeux, hein ? ;)). En fin de course, le jmp eax saute au code décompressé.
La seule chose que je n'ai pas reprise c'est le popad, mais ça n'a pas d'importance, la pile peut rester dans cet état car on boucle sur une procédure " neuve ".

5° - ATTENTION : nous avons effectué à plusieurs reprises (4499D1 movsd / 4499F0 mov dword ptr [00480BA6], 5F40C033 / 4E0020 add byte ptr [004E000A], 10 / 4499D2 mov dword ptr [004E0044], FFEFB7E8) des écritures dans des parties de mémoire contenant du code. Ceci n'a été possible que parce que nous nous trouvions dans une situation de décompression. En effet, pour qu'un soft puisse se décompresser, il doit pouvoir écrire le résultat de cette décompression sur " lui-même ", donc avoir accès à sa mémoire en lecture / écriture. Dans les situations traditionnelles, Windows empêche d'écrire dans le code et provoque un message d'erreur si on tente cette manipulation.
Donc, ne tentez jamais cela dans d'autres situations ou vous allez vers le crash assuré.

6° - Habituellement, quand on greffe son code on utilise un JMP vers le patch puis un JMP du patch pour retourner au programme principal. La raison pour laquelle j'ai choisi un call est due au fait que le retour varie suivant la cas où le patch est en 4499C1 ou en 4DF000. Dans le listing ce serait toujours un jmp 4E0000 mais au niveau du code (les octets en mémoire) ce serait différent. En effet, un jump se calcul en fonction du nombre d'octets entre le JMP et sa destination. Dès lors, suivant que le JMP 4E0000 se trouve en fin du patch 1 ou du patch 2, le nombre d'octets varie. Par contre, un Push eax RET permet de contourner ce problème puisque le RET attribue l'adresse de retour au registre qui pointe vers l'instruction actuelle (EIP), sans qu'un calcul ne soit nécessaire.

Conclusion

Nous voici au terme de ce cours sur la multicompression.
Je ne prétends pas que ma méthode soit la meilleure, loin de là, mais elle a un avantage sur celle généralement utilisée qui consiste à patcher successivement les différents niveaux de compressions, c-à-d qu'on repère l'endroit où le décompresseur passe la main au programme principal et juste avant on le fait sauter vers le patch. Le problème réside dans le fait que cette adresse n'existera en clair qu'après plusieurs décompressions. Il faut alors patcher d'une décompression à l'autre et plus il y a de couches de compressions, plus il y a de patches à appliquer.
L'avantage ici est de ne pas être dépendant du nombre de compressions. Il suffit de connaître l'EP du programme décompressé et les adresses à patcher dans celui-ci.

Ainsi, il y aurait pu y avoir 15 compressions que ça n'aurait rien changé … le programme aurait bouclé tant que l'EP n'était pas atteint ... enfin presque …
Il faudrait alors prendre soin de modifier les lignes suivantes :

```
:4E0020 6683050A004E0010        add word ptr [004E000A], 10
:4E0028 EB05                    jmp 4E002F
```

En effet, j'avais choisi de faire un ADD sur l'octet 4E000A car l'addition ne se faisait que 3 fois et la ligne se changeait de SUB EBP, 43D93E en 43E93E puis 43F93E. S'il avait fallu ajouter encore 10 on aurait eu 43093E donc pour éviter cela, il faut faire un ADD sur le " word " en 4E000A car de cette manière, après 43F93E on a 44093E, ce qui est correct.

Malgré l'aspect un peu ardu de la chose, nous avons tenté de rendre ça le plus compréhensible possible. Toutefois si des zones d'ombre subsistent vous pouvez toujours demander des clarifications que j'ajouterai si nécessaire à ce tut.
Une chose encore, comme vous avez pu le constater, les tout débutants seront certainement noyés par ce texte et c'est normal, d'autres lectures sont nécessaires avant d'aborder celle-ci.
C'est aussi pourquoi je n'ai pas détaillé à 100% l'intégralité de la démarche, cela aurait été vraiment fastidieux.
Sur ce, nous vous souhaitons une bonne relecture ☺ et une bonne journée (n'est-ce pas Christal ? ;))