Comprendre FSG

(Analyse de l'ApLib)

FAST SMALL GOOD

par BeatriX

1 . INTRODUCTION.....2

2 . Packer un exe avec FSG..... 2

3 . Le loader de FSG..... 4

a . Schéma du loader.....5
b . Reconstruction de l'IAT..... 5
c . Analyse de l'ApLib 0.34 de Joergen Ibsen..... 6
Les compresseurs d'exe : principe général..... 7
La compression LZ78 : principe..... 7
La décompression LZ78..... 8
Remarque sur les index..... 12
Analyse du décompresseur de l'ApLib 0.34.....12
Comment procède le compresseur ?.....12
Que se passe-t-il s'il n'y a pas de répétitions ?..... 14
Contraintes.....14
Déroulement des opérations..... 15
Optimisation du code.....15
Peut-on récupérer une position déjà calculée ?.....16
Schéma de la décompression..... 18
Comment fonctionnent les index ?..... 18
Peut-on patcher un exe packé par FSG ?..... 24
Analyse du code du décompresseur..... 25

4 . Le packer FSG.EXE..... 28

a . Schéma du packer..... 28
b . anti-debuggers.....29
Les obfuscations.....29
Le CCA (faiblesse de OllyDbg).....30

5 . Remerciements.....38

1 . INTRODUCTION

Je vous propose ici d'étudier le packer d'exécutables F[ast] S[mall] G[ood]. Il ne s'agit pas ici de présenter une technique de manual unpacking mais plutôt d'étudier le code du packer et de comprendre ses subtilités. FSG ne cherche pas à protéger le programme comme certains packers/crypters mais « simplement » à compresser les données tout en laissant l'exécutable autonome. Dans cette étude, je me suis intéressé de près à l'algorithme de compression et je vais donc vous expliquer comment fonctionne cette compression. La compression de données relève d'une programmation subtile et astucieuse. En expliquer tous les aspects reste quelque chose de très technique. Je ne prétends pas avoir fait le tour de la question mais je vous propose un aperçu aussi complet que possible tout en essayant de rester clair et abordable rapidement.

Pourquoi avoir choisi FSG ?

J'ai constaté que de nombreux crackmes sont packés avec FSG. J'ai donc cherché à comprendre pourquoi. Surtout, pourquoi avoir inventer un nouveau packer alors qu'il en existe déjà dont, entre autre, l'excellent UPX. Vous verrez en fait qu'il ne fait pas double emploi et qu'il a bien sa place dans l'univers des packers même si son taux de compression est moins bon que UPX.

FSG est l'oeuvre récente d'une équipe polonaise menée par Bart. Le coder principal de ce petit packer est Dulek. Voici les différentes versions proposées depuis la création de FSG :

FSG 1.0 : 14/01/2002
FSG 1.1 : 26/04/2002
FSG 1.2 : 04/05/2002
FSG 1.3 : 21/08/2002
FSG 1.31: 24/08/2002
FSG 1.33: 15/11/2002
FSG 2.0 : 24/05/2004

Je vais vous montrer comment fonctionnent toutes ces versions sans vraiment faire de distinction réelle. La raison est simple: il n'y a pas de changements radicaux d'une version à l'autre.

2 . Packer un exe avec FSG.

Premier constat : c'est extrèmement simple d'utilisation. Inutile de rentrer une ligne de commande pour compresser le fichier voulu puisque le compresseur vous demande via l'interface standard d'ouvrir l'exe à packer. Suivant la version utilisée, vous avez (ou non) une petite fenêtre qui vous informe de l'avancement de la compression au fur et à mesure que le travail se fait. Vous avez même droit sur les dernières versions à un petit rire sarcastique qui signale la fin des opérations .

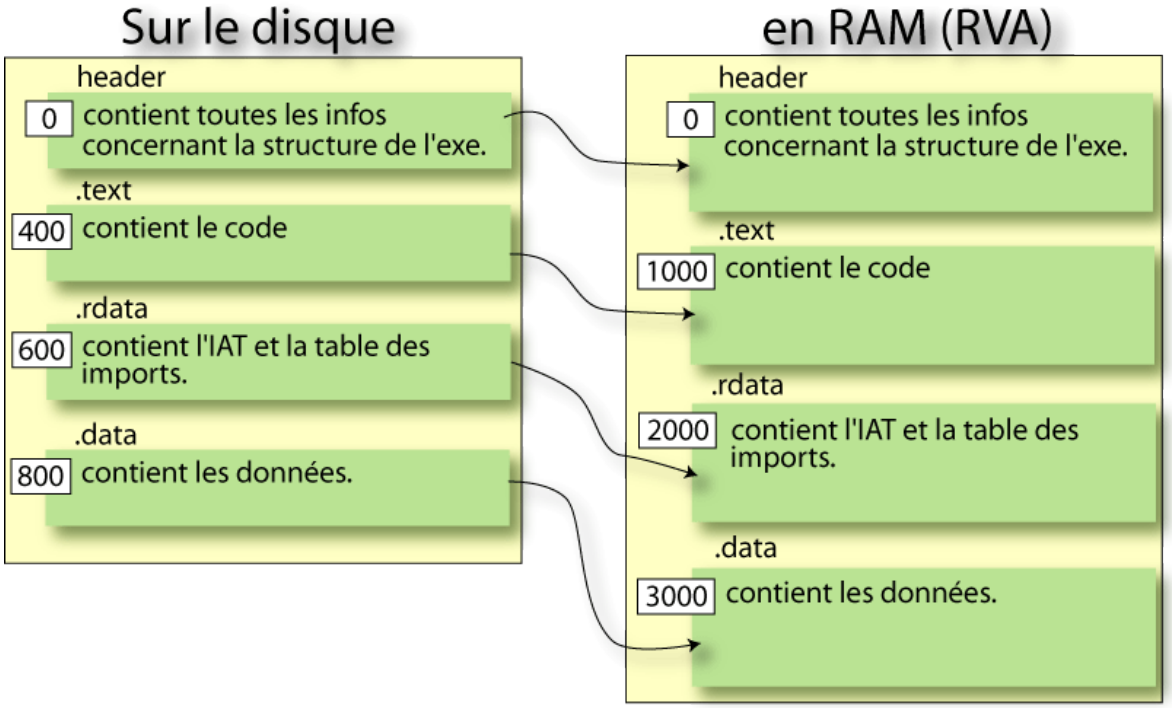
Deuxième constat : FSG ne compresse qu'une catégorie d'exécutables. Il est reservé, comme le précise ses auteurs, pour de petits exe de 4 ko (voire 64 ko), pas plus. Si vous tentez de packer des exe trop gros, la compression déraille et plante l'exécutable. Inversement, UPX a énormément de mal à compresser de petits exécutables...FSG comble donc le trou ! Evidemment, on peut toujours se demander quel est l'intérêt de compresser des exe de 4 ko !!! là, je n'ai pas de réponse...

A quoi ressemble un exe packé avec FSG ?

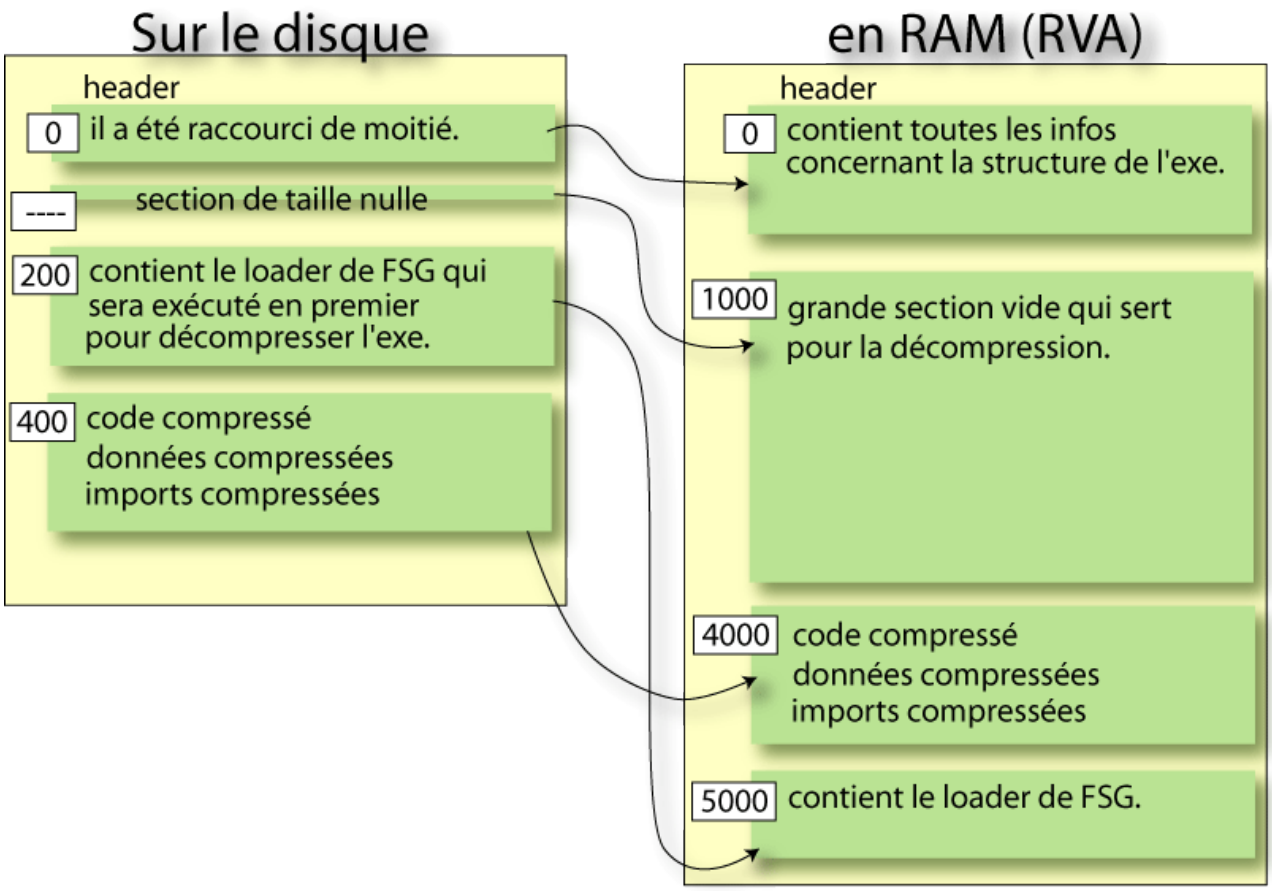
Prenons un exemple pour répondre à cette question. Tout au long de cette étude, j'ai travaillé sur un exe appelé msgbox.exe qui n'est rien d'autre que le résultat du deuxième cours des tutos d'iczelion. Voici le code asm de ce programme :

```
.386
.model flat,stdcall
option casemap:none
include \masm32\include\windows.inc
include \masm32\include\kernel32.inc
includelib \masm32\lib\kernel32.lib
include \masm32\include\user32.inc
includelib \masm32\lib\user32.lib

.data
MsgBoxCaption db "Iczelion Tutorial No.2",0
MsgBoxText    db "Win32 Assembly is Great!",0

.code
start:
invoke MessageBox, NULL, addr MsgBoxText, addr MsgBoxCaption, MB_OK
invoke ExitProcess, NULL
end start
```

Il ne fait que 2560 octets (2,5 Ko) et se compose des sections suivantes :



Après la compression, le décor a changé :



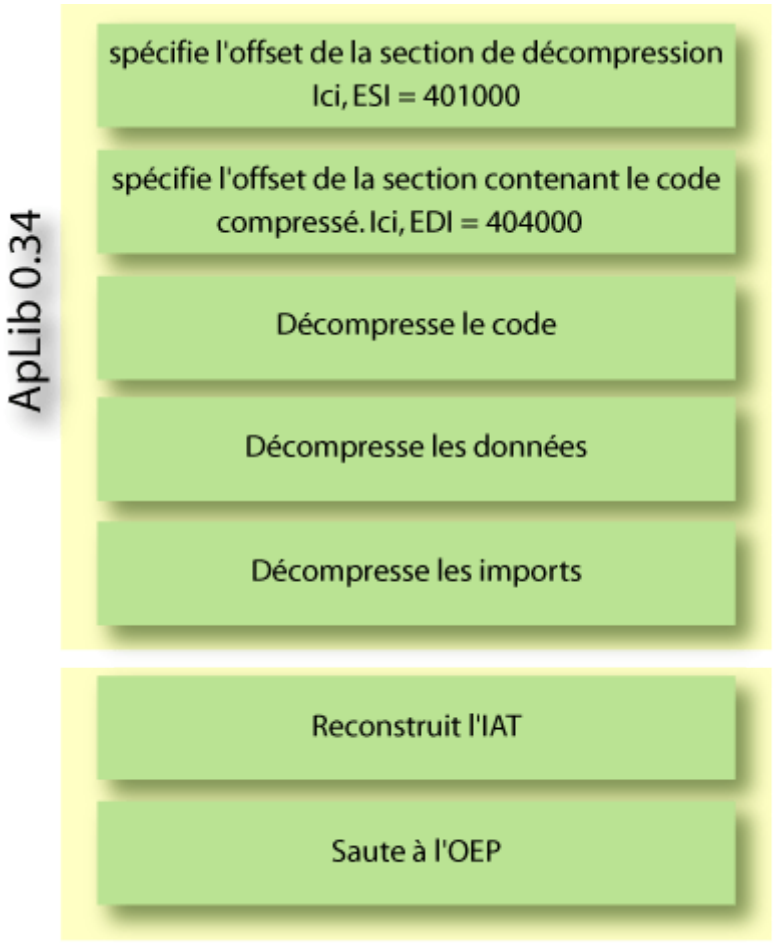
Vous remarquez que les sections n'ont plus du tout le même rôle. D'abord, il y a une section de taille nulle ; elle permet de mettre en place dans la RAM une section de taille 3000 qui va recevoir tout le code décompressé. Ensuite, il y a une section qui contient le loader de FSG qui se chargera de décompresser le code dans la grande section citée précédemment. Enfin, la dernière section contient le code, les données et les imports compressés. Notez également que le header a été diminué de moitié. Il ne s'agit pas ici de compression au sens noble du terme mais de se débarasser du padding (espace rempli de zéro) qui permet l'alignement indiqué dans le header. Les données dans ce dernier ne sont pas compressées. Tout est permis pour réduire la taille d'un exe !

3 . Le loader de FSG

Après avoir packé notre exécutable, nous allons analyser son comportement en milieu naturel. Nous allons plus particulièrement nous intéresser à la section contenant le loader (chargée de décompresser l'exe).

a . Schéma du loader

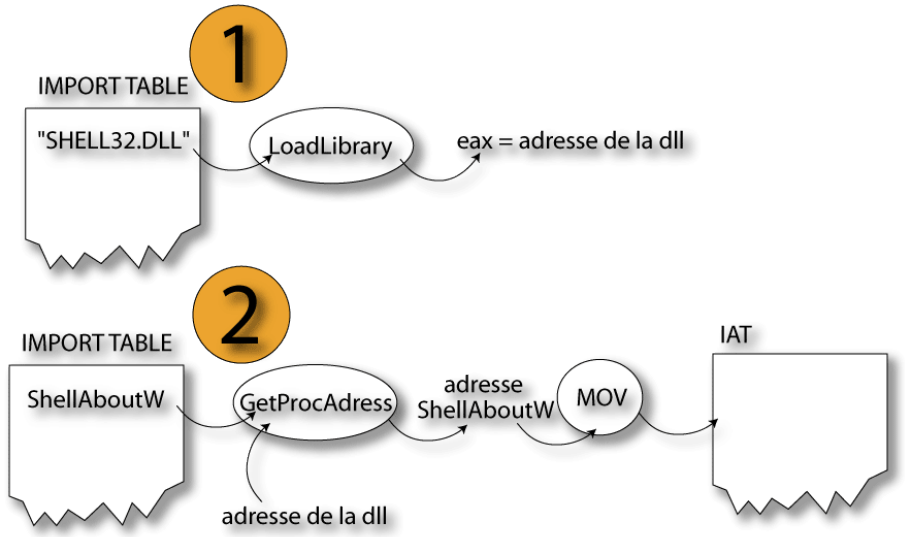
Voici schématiquement comment se présente le loader :



Il s'agit d'un schéma standard propre à tous les dépackers. Notez que l'algorithme de décompression est issu de l'ApLib, bibliothèque de procédures pour compresser et décompresser des exe.

b . Reconstruction de l'IAT

FSG suit le schéma classique pour reconstruire l'IAT : il utilise les deux fonctions LoadLibraryA et GetProcAddress de Kernel32.dll. Voici un exemple du déroulement de l'opération :



Voici le code commenté (pour la version 1.33) :

```
00404365  LODS DWORD PTR DS:[ESI]          Chaîne LoadLibraryA
00404366  XCHG EAX,EBX
00404367  POP ESI
00404368  INC ESI
00404369  LODS DWORD PTR DS:[ESI]          Chaîne Kernel32.dll
0040436A  XCHG EAX,EDI
0040436B  PUSH ESI                         Offset du nom de la Dll
0040436C  CALL DWORD PTR DS:[EBX]          Charge la Dll avec LoadLibraryA
0040436E  XCHG EAX,EBP
0040436F  LODS BYTE PTR DS:[ESI]           Boucle pour récupérer l'API suivante
00404370  TEST AL,AL
00404372  JNZ SHORT dialogBo.0040436F
00404374  DEC BYTE PTR DS:[ESI]            Diminue la valeur du premier caractère de l'API
00404376  JE SHORT dialogBo.00404368       Saut si on doit changer de Dll
00404378  JNS SHORT dialogBo.0040437F
0040437A  INC ESI
0040437B  LODS DWORD PTR DS:[ESI]
0040437C  PUSH EAX
0040437D  JMP SHORT dialogBo.00404388
0040437F  DEC BYTE PTR DS:[ESI]            Diminue la valeur du premier caractère de l'API
00404381  JE dialogBo.00401000             saut à l'OEP (version 1.33)
00404387  PUSH ESI                         Offset du nom de l'API
00404388  PUSH EBP                         Adresse de la Dll
00404389  CALL DWORD PTR DS:[EBX+4]         GetProcAddress
0040438C  STOS DWORD PTR ES:[EDI]          Remplit l'IAT
0040438D  JMP SHORT dialogBo.0040436F
```

c . Analyse de l'ApLib 0.34 de Joergen Ibsen

Comme je l'ai déjà précisé plus haut, le compresseur/décompresseur n'est pas l'oeuvre de l'équipe de Bart. Ils utilisent le travail remarquable de Joergen Ibsen (alias Jibz) compilé dans ce qu'il appelle l'ApLib. Il s'agit d'une solution complète de procédures pour compresser et décompresser des exécutables. Tant que vous ne commercialisez pas votre produit, vous pouvez utiliser librement cette bibliothèque. Jibz utilise ces algorithmes dans son packer aPack. J'ajoute à cela que de nombreux

packers/crypters d'exe freeware utilisent également l'ApLib (tElock, PeSpin...)

Les compresseurs d'exe : principe général.

A priori, il vient naturellement à l'esprit que dans toute chaîne de caractères, on trouve facilement des éléments qui se répètent (caractères – mots). Ce phénomène de répétition (appelé redondances) se trouve être l'une des clés pour parvenir à compresser des données. Il s'agit en effet dans un processus de compression de limiter ces redondances et de gagner ainsi de la place tout en garantissant l'intégrité des données (on dit alors que la compression est non-dégradante).

La compression LZ78 (principe) :

Portant le nom de ses deux créateurs Abraham Lempel et Jacob Ziv, la compression LZ regroupe des processus de compression très efficaces. En 1977, Lempel et Ziv publièrent leur première version d'un algorithme de compression général baptisé LZ77. En 1978, ils proposèrent une première amélioration. En 1984, Welch publia un algorithme LZ amélioré, le fameux LZW (utilisé de nos jours pour le format gif par exemple).

Ce type de compression se base sur l'idée évoquée précédemment : il existe des redondances dans toute chaîne « binaire ». Moyennant cet axiome de base, le principe d'une compression LZ78 consiste à indexer dans un dictionnaire les « mots » répétés et à les remplacer (dans les données compressées) par leur seul index. Le compression ne s'effectue qu'en un seul passage. La lecture des données est séquentielle du début de la chaîne à sa fin.

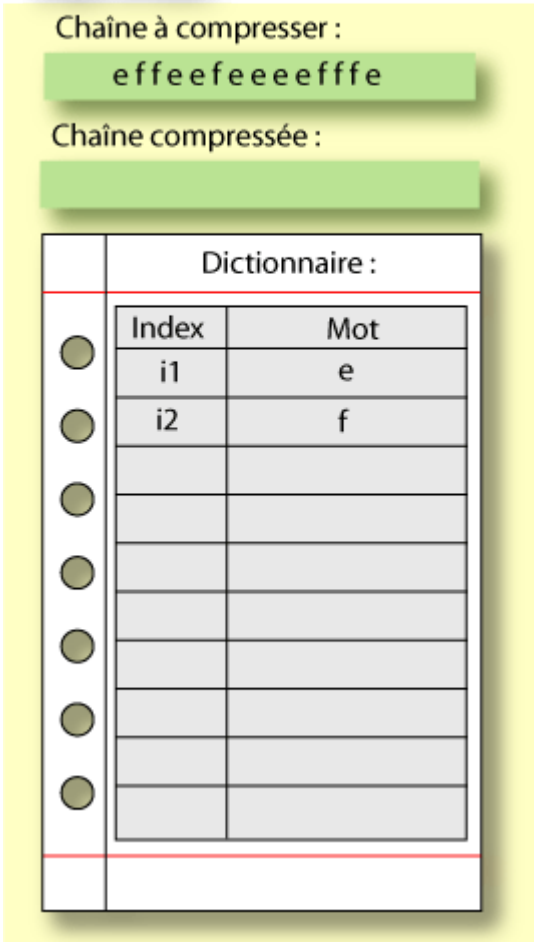
Chaque étape se déroule en deux temps :

1) On cherche si le mot est dans le dictionnaire. Si c'est le cas, on utilise l'index pour coder.

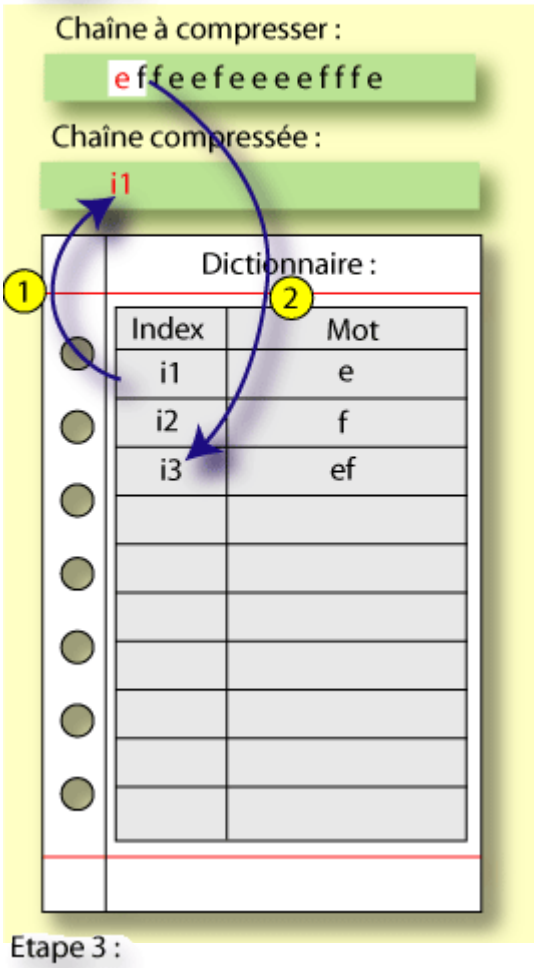
2) On ajoute une entrée dans le dictionnaire en concaténant le mot en cours avec le caractère suivant.

Voici illustré un exemple de compression LZ78 :

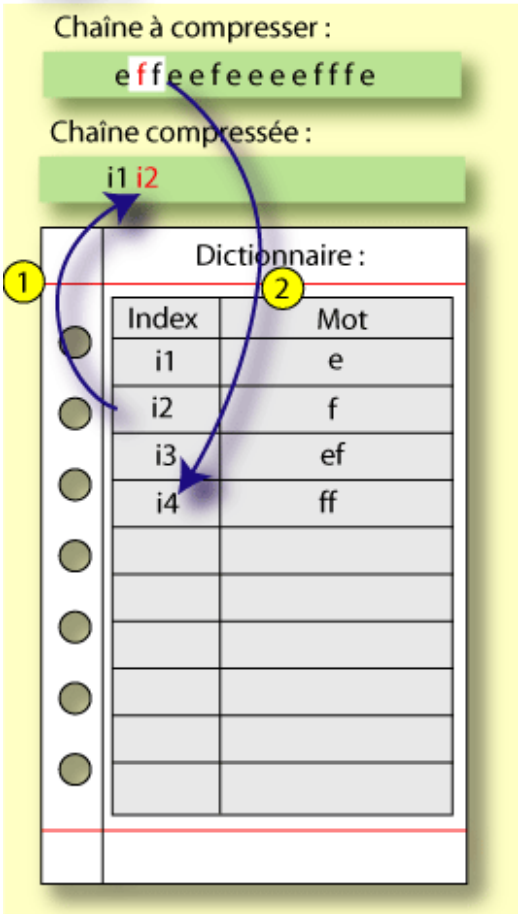
Etape initiale :



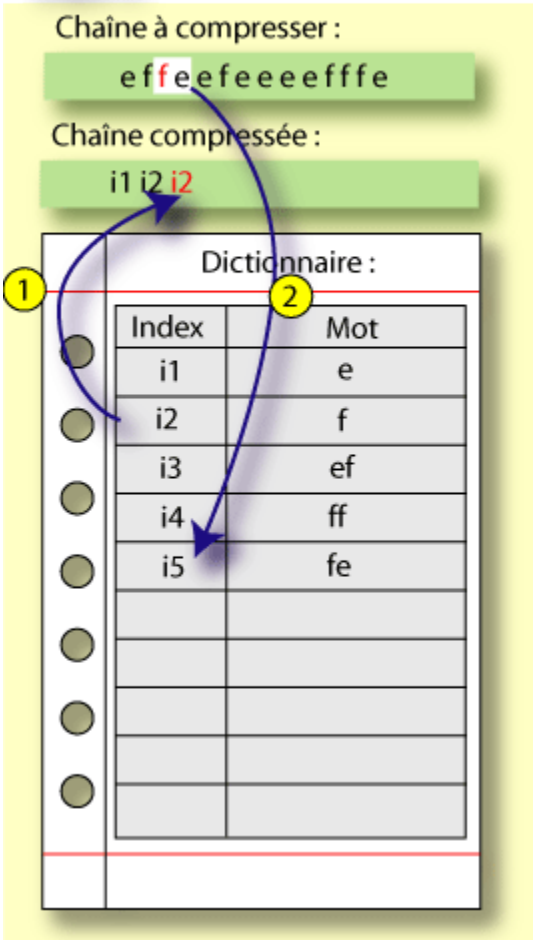
Etape 1 :



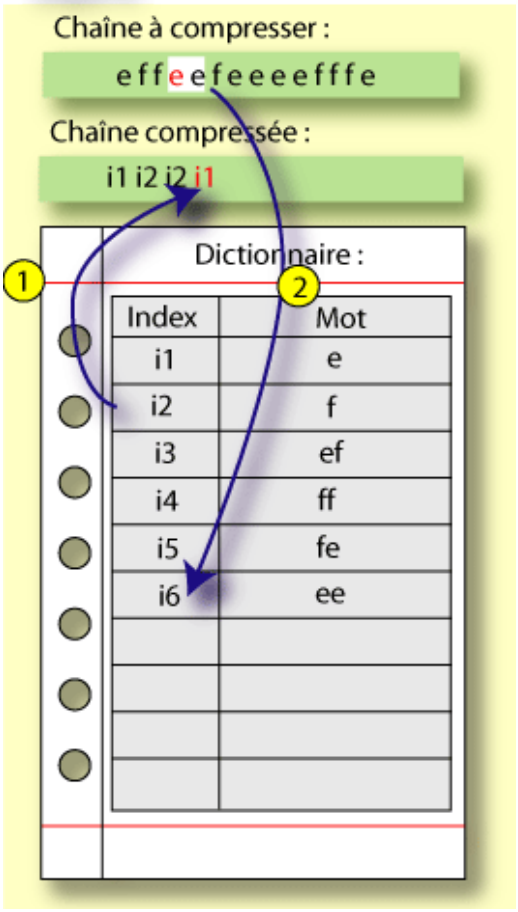
Etape 2 :



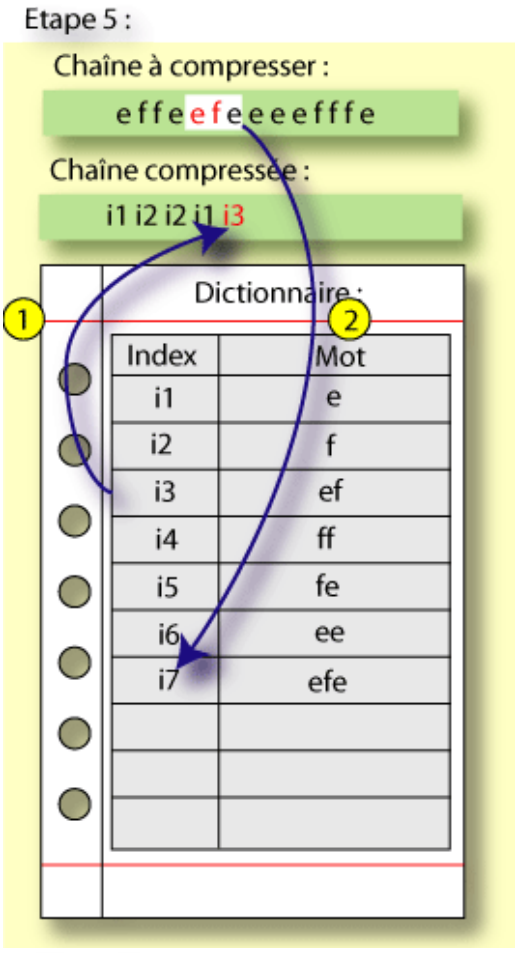
Etape 3 :



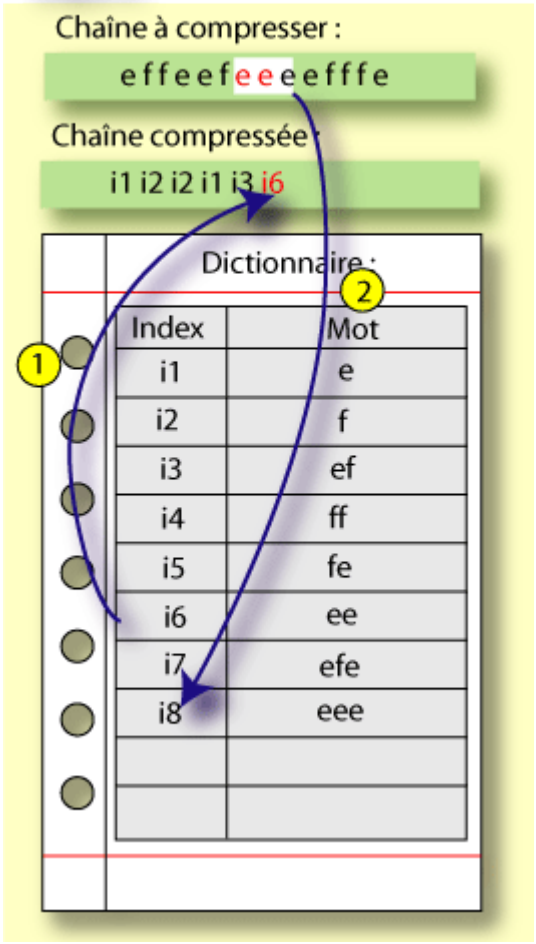
Etape 4 :



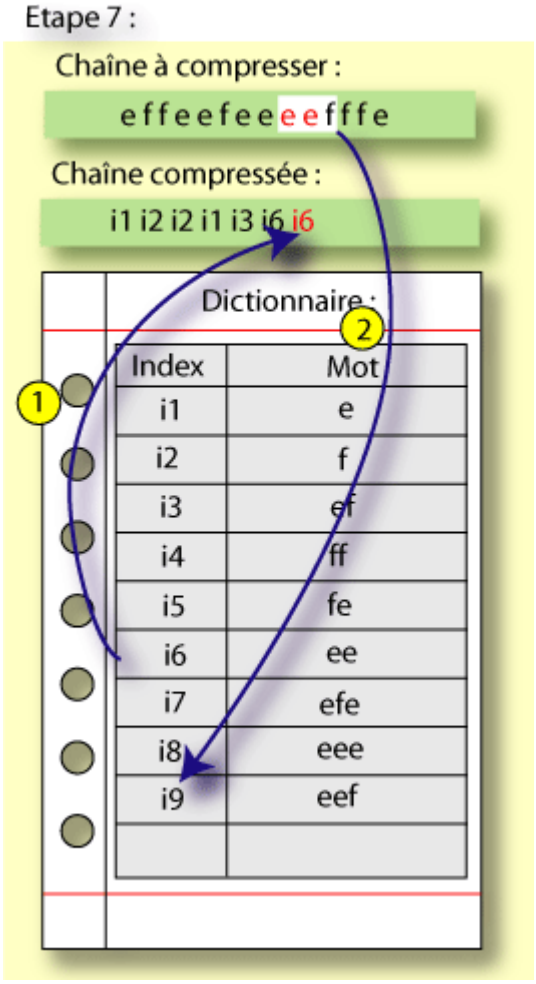
Etape 5 :

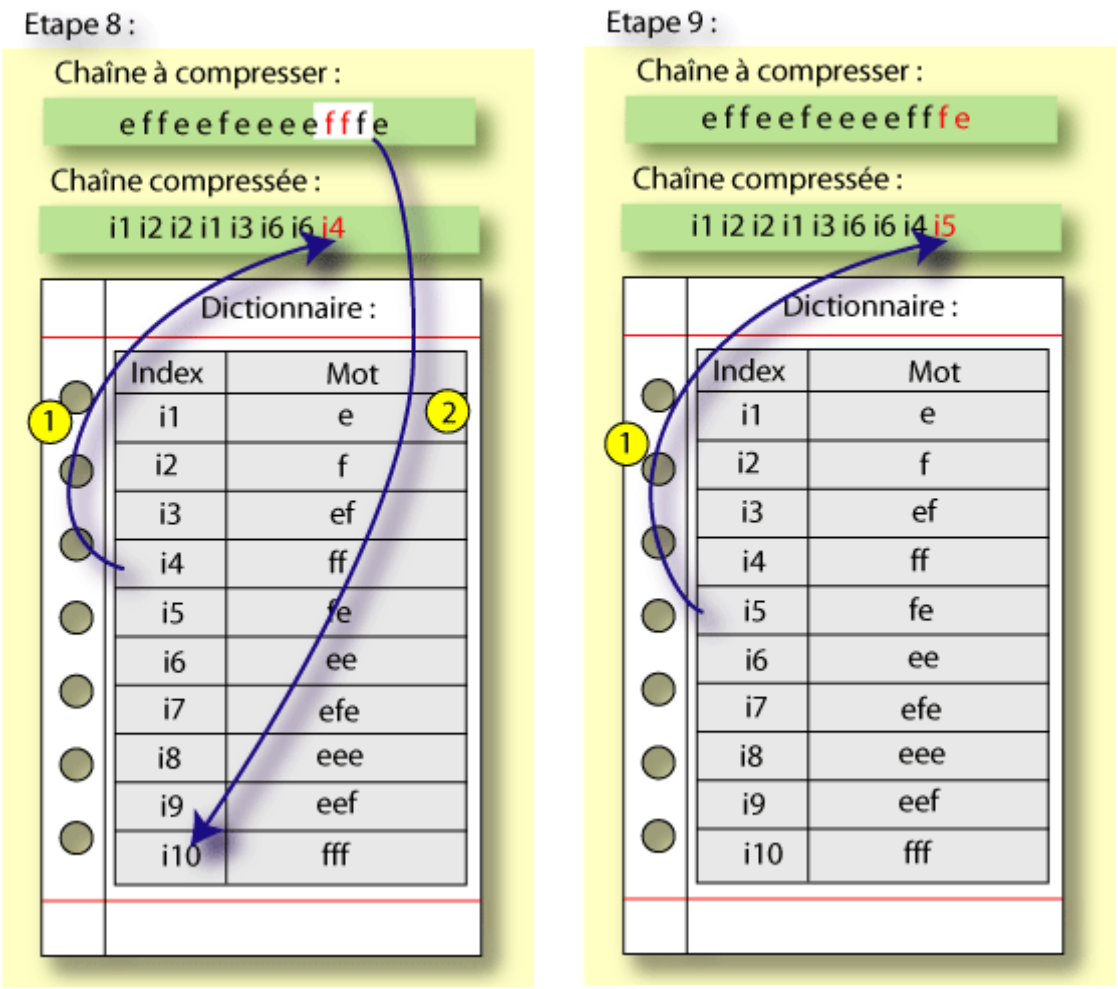


Etape 6 :



Etape 7 :





On obtient donc au final :

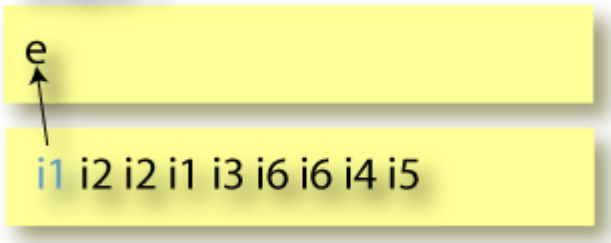
Chaîne à compresser : e f f e e f e e e e f f f e
Chaîne compressée : i1 i2 i2 i1 i3 i6 i6 i4 i5

Chaque indice représente un mot composé d'un ou plusieurs caractères. L'idéal est que chaque indice soit codé sur un octet voire moins.

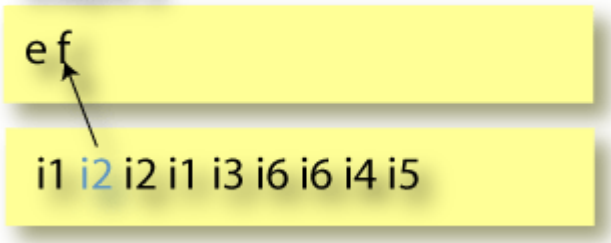
La décompression LZ78 :

Pour la décompression, il suffit d'inverser le processus en utilisant le dictionnaire et ses index. Si nous nous en tenons à cette présentation, on peut croire que le dictionnaire est construit dynamiquement quelque part. En réalité, le dictionnaire existe déjà puisqu'il s'agit de la chaîne décompressée.

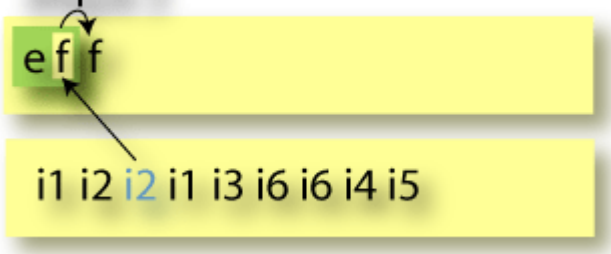
étape 1



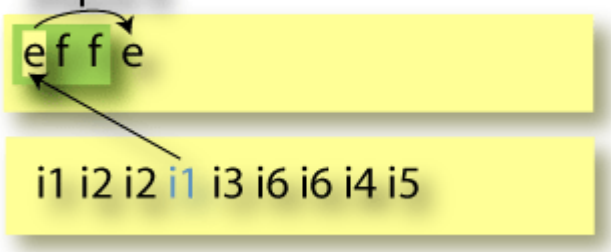
étape 2



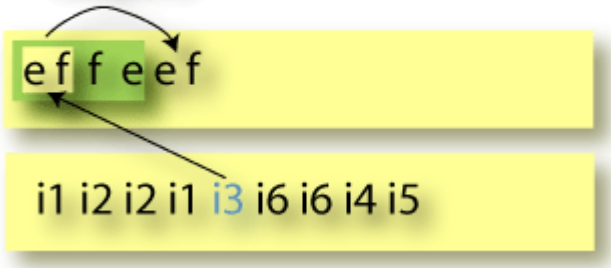
étape 3



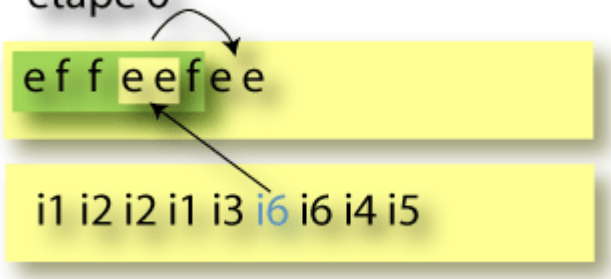
étape 4



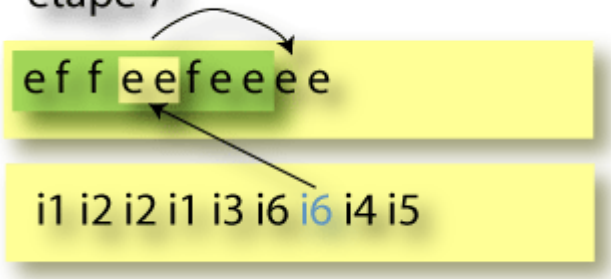
étape 5



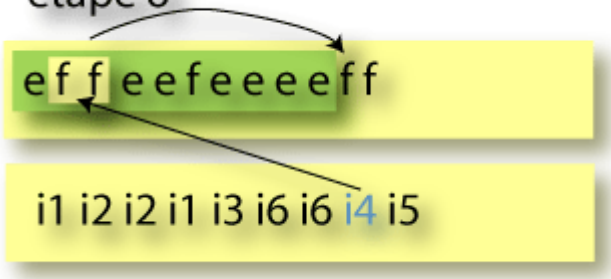
étape 6



étape 7



étape 8



Remarque sur les index :

Comment à l'étape 8 avec l'index i4 sommes nous capables de dire qu'il s'agit de la chaîne « ff » située juste après le premier caractère ?

Contrairement aux apparences, l'index i4 n'est pas un simple nombre. Il regroupe deux informations fondamentales à lui tout seul :

1) La position de la chaîne dans le dictionnaire.
2) La taille de cette chaîne.

Cependant, cet index doit être codé sur 1 octet (voire deux) pour espérer compresser quelque chose. Il ne nous intéresse donc pas en tant que valeur héxadécimale mais plutôt en tant que valeur binaire. Ce sont ses bits qui vont se comporter comme des drapeaux. On pourrait par exemple imaginer que les 4 bits de poids fort indiquent la position du mot et les 4 de poids faible la taille du mot.

Analyse du décompresseur de l'ApLib 0.34.

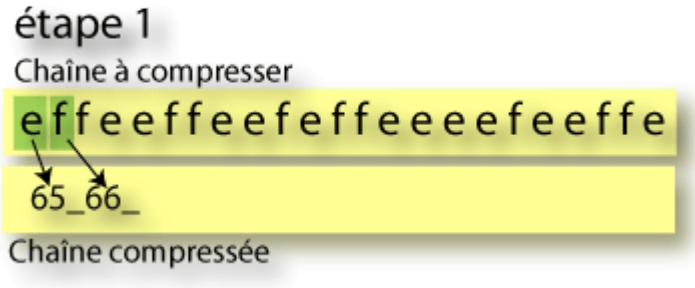
Algorithme très récent qui date de 2003, l'ApLib de Joergen Ibsen utilise le principe de la compression LZ78, à savoir, qu'il construit un dictionnaire au fur et à mesure de la compression des données qui servira ,à l'aide ses index, à la décompression. Néanmoins, il ne construit pas son dictionnaire comme LZ78.

Comment procède le compresseur ?

Pour répondre à cette question, prenons l'exemple suivant :

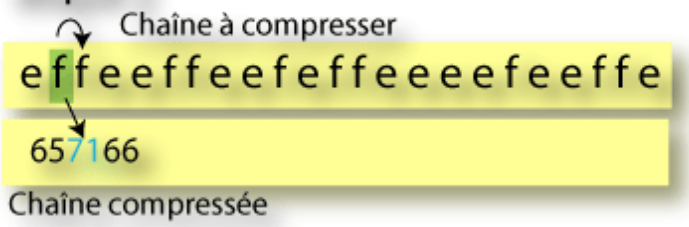
Chaîne à compresser :

e f f e e f f e e f e f f e e e e f e e f f e



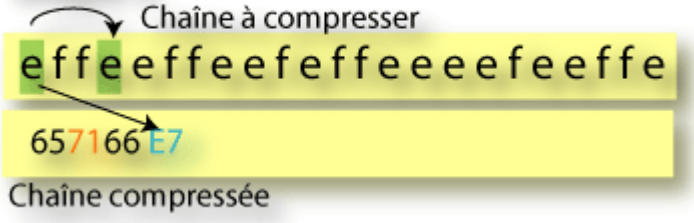
Il s'agit d'une phase d'initialisation. Il inscrit les caractères e et f au format UTF-16 dans la chaîne compressée.

étape 2



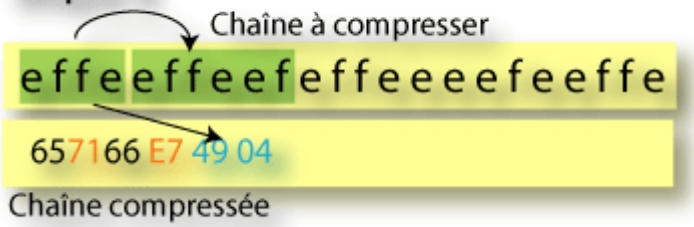
On scanne le dico à la recherche d'un mot déjà existant. On place l'index 71h qui indiquera au décompresseur la position relative et la taille du mot à copier. (nous verrons plus tard comment sont calculés ces index)

étape 3



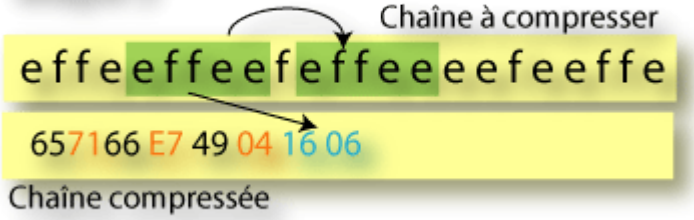
On place également l'index E7h qui indique la position -3 et la taille 1.

étape 4



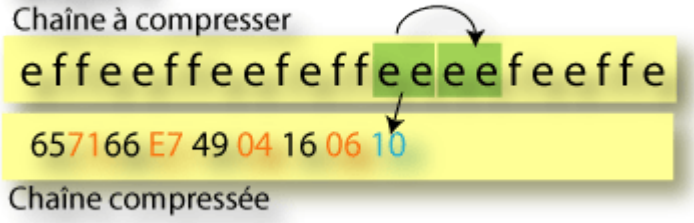
Le compresseur a repéré le motif «effe» répété 1,5 fois.Cette fois-ci, l'index tient sur 2 octets.

étape 5

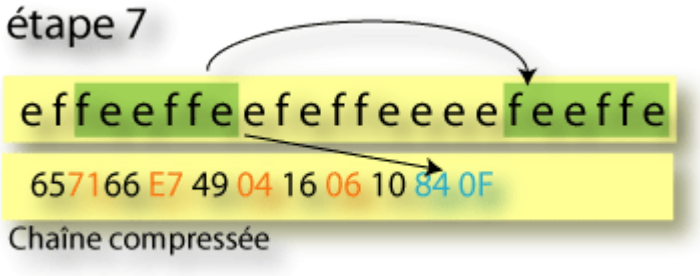


L'index tient également sur 2 octets.

étape 6



Si vous observez bien la chaîne de départ, au lieu de ne prendre que le mot «ee», il aurait pu prendre d'un coup «eefe». En fait, c'est bien ce que le compresseur a fait au départ mais, dans un souci d'optimisation avec le mot suivant, il préfère coder ce mot sur 1 octet et le suivant sur 2 plutôt que de coder celui-ci et le suivant sur 2 octets chacun.



Finalement,

Chaîne à compresser (en héxadécimal) :

65	66	66	65	65	66	66	65	65	66	65	66	66	65	65	65	65	66	65	65	66	66	65

Chaîne compressée (en héxadécimal) :

65	71	66	E7	49	4	16	6	10	84	0F												

Que se passe-t-il s'il n'y a pas de répétitions ?

C'est une situation improbable cependant elle illustre bien le fait que l'ApLib se base sur la répétition de motifs. Prenons, une chaîne sans répétition :

a b c d e f g h i j k l m n o

La chaîne compressée (en héxadécimal) est :

61	0	62	63	64	65	66	67	68	69	3	6A	6B	6C	6D	6E	6F

Le premier index signale au décompresseur de placer les 8 lettres suivantes et le second, les 6 dernières.

Nous n'avons clairement pas compressé la chaîne mais bien au contraire agrandi de 2 octets.

Contraintes.

Cet algorithme doir remplir trois fonctions primordiales :

1) Il doit décompresser les données en garantissant leur intégrité. (algorithme sans perte).

2) Il doit décompresser au plus vite en effectuant le moins de cycles possibles. Ceci nécessite donc d'optimiser le code.

3) Son code doit être le plus court possible pour ne pas trop surcharger l'exe.

Déroulement des opérations.

A chaque étape de la décompression, le décompresseur doit faire face à plusieurs cas de figure :

1) Si le caractère à décompresser ne se trouve pas déjà dans le dictionnaire, il faut le récupérer dans la zone compressée.

2) Si le caractère ou le mot à décompresser se trouve déjà dans le dictionnaire, il faut le récupérer dans ce dictionnaire en indiquant sa *position relative* et *sa taille*.

Optimisation du code :

L'étape 2 peut être déclinée en trois sous-étapes bien distinctes. Il s'agit en fait de gagner un maximum de temps durant les opérations de décompression. On considère donc ici que le caractère ou le mot à décompresser se trouve dans le dictionnaire.

2-1) Si le mot à décompresser est un même caractère répété plusieurs fois (exemple : eeee), ou si le mot n'excède pas 3 caractères, nous utiliserons une procédure optimisée « Shortmatch ».

2-2) Si le mot à décompresser est un caractère seul, nous utiliserons une procédure optimisée que Jibz appelle « GetMoreBits ».

2-3) Dans les autres cas, nous utiliserons la procédure de base que Jibz appelle « codepair ». En l'appelant ainsi, Jibz veut signifier que l'index utilisé pour le mot tient sur 2 octets au lieu d'un seul.

Cette dernière sous-étape 2-3 se décline à son tour en deux sous-sous-étapes :

2-3-1) La position du mot à placer est la même que le mot « précédent ». Il est donc inutile de la recalculer et on va donc juste calculer la taille.

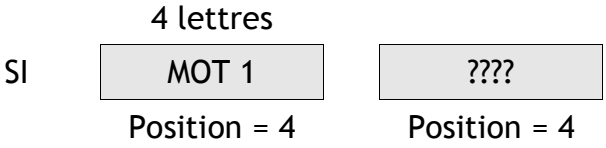
2-3-2) Il s'agit di cas général. On calcule la taille et la position dans la procédure « normalcodepair ».

L'étape 2-3-1 nécessite quelques explications. En analysant le code de Jibz, on comprend que certaines configurations ne peuvent jamais nous amener à l'étape 2-3-1. La question qui se pose donc est la suivante :

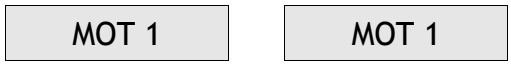
Peut-on récupérer une position déjà calculée pour le mot précédent ?

Démontrons par l'exemple que les positions de deux mots (d'au moins 4 lettres) qui se suivent sont forcément différentes.

Voici une situation imaginée. Un mot « MOT 1 » vient d'être placé en utilisant une position = 4.

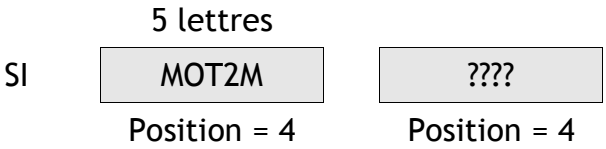


Le mot suivant peut-il avoir une position = 4 ? La réponse est clairement non car à ce moment nous aurions :



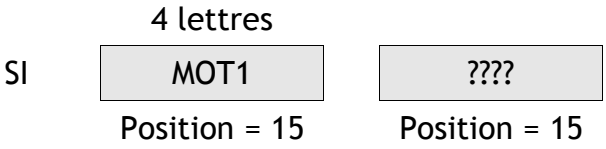
Cette répétition aurait été gérée avant.

Voici une autre situation :



Cette fois, le nouveau mot ne commence que par « OT2M » et on obtient « MOT2MOT2M » ce qui également une répétition qui est gérée par le mot précédent donc c'est impossible.

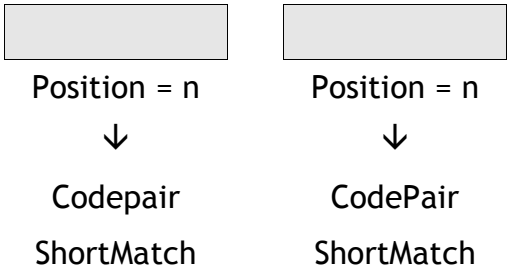
Dernière situation :



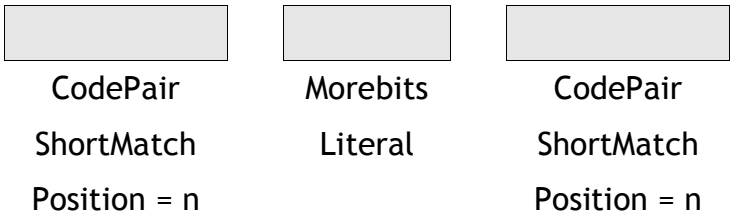
Ceci est encore impossible puisque l'on obtient encore une répétition du type MOT1MOT2.

Pour résumer :

Situation impossible :



Seule situation possible :

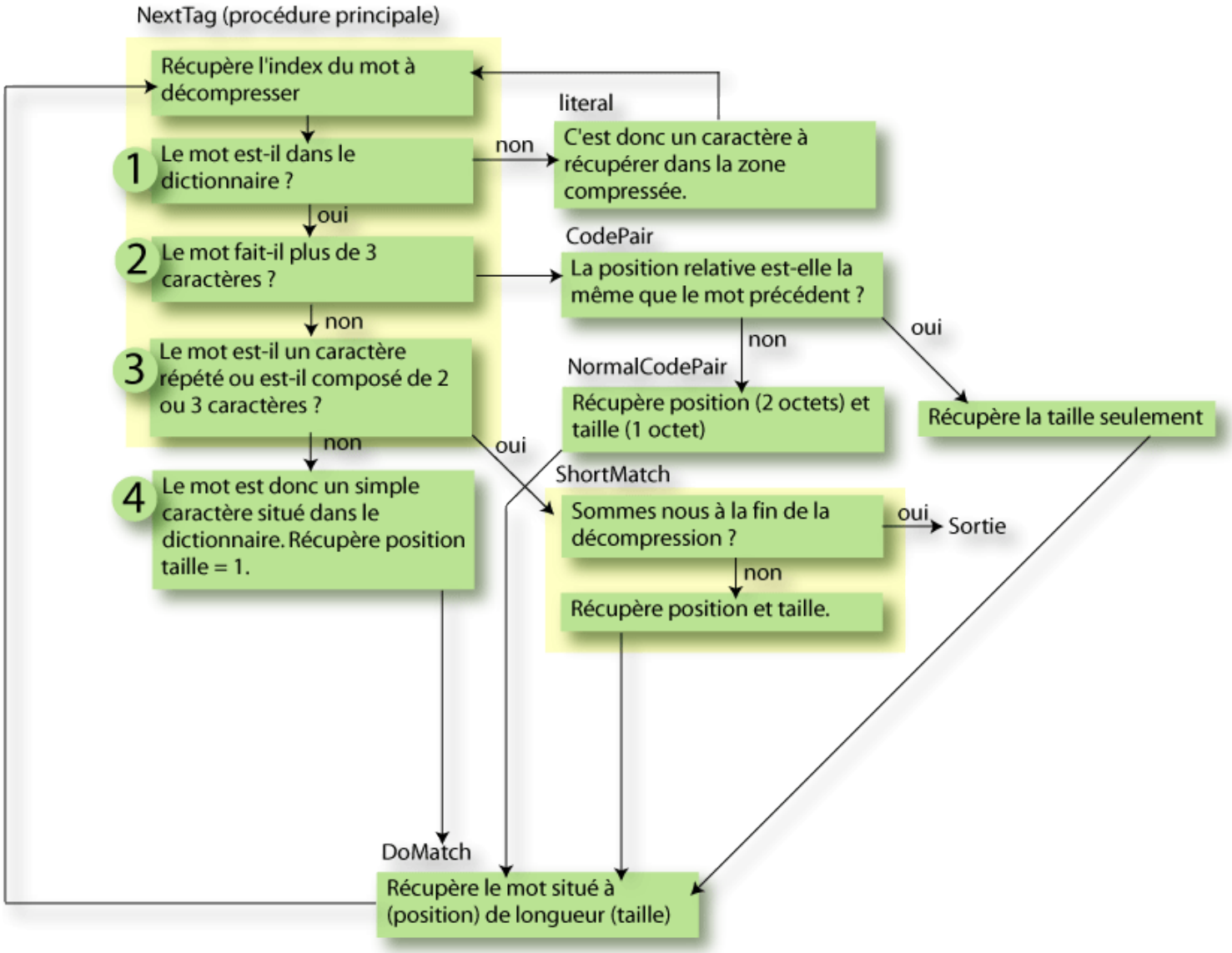


Pour différencier les deux situations, Jibz utilise un indicateur ebx.

Si ebx = 1, cela signifie que l'on est dans la situation impossible.

Si ebx = 2, nous sommes susceptible de récupérer la position du «codepair» précédent.

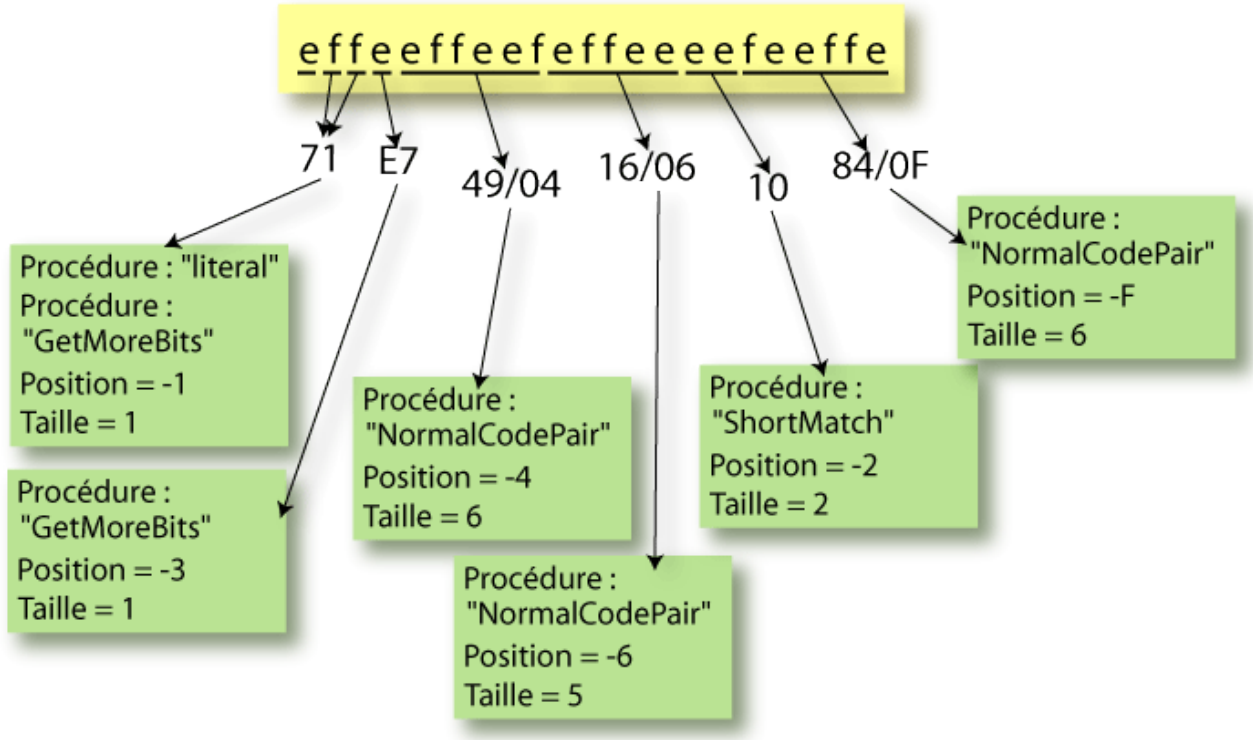
Voici un schéma qui illustre la procédure de décompression :



L'algorithme est composé d'un axe principal « NextTag – Domatch ». NetxTag accède à Domatch par l'intermédiaire de procédures annexes qui dépendent de la nature du mot à décompresser.

Comment le compresseur utilise-t-il les index de la chaîne compressée pour répondre à toutes ces questions et en même temps déterminer la position relative et la taille du mot à placer ?

Prenons la chaîne de départ :



Vous remarquez que chaque index indique au décompresseur la procédure à utiliser, la position relative et la taille du mot à placer.

● Prenons l'index 71h. Le décompresseur commence par le placer dans dl.

CF									
	0	1	1	1	0	0	0	1	dl = 71

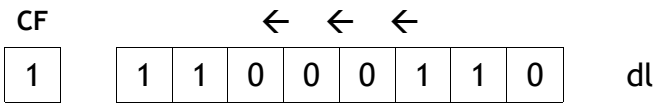
● Puis le décale d'un bit vers la gauche (en ajoutant une retenue)

CF	←	←	←						
0	1	1	1	0	0	0	1	1	dl

1) Le mot est-il dans le dictionnaire ?

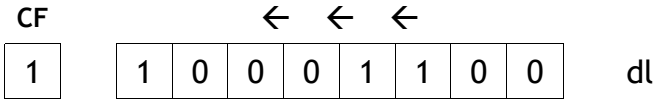
● Il teste pour cela la retenue CF. Ce premier zéro signifie que le prochain mot à placer est un caractère qui n'existe pas encore dans le dictionnaire. Il faut aller le chercher dans la chaîne compressée à la position courante en utilisant la procédure « literal ».

● A ce stade, nous avons placé le premier « f ». Le décompresseur redécale dl et recommence la procédure principale « NextTag ».



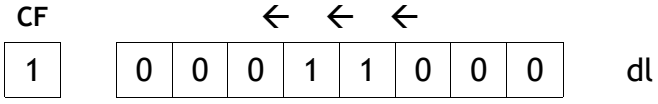
1) Le mot est-il dans le dictionnaire ?

● Là, la retenue (=1) signifie que le prochain mot est dans le dictionnaire.
● On redécale :



2) Le mot est-il un caractère répété ou est-il composé de 2 ou 3 caractères ?

● La retenue (=1) indique que la réponse est non.
● On redécale :



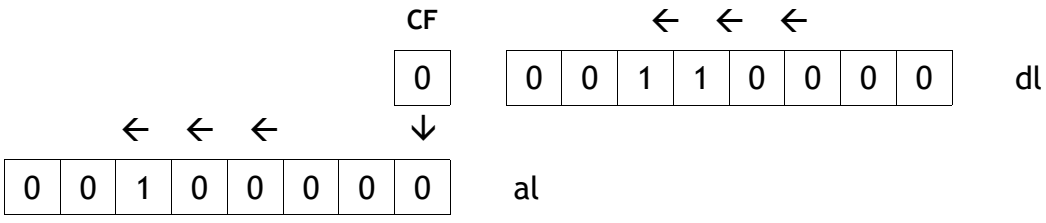
3) Le mot fait-il plus de trois caractères ?

● La retenue (=1) indique que la réponse est non.

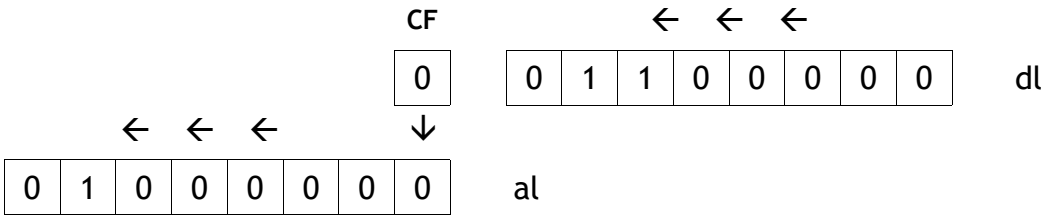
4) Le mot est donc un simple caractère du dictionnaire . (GetMoreBits)

Il ne reste plus qu'à récupérer sa position relative. al va servir de compteur de boucle (on l'initialise à 10h) et va en même temps servir à récupérer la position du mot.

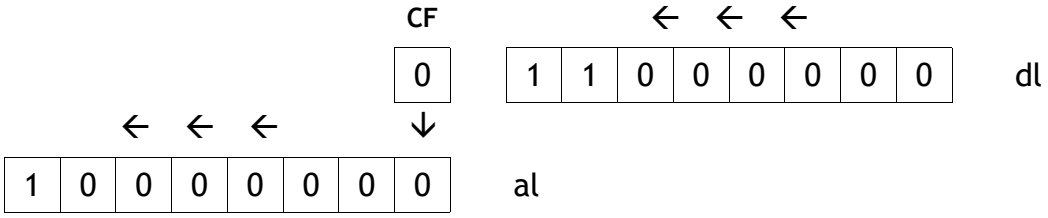
● On décale al et dl et on ajoute CF à al.



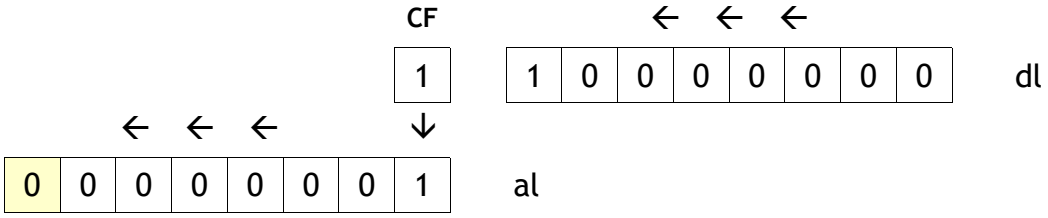
● On recommence :



● On recommence :



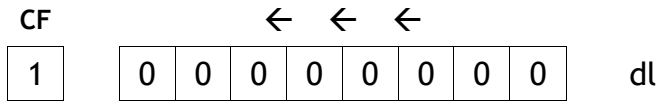
● On recommence :



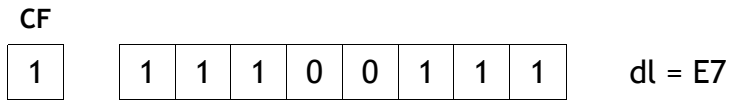
al signale la fin de la boucle. Il contient la position relative du mot, valeur à soustraire à la position actuelle pour obtenir la position absolue du mot. Ici, la lettre « f » est à la position -1.

Prenons maintenant l'index E7h :

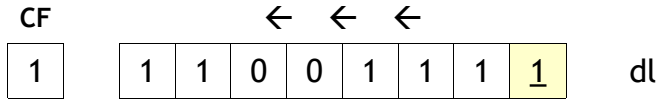
● D'abord, on décale dl :



● dl = 0 donc on récupère l'index suivant (E7) dans la chaîne compressée :



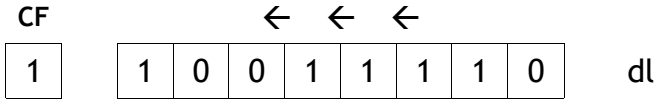
● On décale (et on ajoute la retenue) :



● *1) Le mot est-il dans le dictionnaire ?*

La retenue (=1) indique que la réponse est non.

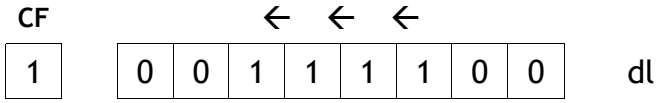
● On décale,



● *2) Le mot fait-il plus de 3 caractères ?*

La retenue (=1) indique que la réponse est non.

● On décale,

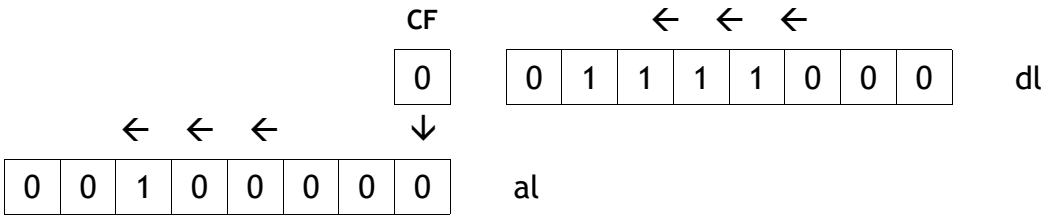


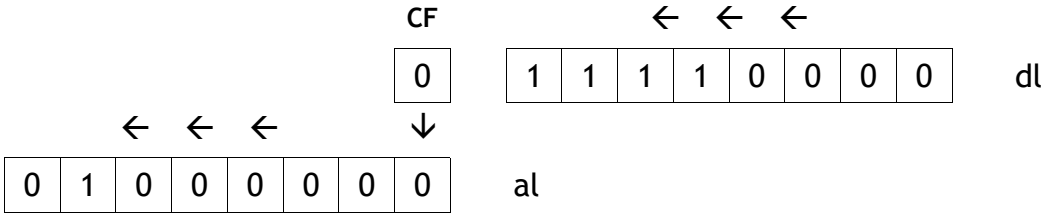
● *3) Le mot est-il un caractère répété ou est-il composé de 2 ou 3 caractères ?*

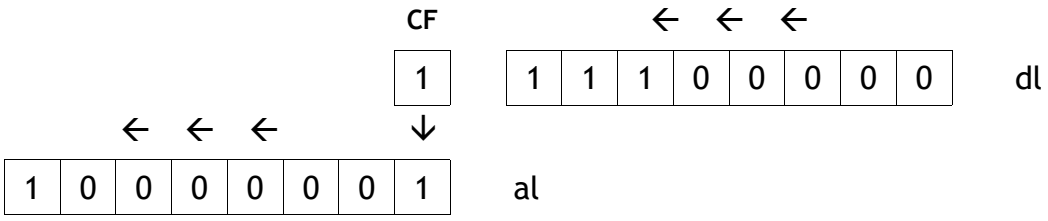
La retenue (=1) indique que la réponse est non.

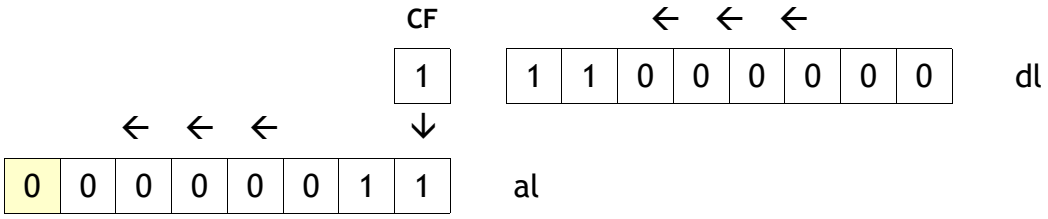
● *4) Le mot est donc un simple caractère.*

On récupère la position relative.



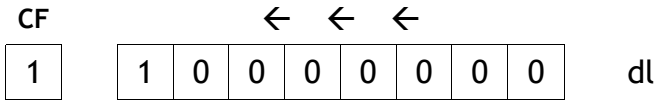




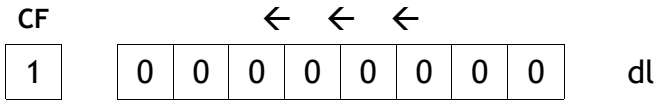


al = 3 donc la lettre «e» à récupérer se situe à la position relative -3 dans le dictionnaire.

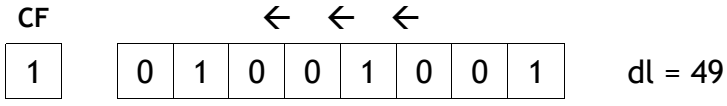
On recommence la boucle principale NextTag.
● On décale,



● *1) Le mot est-il dans le dictionnaire ?* : NON
● On décale,



dl = 0 donc on récupère l'index suivant (49) situé dans la chaîne compressée.



En fait, on ne récupère pas forcément les index en début de procédure NextTag. Il faut en réalité imaginer que les index sont mis bout à bout et forment ainsi une chaîne continue de 0 et de 1.

● On décale,

CF				←	←	←			
0	1	0	0	1	0	0	1	1	dl

● *2) Le mot fait-il plus de 3 caractères ?*

Carry étant à zéro, la réponse est oui. Nous allons alors dans la procédure « codepair ».

A ce stade, comme le mot précédent a été obtenu par la procédure « GetMoreBits », nous sommes susceptibles de pouvoir récupérer la position du mot précédent. Nous avons donc ebx = 2. Je ne rentre pas dans les détails mais dl est décalé plusieurs fois et les retenues sont récupérées dans ecx. Si ecx = ebx, alors nous pouvons récupérer la position précédente.

Dans notre cas, après décalages, nous avons :

CF				←	←	←			
0	0	1	0	0	1	1	0	0	dl
									ecx = 3

Ceci nous amène donc à la procédure NormalCodePair. On récupère la position dans eax en chargeant l'octet « 04 » situé dans la zone compressée. La taille est stockée dans ecx en effectuant à nouveau des décalages sur dl.

Peut-on patcher un exe packé par FSG ?

Sachant maintenant comment fonctionne l'ApLib, il est aisé de répondre à cette question. Prenons un exemple simple. Voici donc notre chaîne de départ (non compressée) que l'on veut patcher comme suit :

e	f	f	e	e	f	f	e	e	f	e	f	f	e	e	e	e	f	e	e	f	f	e
	↓																					
e	e	f	e	e	f	f	e	e	f	e	f	f	e	e	e	e	f	e	e	f	f	e

Nous ne voulons modifier qu'un seul octet en début de chaîne. Si nous compressons les deux chaînes précédentes, voilà ce que nous obtenons :

Compression de la première chaîne :

65 71 66 E7 49 4 16 6 10 84 0F

Compression de la deuxième chaîne :

65 E2 66 D0 7 4 45 6 19 0F 80

Le résultat, bien que prévisible, est sans appel : il faut quasiment modifier tous les octets qui suivent. En effet, la compression par dictionnaire fait que les octets en fin de chaîne sont codés en fonction des octets précédents. En modifiant un octet, on modifie du même coup le dictionnaire ! Le patch est donc réalisable mais pour un seul octet de la chaîne décompressée, on risque d'être obligé de patcher tout l'exe.

Analyse du code :

On commence par indiquer la section qui va contenir le code décompressé et celle qui contient le code compressé :

```
mov    esi, [esp + 36]      ; esi, section vide
mov    edi, [esp + 40]      ; edi, code compressé.

cld                         ; mise à zéro de CF
mov    dl, 80h              ; initialisation de dl.
xor    ebx, ebx             ; mise à zéro de ebx.
```

Voici la procédure « literal » :

```
literal:
   movsb                    ; copie d'un octet de edi vers esi
   mov    bl, 2             ; ebx ← 2
```

Et voilà la procédure principale « NextTag » :

```
nexttag:
   call   getbit
   jnc    literal           ; 1 ) Le mot est-il dans le dictionnaire ?

   xor    ecx, ecx          ; mise à zéro de ecx
   call   getbit
   jnc    codepair          ; 2 ) Le mot fait-il plus de 3 caractères ?

   xor    eax, eax          ; mise à zéro de eax
   call   getbit
   jnc    shortmatch        ; 3 ) Le mot fait-il 2 ou 3 caractères ?
```

Dans le cas où aucune condition n'est satisfaite, on tombe sur la procédure «GetMoreBits» : Le mot n'est composé que d'un seul caractère.

```
   mov    bl, 2                ebx ← 2
   inc    ecx                  ecx ← 1
   mov    al, 10h              compteur de boucle
getmorebits:
   call   getbit
   adc    al, al               décale al et stocke CF.
   jnc    getmorebits          boucle
   jnz    domatch
   stosb
   jmp    short nexttag
```

On trouve ensuite la procédure «codepair» :

```
codepair:
   call   getgamma_no_ecx
   sub    ecx, ebx
   jnz    normalcodepair       ; test pour savoir si on peut récupérer la
                               ; position du mot précédent.
   call   getgamma             ; récupère la taille du mot.
   jmp    short domatch_lastpos
```

La procédure «ShortMatch» :

```
shortmatch:
   lodsb                       ; récupère la position dans eax
   shr    eax, 1
   jz     donedepacking        ; A-t-on fini la décompression ?
   adc    ecx, ecx             ; décale ecx et récupère CF.
   jmp    short domatch_with_2inc
```

Voilà ensuite la procédure «normalcodepair» :

```
normalcodepair:
   xchg   eax, ecx             ; échange eax et ecx
   dec    eax                  ; eax ← eax – 1.
   shl    eax, 8               ; prépare la position sur deux octets
   lodsb                       ; récupère le deuxième octet.
   call   getgamma             ; récupère la taille
   cmp    eax, 32000
   jae    domatch_with_2inc        ; si eax > 7D00 alors  ecx ← ecx + 2
   cmp    ah, 5
   jae    domatch_with_inc         ; si 7D00 > eax >= 500 alors
                                   ; ecx ← ecx + 1
   cmp    eax, 7fh
   ja     domatch_new_lastpos      ; si 4FF > eax >= 100 alors
                                   ; ecx ← ecx sinon ecx ← ecx + 2
```

```
domatch_with_2inc:
   inc    ecx

domatch_with_inc:
   inc    ecx

domatch_new_lastpos:
   xchg   eax, ebp
domatch_lastpos:              ; Stocke la position du mot pour le mot
   mov    eax, ebp            ; ultérieur.

   mov    bl, 1               ; fameuse situation impossible
```

Voici la procédure «domatch» qui récupère un mot situé dans le dictionnaire.

```
domatch:
   push   esi                 ; sauvegarde esi
   mov    esi, edi
   sub    esi, eax            ; calcule la position du mot
   rep    movsb               ; récupère le mot
   pop    esi                 ; restaure esi
   jmp    short nexttag
```

La procédure «getbit» permet de faire le décalage sur dl :

```
getbit:
   add     dl, dl             ; décale dl d'un bit vers la gauche
   jnz     stillbitsleft
   mov     dl, [esi]          ; récupère l'octet suivant si dl = 0
   inc     esi
   adc     dl, dl             ; décale dl et lui ajoute CF
stillbitsleft:
   ret
```

La procédure getgamma récupère la longueur du mot :

```
getgamma:
   xor    ecx, ecx
getgamma_no_ecx:              optimisation pour gagner une instruction
   inc    ecx                 longueur du mot minimum = 1
getgammaloop:
   call   getbit
   adc    ecx, ecx            décale et récupère la retenue.
```

```
call   getbit
jc     getgammaloop
ret
```

```
donedepacking:
   sub    edi, [esp + 40]
   mov    [esp + 28], edi    ; return unpacked length in eax
```

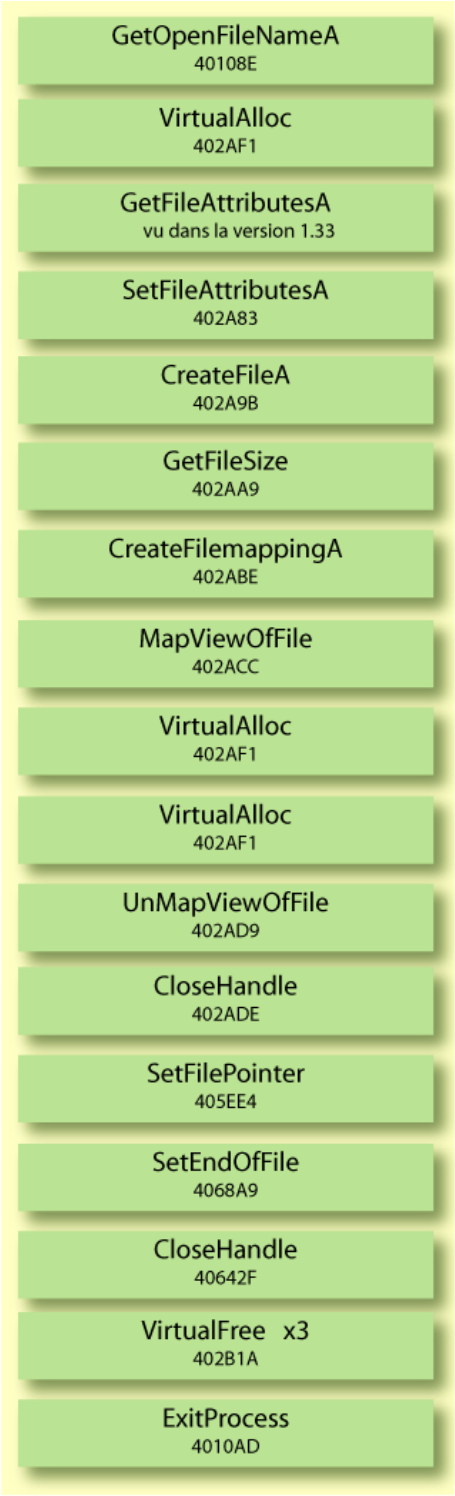
4 . Le packer FSG.EXE

Jusqu'ici, nous n'avons parlé que de la décompression. Cependant, on aurait pu voir le problème à l'envers en analysant l'algorithme de compression qui se trouve, pour sa part, dans le packer. J'ai donc voulu savoir comment fonctionnait ce dernier. A priori, sa fonction consiste à ouvrir l'exe à packer, compresser le code et ajouter le loader. En fait, le packer est protégé pour ralentir toute tentative d'analyse. Compression, CCA et OBFUSCATIONS sont au menu ! Rien de vraiment bien méchant mais c'est toujours intéressant d'étudier ce genre de protections.

Commençons, dans un souci de clareté, par présenter le packer comme s'il n'y avait aucune protection.

a . Schéma du packer version 2.0

Voici donc un schéma simplifié qui représente les différentes étapes nécessaires pour packer un exe avec FSG. Je présente les protections dans les paragraphes suivants.



On commence par choisir le fichier à compresser.

On modifie ses attributs pour le rendre modifiable.

On ouvre le fichier

On récupère sa taille

On crée un mappage en mémoire

On alloue une zone mémoire dans laquelle on compresse les imports.
On alloue une autre zone mémoire dans laquelle on compresse le code et les datas.
On détruit le mappage

On ferme le fichier

On libère la mémoire allouée

b . anti-debuggers

Le packer packé.

Le fichier FSG.exe est bien évidemment packé par FSG lui-même. Avant de pouvoir analyser ce packer, il est préférable de l'unpacker. Je ne m'attarde pas sur l'unpacking de FSG qui ne présente aucune difficulté. Vous pouvez faire le travail uniquement à partir d'OllyDbg si vous avez pris la peine d'installer le plugin OllyDump.

Dans la suite, je suppose donc le packer unpacké !

Les obfuscations.

Le code du loader est en fait perdu au milieu d'une multitude de JUMP. Tracer le code devient alors un peu délicat puisqu'il faut surveiller le code tout en avançant suffisamment vite pour ne pas rester trop longtemps sur du code inutile. Voici un extrait de ce genre d'obfuscations :

```
00401000    EB 02        JMP SHORT fsg_dump.00401004
00401002    65           DB 65                          ;  CHAR 'e'
00401003    10           DB 10
00401004    EB 02        JMP SHORT fsg_dump.00401008
00401006    65           DB 65                          ;  CHAR 'e'
00401007    8C           DB 8C
00401008    EB 03        JMP SHORT fsg_dump.0040100D
0040100A    C7           DB C7
0040100B    84           DB 84
0040100C    73           DB 73                          ;  CHAR 's'
0040100D    EB 02        JMP SHORT fsg_dump.00401011
0040100F    65           DB 65                          ;  CHAR 'e'
00401010    C6           DB C6
00401011    EB 02        JMP SHORT fsg_dump.00401015
00401013    0F           DB 0F
00401014    9C           PUSHFD
00401015    EB 02        JMP SHORT fsg_dump.00401019
00401017    CD 20        INT 20
00401019    C1F0 00      SAL EAX,0                      ;  Shift constant out of range 1..31
0040101C    9B           WAIT
0040101D    DBE3         FINIT
0040101F    EB 02        JMP SHORT fsg_dump.00401023
```

On ajoute à cela que les APIs sont enfermées dans des CALLS disséminés un peu partout. Il est alors préférable de poser des BPX sur toutes les APIs du packer pour comprendre son cheminement.

Le CCA (faiblesse de OllyDbg)

Pour afficher la fenêtre d'état de la décompression, FSG 1.33 et supérieur utilise une DialogBoxParamA (voir schéma plus haut). Cette dernière indique que sa procédure commence en 401D9E (pour la version 1.33). En analyse statique, 401D9E se présente, malgré l'analyse de Olly (CTRL + A), comme ceci :

```
00401D5C    84FF          TEST BH,BH
00401D5E    E9 93790000   JMP fsg_dump.004096F6  ; Cette ligne sert d'écran pour protéger le code
qui suit.
00401D63    EB            DB EB
00401D64    02            DB 02
00401D65    CD            DB CD
00401D66    20            DB 20                          ;  CHAR ' '
00401D67    EB            DB EB
00401D68    03            DB 03
00401D69    C7            DB C7
00401D6A    84            DB 84
00401D6B    29            DB 29                          ;  CHAR ')'
00401D6C    EB            DB EB
```

```
00401D6D    02          DB 02
00401D6E    65 6B       ASCII "ek"
00401D70    EB          DB EB
00401D71    02          DB 02
00401D72    CD          DB CD
00401D73    20          DB 20                   ; CHAR ' '
00401D74    EB          DB EB
00401D75    02          DB 02
00401D76    CD          DB CD
00401D77    20          DB 20                   ; CHAR ' '
00401D78    C1          DB C1
00401D79    F0          DB F0
00401D7A    00          DB 00
00401D7B    7C          DB 7C                   ; CHAR 'l'
00401D7C    03          DB 03
00401D7D    EB          DB EB
00401D7E    03          DB 03
00401D7F    63          DB 63                   ; CHAR 'c'
00401D80    74          DB 74                   ; CHAR 't'
00401D81    FB          DB FB
00401D82    EB          DB EB
00401D83    03          DB 03
00401D84    C7          DB C7
00401D85    84          DB 84
00401D86    9C          DB 9C
00401D87    EB          DB EB
00401D88    02          DB 02
00401D89    0F          DB 0F
00401D8A    BD          DB BD
00401D8B    EB          DB EB
00401D8C    02          DB 02
00401D8D    CD          DB CD
00401D8E    20          DB 20                   ; CHAR ' '
00401D8F    EB          DB EB
00401D90    02          DB 02
00401D91    0F          DB 0F
00401D92    7B          DB 7B                   ; CHAR '{'
00401D93    EB          DB EB
00401D94    02          DB 02
00401D95    0F          DB 0F
00401D96    6B          DB 6B                   ; CHAR 'k'
00401D97    C1          DB C1
00401D98    F0          DB F0
00401D99    00          DB 00
00401D9A    EB          DB EB
00401D9B    02          DB 02
00401D9C    65          DB 65                   ; CHAR 'e'
00401D9D    7B          DB 7B                   ; CHAR '{'
00401D9E    C8          DB C8                   ; début de la procédure de DialogParamBoxA
00401D9F    00          DB 00
00401DA0    00          DB 00
00401DA1    00          DB 00
00401DA2    53          DB 53                   ; CHAR 'S'
00401DA3    57          DB 57                   ; CHAR 'W'
```

Impossible de voir le code en clair ! L'astuce se trouve en 401D5E. Ce JUMP sert d'écran et protége le code qui suit d'une éventuelle analyse de Olly. Pour contourner cette protection, il suffit de nopper ce jump et de refaire une analyse du code (CTRL + A).

Comment générer un tel code ?

Voyons comment reproduire en asm ce genre d'obfuscations. Voici un petit programme que j'ai codé qui reproduit ces séries de JUMP inutiles ainsi que le CCA de la DialogBox :

```
.386
.model flat,stdcall
option casemap:none
include \masm32\include\windows.inc
include \masm32\include\kernel32.inc
includelib \masm32\lib\kernel32.lib
include \masm32\include\user32.inc
includelib \masm32\lib\user32.lib

.data
MsgBoxCaption     db "obfuscations style FSG",0
MsgBoxText        db "tests proposés par BeatriX ;)",0

.code
start:

; *************** Prépare le JMP EAX ******************

push $+3Ch
pop eax

; *************** Obfuscations *********************************

WORD 02EBh, 1065h, 02EBh, 20CDh, 02EBh,1065h, 02EBh, 20CDh
WORD 02EBh, 1065h, 02EBh, 20CDh, 02EBh,1065h, 02EBh, 20CDh
WORD 02EBh, 1065h, 02EBh, 20CDh, 02EBh,1065h, 02EBh, 20CDh

; ****************** CCA + Obfuscations ************************

jmp eax       ; <------Empêche OllyDbg de poser une étiquette en eax
WORD 42EBh    ; <------Piège qui leurre OllyDbg

WORD 1065h, 02EBh, 20CDh, 02EBh,1065h, 02EBh, 20CDh
push 0
push offset MsgBoxCaption
push offset MsgBoxText
WORD 02EBh, 1065h, 02EBh, 20CDh, 02EBh,1065h, 02EBh, 20CDh
push 0
WORD 02EBh, 1065h, 02EBh, 20CDh, 02EBh,1065h, 02EBh, 20CDh
Call MessageBox
WORD 02EBh, 1065h, 02EBh, 20CDh, 02EBh,1065h, 02EBh, 20CDh
push 0
Call ExitProcess

end start
```

On saisit les octets 1 par 1. Pour les obfuscations, on peut constater que la technique est puissante puisqu'elle permet de créer des centaines voire des milliers de sauts en quelques secondes (je suppose que Dulek a utilisé un procédé similaire avec tirage aléatoire des codes à afficher entre chaque JMP).

Cette méthode nous permet également d'avoir une plus grande souplesse pour la gestion du CCA. On peut ainsi organiser très facilement des sauts en plein milieu d'une instruction (c'est le cas du « piège ») !

Voici ce qu'on obtient sous OllyDbg :

```
00401000 >PUSH obfuscat.0040103C
00401005  POP EAX
00401006  JMP SHORT obfuscat.0040100A
00401008  DB 65                          ;  CHAR 'e'
00401009  DB 10
0040100A  JMP SHORT obfuscat.0040100E
0040100C  INT 20
0040100E  JMP SHORT obfuscat.00401012
00401010  DB 65                          ;  CHAR 'e'
00401011  DB 10
00401012  JMP SHORT obfuscat.00401016
00401014  INT 20
00401016  JMP SHORT obfuscat.0040101A
00401018  DB 65                          ;  CHAR 'e'
00401019  DB 10
0040101A  JMP SHORT obfuscat.0040101E
0040101C  INT 20
0040101E  JMP SHORT obfuscat.00401022
00401020  DB 65                          ;  CHAR 'e'
00401021  DB 10
00401022  JMP SHORT obfuscat.00401026
00401024  INT 20
00401026  JMP SHORT obfuscat.0040102A
00401028  DB 65                          ;  CHAR 'e'
00401029  DB 10
0040102A  JMP SHORT obfuscat.0040102E
0040102C  INT 20
0040102E  JMP SHORT obfuscat.00401032
00401030  DB 65                          ;  CHAR 'e'
00401031  DB 10
00401032  JMP SHORT obfuscat.00401036
00401034  INT 20
00401036  JMP EAX
00401038  JMP SHORT obfuscat.0040107C
0040103A  DB 65                          ;  CHAR 'e'
0040103B  DB 10
0040103C  DB EB                          ; il s'agit d'un JMP
0040103D  DB 02
0040103E  DB CD
0040103F  DB 20                          ;  CHAR ' '
00401040  DB EB
00401041  DB 02
00401042  DB 65                          ;  CHAR 'e'
00401043  DB 10
00401044  DB EB
00401045  DB 02
00401046  DB CD
00401047  ASCII " j",0
0040104A  DB 68                          ;  CHAR 'h'
0040104B  DD obfuscat.00403000               ;  ASCII "obfuscations style FSG"
0040104F  DB 68                          ;  CHAR 'h'
00401050  DD obfuscat.00403017
00401054  DB EB
00401055  DB 02
00401056  DB 65                          ;  CHAR 'e'
00401057  DB 10
00401058  DB EB
00401059  DB 02
0040105A  DB CD
0040105B  DB 20                          ;  CHAR ' '
```

```
0040105C  DB EB
0040105D  DB 02
0040105E  DB 65                           ;  CHAR 'e'
0040105F  DB 10
00401060  DB EB
00401061  DB 02
00401062  DB CD
00401063  ASCII " j",0
00401066  DB EB
00401067  DB 02
00401068  DB 65                           ;  CHAR 'e'
00401069  DB 10
0040106A  DB EB
0040106B  DB 02
0040106C  DB CD
0040106D  DB 20                           ;  CHAR ' '
0040106E  DB EB
0040106F  DB 02
00401070  DB 65                           ;  CHAR 'e'
00401071  DB 10
00401072  DB EB
00401073  DB 02
00401074  DB CD
00401075  DB 20                           ;  CHAR ' '
00401076  DB E8
00401077  DB 23                           ;  CHAR '#'
00401078  DB 00
00401079  DB 00
0040107A  ADD BL,CH
0040107C  ADD AH,BYTE PTR SS:[EBP+10]
0040107F  JMP SHORT obfuscat.00401083
00401081  INT 20
00401083  JMP SHORT obfuscat.00401087
00401085  DB 65                           ;  CHAR 'e'
00401086  DB 10
00401087  JMP SHORT obfuscat.0040108B
00401089  INT 20
0040108B  PUSH 0                           ; /ExitCode = 0
0040108D  CALL <JMP.&kernel32.ExitProcess>         ; \ExitProcess


00401092  JMP DWORD PTR DS:[<&kernel32.ExitProcess>];kernel32.ExitProcess
00401098  JMP DWORD PTR DS:[<&user32.wsprintfA>]    ;user32.wsprintfA
0040109E  JMP DWORD PTR DS:[<&user32.MessageBoxA>]  ;user32.MessageBoxA
```

Le code est lu par OllyDbg octet par octet même si on fait une analyse (CTRL + A). L'API MessageBox est «invisible» ainsi que ses paramètres. Le seul moyen d'obtenir du code clair est de nopper le jump en 40103B et de refaire une analyse.

IDA 4.51 se retrouve également complètement leurré au premier désassemblage !

```
[...]
.text:0040101D ; ---------------------------------------------------------------------------
.text:0040101D
.text:0040101D loc_40101D:                             ; CODE XREF: start+19#j
.text:0040101D                 jmp     short loc_401021
.text:0040101D ; ---------------------------------------------------------------------------
.text:0040101F                 db 0CDh, 20h
.text:00401021 ; ---------------------------------------------------------------------------
```

```
.text:00401021
.text:00401021 loc_401021:                             ; CODE XREF: start+1D#j
.text:00401021                 jmp     short loc_401025
.text:00401021 ; ---------------------------------------------------------------------------
.text:00401023                 db 65h, 10h
.text:00401025 ; ---------------------------------------------------------------------------
.text:00401025
.text:00401025 loc_401025:                             ; CODE XREF: start+21#j
.text:00401025                 jmp     short loc_401029
.text:00401025 ; ---------------------------------------------------------------------------
.text:00401027                 db 0CDh, 20h
.text:00401029 ; ---------------------------------------------------------------------------
.text:00401029
.text:00401029 loc_401029:                             ; CODE XREF: start+25#j
.text:00401029                 jmp     short loc_40102D
.text:00401029 ; ---------------------------------------------------------------------------
.text:0040102B                 db 65h, 10h
.text:0040102D ; ---------------------------------------------------------------------------
.text:0040102D
.text:0040102D loc_40102D:                             ; CODE XREF: start+29#j
.text:0040102D                 jmp     short loc_401031
.text:0040102D ; ---------------------------------------------------------------------------
.text:0040102F                 db 0CDh, 20h
.text:00401031 ; ---------------------------------------------------------------------------
.text:00401031
.text:00401031 loc_401031:                             ; CODE XREF: start+2D#j
.text:00401031                 jmp     short loc_401035
.text:00401031 ; ---------------------------------------------------------------------------
.text:00401033                 db 65h, 10h
.text:00401035 ; ---------------------------------------------------------------------------
.text:00401035
.text:00401035 loc_401035:                             ; CODE XREF: start+31#j
.text:00401035                 jmp     short loc_401039
.text:00401035 ; ---------------------------------------------------------------------------
.text:00401037                 db 0CDh, 20h
.text:00401039 ; ---------------------------------------------------------------------------
.text:00401039
.text:00401039 loc_401039:                             ; CODE XREF: start+35#j
.text:00401039                 jmp     eax
.text:00401039 start           endp
.text:00401039
.text:0040103B ; ---------------------------------------------------------------------------
.text:0040103B                 jmp     short loc_40107F
.text:0040103B ; ---------------------------------------------------------------------------
.text:0040103D                 dd 2EB1065h, 2EB20CDh, 2EB1065h, 6A20CDh, 40300068h,
30176800h
.text:0040103D                 dd 2EB0040h, 2EB1065h, 2EB20CDh, 2EB1065h, 6A20CDh,
106502EBh
.text:0040103D                 dd 20CD02EBh, 106502EBh, 20CD02EBh, 24E8h
.text:0040107D                 align 2
.text:0040107E                 db 0EBh
.text:0040107F ; ---------------------------------------------------------------------------
.text:0040107F
.text:0040107F loc_40107F:                             ; CODE XREF: .text:0040103B#j
.text:0040107F                 add     ah, [ebp+10h]
.text:00401082                 jmp     short loc_401086
.text:00401082 ; ---------------------------------------------------------------------------
.text:00401084                 db 0CDh, 20h
.text:00401086 ; ---------------------------------------------------------------------------
.text:00401086
.text:00401086 loc_401086:                             ; CODE XREF: .text:00401082#j
.text:00401086                 jmp     short loc_40108A
.text:00401086 ; ---------------------------------------------------------------------------
```

```
.text:00401088                  db 65h, 10h
.text:0040108A ; ---------------------------------------------------------------------------
.text:0040108A
.text:0040108A loc_40108A:                              ; CODE XREF: .text:00401086#j
.text:0040108A                  jmp     short loc_40108E
.text:0040108A ; ---------------------------------------------------------------------------
.text:0040108C                  db 0CDh, 20h
.text:0040108E ; ---------------------------------------------------------------------------
.text:0040108E
.text:0040108E loc_40108E:                              ; CODE XREF: .text:0040108A#j
.text:0040108E                  push    0
.text:00401090                  call    ExitProcess


.text:00401095                  int     3               ; Trap to Debugger
.text:00401096 ; [00000006 BYTES: COLLAPSED FUNCTION ExitProcess. PRESS KEYPAD "+" TO
EXPAND]
.text:0040109C ; ---------------------------------------------------------------------------
.text:0040109C                  jmp     ds:wsprintfA
.text:004010A2 ; ---------------------------------------------------------------------------
.text:004010A2                  jmp     ds:MessageBoxA
.text:004010A2 ; ---------------------------------------------------------------------------
.text:004010A8                  align 200h
.text:004010A8 _text            ends
```

WinDasm 9.0, quant à lui, ne se laisse pas faire ! Le listing est à peu près clair et le débuggage se fait presque normalement si on ne regarde que la fenêtre de debuggage. Dans la fenêtre de désassemblage (lors du debuggage), le curseur se bloque dès qu'il ne peut pas lire un code qu'il a analysé a priori. Ceci représente une réelle difficulté pour poser des BP.

Ainsi s'achève ce long et fastidieux travail sur FSG. J'espère que vous avez eu autant de plaisir à lire ce tuto que j'en ai eu à l'écrire. (même si je reconnais que j'en avais un peu ras le bol de l'ApLib ces dernières semaines).

5 . Remerciements

Tout d'abord, je tiens à féliciter Joergen Ibsen pour son remarquable travail contenu dans l'ApLib. L'algorithme est très bien pensé, très subtile, très astucieux. Un grand bravo. Merci à lui de nous livrer ce code et de nous laisser l'exploiter à volonté. En livrant une analyse du code de l'ApLib au grand jour, je considère que je rend hommage à Jibz à ma façon. Il ne s'agit pas ici de vous proposer des infos pour plagier ou voler son travail mais plutôt de révéler (de façon simple) son excellence. Encourageons-le à développer davantage cette compression en l'utilisant et en le lui faisant savoir.

Merci à Dulek pour avoir codé FSG.

Merci au CNDP qui propose sur le net une petite documentation sur la compression LZ.

De Gros bisous et un grand merci à tous les membres actifs de forumcrack pour leur compétence, leur gentillesse, leur patience (et leurs coups de gueules parfois) :)

Merci à tous ceux qui contribuent au développement de la connaissance en matière de cracking.

Vendredi 15 octobre 2004 – 19h 11min GMT- BeatriX