Outils utilisés

- IDA pro 5.4 démo
- Bochs
- du café

Méthode

- un coup de « file » sur le fichier : apparemment, le magic nous dit que c'est un grub.
- Je monte le fichier (-t ext2 -o loop) et le parcours : on a un fichier kernel.bin à la racine, et grub dans /grub. J'affiche le menu.lst : on est sensé booter sur kernle.bin. Le (fd0) m'indique que cela doit correspondre à une image disquette.
- Je charge le fichier dans Bochs et le démarre, je tombe sur le prompt pour rentrer la passphrase.
- Je réessaie bochs avec connecteur GDB, plug GDB dessus et « continue », mais le systeme a l'air de freezer. Surement une protection anti-gdb (mais je n'ai pas creusé le problème).
- En faisant quelques recherches sur le net, je vois sur le blog de hex-rays qu'une nouvelle version d'IDA pro 5.4 vient de sortir le jour même (coïncidence troublante...), et qu'elle inclut un connecteur Bochs pour pouvoir debugger des applis dans bochs.
- Je me jette dessus et constate les limitations suivantes :
  - utilisation limitée à 30 minutes
  - 496 commandes maximum lancées dans bochs depuis IDA
  - pas de sauvegarde

==> très contraignant, mais on fera avec..

- j'extrais kernel.bin de l'image, je renomme le fichier du challenge en kernel.bochsimg et créé un fichier de configuration bochs kernel.bochsrc (nécessaire pour faire fonctionner bochs avec IDA).
- Après lancement et au moment où est demandé la passphrase, j'interromps l'exécution pour regarder à quoi ressemble le binaire.

- on voit rapidement que l'on est limité à 16 caractères, et qu'après avoir entré notre passphrase, on jump en **0x200B94**.

Initialisation

- Dans cette fonction (**0x200B94**), l'élément principal est le « ***lldt ax*** » :
  - La **GDT** est positionnée à l'adresse **0x322288** (après le segment LOAD) : taille **0x1F**
  - **L'IDT** est positionnée à l'adresse **0x3222A8** (après la GDT) : taille **0x7FF**

```
<bochs> sregs
...
gdtr:base=0x00322288, limit=0x1f
idtr:base=0x003222a8, limit=0x7ff
```

  - On essaie de mettre 0x28 dans le registre ***ldtr*** (ax = 0x28). Cette valeur est trop grande (pointe en dehors de la gdt), une exception (**0xD**) est alors levée :

```
<bochs:32> BOCHS>step
(0).[280426696] [0x00200be8] 0008:0000000000200be8 (unk. ctxt): lldt ax                ; 0f00d0
00280426696e[CPU0 ] fetch_raw_descriptor: GDT: index (2f)5 > limit (1f)
CPU 0: Exception 0x0d - (#GP) general protection fault occured (error_code=0x0028)
CPU 0: Interrupt 0x0d occured (error_code=0x0028)
```

Si on regarde l'Interrupt Service Routine associée à l'interruption 0xD, on voit qu'elle pointe sur **0x2007C0**, et c'est donc là que l'on jump en executant le ***lldt***.

```
PHYSMEM:003222A8 db 0F0h,   6,   8,   0,   0, 8Eh, 20h,   0; 0
PHYSMEM:003222A8 db    0,   7,   8,   0,   0, 8Eh, 20h,   0; 8
PHYSMEM:003222A8 db  10h,   7,   8,   0,   0, 8Eh, 20h,   0; 16
PHYSMEM:003222A8 db  20h,   7,   8,   0,   0, 8Eh, 20h,   0; 24
PHYSMEM:003222A8 db  30h,   7,   8,   0,   0, 8Eh, 20h,   0; 32
PHYSMEM:003222A8 db  40h,   7,   8,   0,   0, 8Eh, 20h,   0; 40
PHYSMEM:003222A8 db  50h,   7,   8,   0,   0, 8Eh, 20h,   0; 48
PHYSMEM:003222A8 db  60h,   7,   8,   0,   0, 8Eh, 20h,   0; 56
PHYSMEM:003222A8 db  70h,   7,   8,   0,   0, 8Eh, 20h,   0; 64
PHYSMEM:003222A8 db  80h,   7,   8,   0,   0, 8Eh, 20h,   0; 72
PHYSMEM:003222A8 db  90h,   7,   8,   0,   0, 8Eh, 20h,   0; 80
PHYSMEM:003222A8 db 0A0h,   7,   8,   0,   0, 8Eh, 20h,   0; 88
PHYSMEM:003222A8 db 0B0h,   7,   8,   0,   0, 8Eh, 20h,   0; 96
PHYSMEM:003222A8 db 0C0h,   7,   8,   0,   0, 8Eh, 20h,   0; 104
PHYSMEM:003222A8 db 0D0h,   7,   8,   0,   0, 8Eh, 20h,   0; 112
PHYSMEM:003222A8 db 0E0h,   7,   8,   0,   0, 8Eh, 20h,   0; 120
```

Depuis cet **ISR**, on va sauter et appeler différentes fonctions pour “initialiser” le “contexte” de traitement de la passphrase :

- **0x1FFFD1** contient la passphrase (16 octets max)
- **0x60000** contient la passphrase initiale, copiée depuis **0x1FFFD1** (16 octets max)
- **0x80000** contient des constantes qui vont servir de clé plus tard. Ces constantes sont initialisées puis XORées avec la clé **0x1337C0DE** pour finalement avoir :

		apres XOR 1337C0DE
80000	EF	31h
80001	E7h	27h
80002	A9h	9Eh
80003	58h	4Bh
80004	38h	E6h
80005	7Bh	BB
80006	86h	B1h
80007	B9h	AA
80008	3Dh	E3h
80009	0Fh	CF
8000A	D3h	E4h
8000B	F7h	E4h
8000C	63h	BD
8000D	F6h	36h
8000E	9Dh	AA
8000F	48h	5Bh

- **0x326AE0** (sur environ 80 octets) va contenir des variables intermédiaires pour les différents calculs.

Après les différents appels des fonctions d'initialisation, on revient après le « **lldt ax** » et on continue l'exécution jusqu'à , entre autre, **0x15D9**.
A ce moment, on lève l'interruption **0xC0**, et on jump dans **l'ISR** défini à **IDTR** + (0xC0*8).
Après le retour de l'interruption (et quelques modifications sur la pile pour revenir EXACTEMENT à la même adresse), comme on est toujours à **0x15D9**, on re-lève l'interruption 0xC0.
Par contre, on exécute maintenant l'ISR situé à l'addrese **0x300** (**0xC0** * 4).
Entre les 2 interruptions **0xC0**, on se met à utiliser **l'IDT** situé à l'adresse **0x0,** au lieu de celle pointée par l'IDTR. (passage en mode réel?)

De manière générale, les ISR sont ensuite beaucoup utilisées pour modifier les valeurs successives de la passphrase en faisant des “int XX”.

Premier essai

La première modification de **0x60000** est faite en **0x1527** (int 21h levée juste après l'interrupt C0h, avec ISR de cette interruption qui est à **0x1527**) :

```
PHYSMEM:00001527 mov     bx, [si]
PHYSMEM:0000152B xchg    eax, esi
PHYSMEM:0000152C dec     dx
PHYSMEM:0000152E mov     ax, dx
PHYSMEM:00001531 and     ax, 3
PHYSMEM:00001535 xor     ax, bp
PHYSMEM:00001538 mov     ax, fs:[si]
PHYSMEM:0000153D xchg    ah, [esi+31h]
PHYSMEM:00001540 enter   5766h, 66h
PHYSMEM:00001544 push    ecx
PHYSMEM:00001545 push    bx
PHYSMEM:00001547 xor     di, bx
PHYSMEM:0000154A add     ax, di
PHYSMEM:0000154D shr     bx, 3
PHYSMEM:00001551 shl     cx, 4
PHYSMEM:00001555 xor     bx, cx
PHYSMEM:00001558 mov     di, bx
PHYSMEM:0000155B mov     bx, [si]
PHYSMEM:0000155F and     al, 67h
PHYSMEM:00001561 mov     cx, [esp+4]
PHYSMEM:00001566 shr     cx, 5
PHYSMEM:0000156A shl     bx, 2
PHYSMEM:0000156E xor     bx, cx
PHYSMEM:00001571 add     di, bx
PHYSMEM:00001574 xor     ax, di
PHYSMEM:00001577 pop     bx
PHYSMEM:00001579 pop     cx
PHYSMEM:0000157B pop     di
PHYSMEM:0000157D mov     cx, ax
PHYSMEM:00001580 mov     ax, [si]
PHYSMEM:00001584 xchg    eax, esi
PHYSMEM:00001585 add     cx, ax
PHYSMEM:00001588 mov     [si], cx
PHYSMEM:0000158C xchg    eax, esi
PHYSMEM:0000158D iret
```

C_2 = 4 octets de **0x80004** à **07** (2e bloc de 4 octets de la « clé »)
M_1 = 4 octets de **0x60000** à **03**
M_2 = 4 octets de **0x60004** à **07**
M_4 = 4 octets de **0x6000C** à **0F**

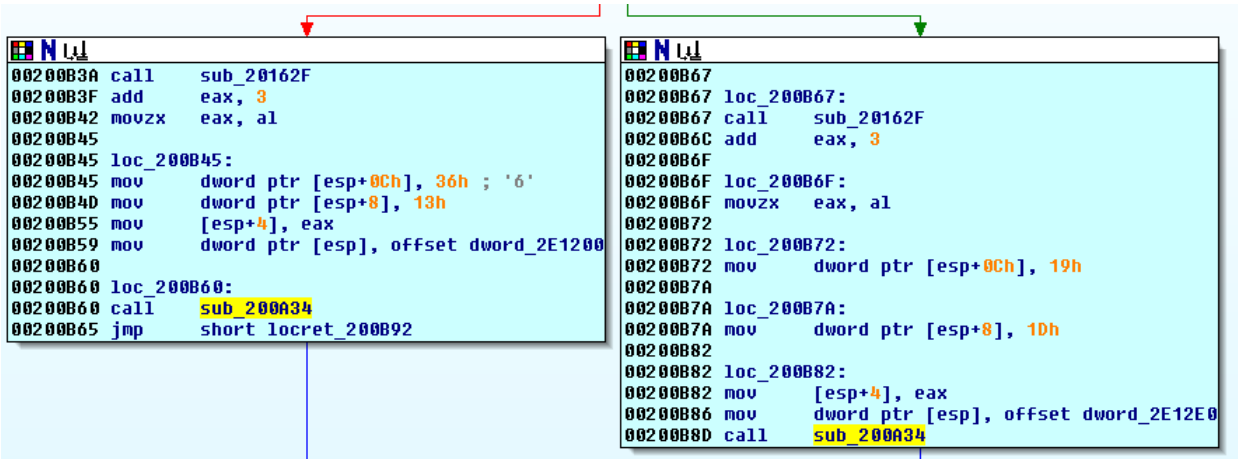
En analysant l'algrithme au dessus, on en déduit que :
M_1 reçoit :
M_1 + (((**M_4** ^ **C_2**) + (**0x7F434625** ^ **M_2**)) ^ ((**M_2**>>3)^(**M_4**<<4)+(**M_2**<<2)^(**M_4**>>5)))

A partir de là, je me suis dis que je ne pourrais pas reverser dans ce sens là, et j'ai donc décidé de partir de la fin.

Deuxième essai

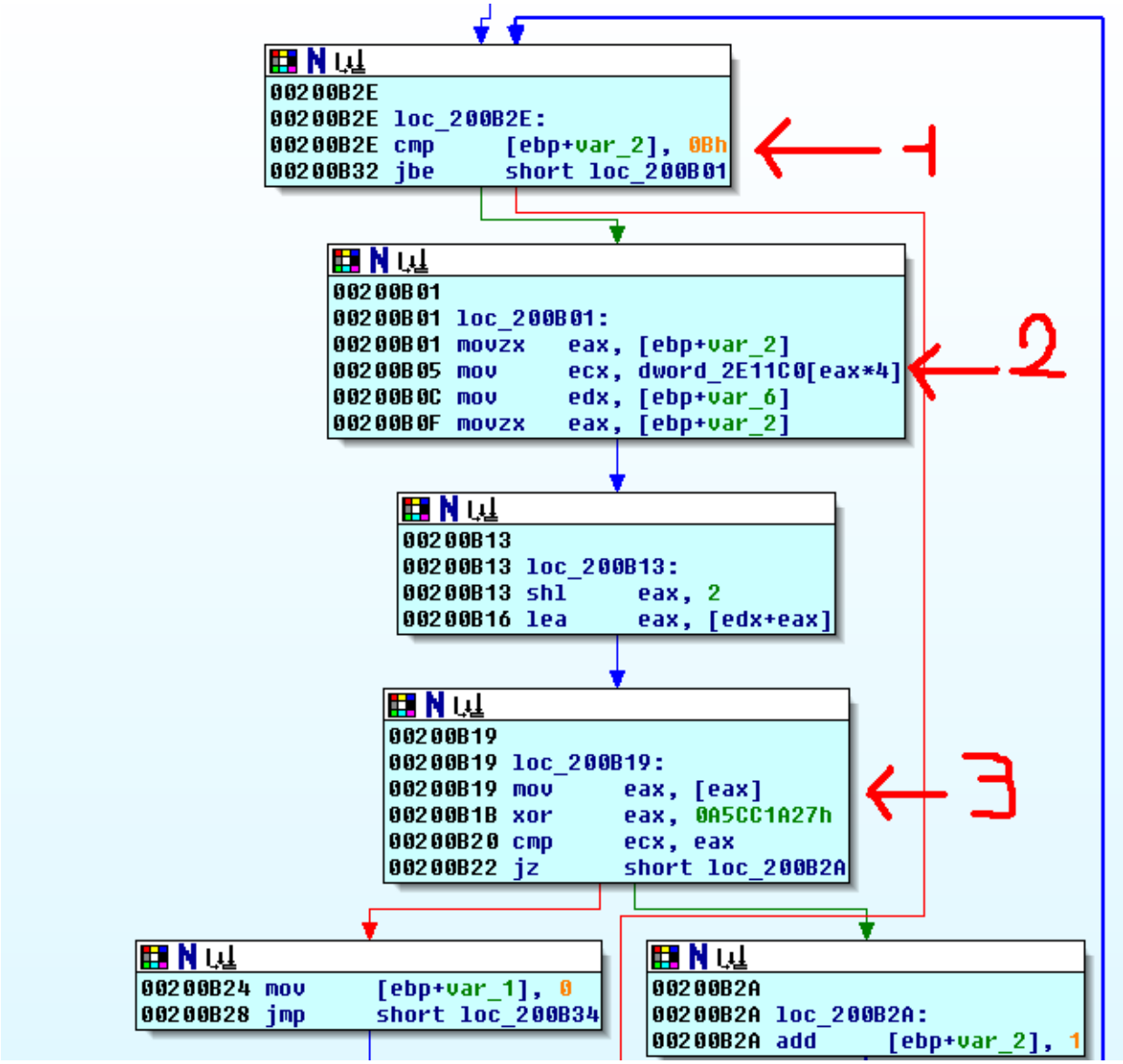
Pour ce faire, j'ai cherché la fonction d'affichage à l'écran.
En cherchant où les caractères « [« et «] » étaient affichés, on trouve que la fonction d'affichage est en **0x2019B3**.
En regardant où cette fonction est utilisée, on tombe sur la fonction **0x200A34**, qui elle même n'est appelée que depuis une seule fonction : **0x200AD5**.
En analysant cette fonction, on voit que :

- La fonction d'affichage (**0x200A34** qui elle appelle ensuite la vrai fonction) peut être appelée avec 2 messages différents : **FAIL**! (stocké en **0x2E12E0**) et **EXCELLENT**! (en **0x2E1200**).
-



Cette fonction va déchiffrer (un XOR, cadre de droite) le message (**FAIL** ou **EXCELLENT**) avec la clé **0xA5CC1A27**, l'afficher puis le rechiffrer (cadre de gauche).

- En remontant plus haut dans la fonction, on voit qu'une comparaison est effectuée pour savoir si on affiche FAIL ou EXCELLENT : c'est donc le résultat que l'on veut obtenir.



- En (1), on voit que l'on va comparer 12 blocs de 4 octets ==> le message que l'on veut obtenir fait 48 octets.
- En (2), on charge 4 octets du résultat de référence pour comparaison (stocké en **0x2E11C0**)
- En (3), on charge 4 octets de notre message (stocké en **0x70000**), on XOR avec **0xA5CC1A27**, et on compare.
- En cas de réussite, on passe au bloc de 4 octets suivants.

A partir de là, on connait les 48 octets que l'on doit obtenir au terme des différentes transformations (stockés en **0x70000**), il s'agit des 48 octets stockés en **0x2E11C0** XORés avec **0xA5CC1A27**.

Reversing du deuxième algortihme

En posant des breakpoints sur la mémoire à **0x70000** et sur les octets suivants, on voit que les blocs sont modifiés 2 octets par 2 octets.
2 opérations successives sont réalisées sur chacun de ces blocs :

- une initialisation (voir plus loin)
- un « byte swap », puis un « not »

Pour connaître la valeur du bloc après l'initialisation, il suffit donc de :

- faire un xor du bloc « résultat de référence » avec **0xA5CC** ou **0x1A27**
- faire un not
- faire un byte swap

	XOR	**resultat**	**not**	**bswap**	
CB 3C	0xA5CC	6EF0	910F	0F91	**70002-3**
6D C1	0x1A27	77E6	8819	1988	**70000-1**
7D 35	0xA5CC	D8F9	2706	0627	**70006-7**
CC CD	0x1A27	D6EA	2915	1529	**70004-5**
3E 3A	0xA5CC	9BF6	6409	0964	**7000A-B**
55 DF	0x1A27	4FF8	B007	07B0	**70008-9**
EF 32	0xA5CC	4AFE	B501	01B5	**7000E-F**
AB DB	0x1A27	B1FC	4E03	034E	**7000C-D**
3F 29	0xA5CC	9AE5	651A	1A65	**70012-13**
F1 E3	0x1A27	EBC4	143B	3B14	**70010-11**
7B 03	0xA5CC	DECF	2130	3021	**70016-17**
A6 C7	0x1A27	BCE0	431F	1F43	**70014-15**
F3 38	0xA5CC	56F4	A90B	0BA9	**7001A-1B**
CF DA	0x1A27	D5FD	2A02	022A	**70018-19**
02 30	0xA5CC	A7FC	5803	0358	**7001E-1F**
97 FD	0x1A27	8DDA	7225	2572	**7001C-1D**
89 31	0xA5CC	2CFD	D302	02D3	**70022-23**
01 E0	0x1A27	1BC7	E438	38E4	**70020-21**
3D 0F	0xA5CC	98C3	673C	3C67	**70026-27**
7E DE	0x1A27	64F9	9B06	069B	**70024-25**
D7 02	0xA5CC	72CE	8D31	318D	**7002A-2B**
79 D8	0x1A27	63FF	9C00	009C	**70028-29**
06 33	0xA5CC	A3FF	5C00	005C	**7002E-2F**
B5 DE	0x1A27	AFF9	5006	0650	**7002C-2D**

Dans les fonctions d'initialisation (des blocs en **0x700XX**), on utilise des blocs de 2 octets aux adresses **0x6000X** (de 0 à F, on a 8 blocs de 2 octets).
Il y a 2 types de fonction d'initialisation :

- **bloc en 0x700XX** = (**bloc_en_0x6000X**) modulo (**constante**)
- **bloc en 0x700XX** = (**bloc_en_0x6000X**) – (((**bloc_en_0x6000X**) * **a1**)>>**a2**) * **a3**
  (avec **a1**, **a2** et **a3** des constantes propres à chaque fonction)

Les blocs en **0x6000X** sont utilisés plusieurs fois (3 fois chaque) pour initialiser différents blocs en **0x700XX**.
Par exemple, on a donc :
bloc_0x70000 = bloc_0x60000 modulo **constante_1**
bloc_0x70010 = bloc_0x60000 modulo **constante_2**
bloc_0x70020 = bloc_0x60000 - (((bloc_en_0x6000X) * a1)>>a2) * a3

1 bloc de 2 octets en **0x6000X** va permettre d'initialiser 3 blocs de 2 octets en **0x700XX**.
On passe donc d'un message de 16 octets à 48 octets.
Grâce au théorème des restes chinois, on va pouvoir calculer **bloc_0x60000.**

Certains blocs en **0x6000X** ne sont utilisés qu'une seule fois avec la méthode du modulo (et 2 fois avec la méthode multiplication, décalage, multiplication) ou pas du tout avec cette méthode (et 3 fois avec l'autre méthode). Dans ce cas, on va bruteforcer la valeur du bloc (étant sur 2 octets, cela prend moins d'une seconde) et vérifier quelle valeur vérifie les 3 conditions (les 3 modulos et/ou autre méthode).

	reste1	modulo1	reste2	modulo2	resultat
60000-60001	1988	34F6	38E4	3FC3	B86A
60002-60003	0F91	23BF			9E8D
60004-60005	1529	15D0	1F43	2693	6C69
60006-60007	0627	6423	3C67	4903	CE6D
60008-60009	07B0	0EC6	009C	125	253C
6000A-6000B	0964	971	318D	459D	BCC7
6000C-6000D			0650	0ED5	D5F6
6000E-6000F					196C

Ainsi, les blocs **60002-03**, **6000C-0D** et **6000E-0F** ont du être bruteforcés.
Les 5 autres blocs ont été calculés par le théorème des restes chinois.

Après cette étape, on connait les valeurs que doivent avoir les 16 octets à l'adresse **0x60000**.

60000	6A
60001	B8
60002	8D
60003	9E
60004	69
60005	6C
60006	6D
60007	CE
60008	3C
60009	25
6000A	C7
6000B	BC
6000C	F6
6000D	D5
6000E	6C
6000F	19

Reversing du premier algorithme

Convention pour cette partie :

C_1 = 4 octets de **0x80000** à **03 = 0x4B9E2731**
C_2 = 4 octets de **0x80004** à **07 = 0xAAB1BBE6**
C_3 = 4 octets de **0x80008** à **0B = 0xE4E4CFE3**
C_4 = 4 octets de **0x8000C** à **0F = 0x5BAA36BD**

M_1 = 4 octets de **0x60000** à **03**
M_2 = 4 octets de **0x60004** à **07**
M_2 = 4 octets de **0x60008** à **0B**
M_4 = 4 octets de **0x6000C** à **0F**

M_1_s = valeur de M_1 au tour de boucle précédent
M_2_s = valeur de M_2 au tour de boucle précédent
M_3_s = valeur de M_3 au tour de boucle précédent
M_4_s = valeur de M_4 au tour de boucle précédent

En breakant sur les modifications de ces zones mémoires (**0x60000** à **0F**), on voit qu'elles sont modifiées 19 fois chacunes (par bloc de 4 octets).
On peut donc analyser les 4 dernières modifications des blocs, et en déduire l'algorithme suivant, pour passer du tour 18 au 19 (rappel : on part de la fin, donc on a **M_1** à **M_4**, et on cherche **M_1_s** à **M_4_s**)

Pour M_1_s:

M_1 = **M_1_s** + ((**C_1** ^ **M_4_s**) + (**M_2_s** ^ **0x693FA863**)) ^ ((**M_2_s**>>3) ^(**M_4_s**<<4)) + ((**M_2_s**<<2) ^(**M_4_s**>>5))
d'où
M_1_s = **M_1** - ((**C_1** ^ **M_4_s**) + (**M_2_s** ^ **0x693FA863**)) ^ ((**M_2_s**>>3) ^(**M_4_s**<<4)) + ((**M_2_s**<<2) ^(**M_4_s**>>5))

==> Pour calculer **M_1_s**, besoin au préalable de **M_2_s** et **M_4_s**

Pour M_2_s:

M_2 = **M_2_s** +((**C_2** ^ **M_1**) + (**M_3_s** ^ **0x693FA863**)) ^ (((**M_3_s**>>3) ^(**M_1**<<4)) + ((**M_3_s**<<2) ^(**M_1**>>5)))
d'où
M_2_s = **M_2** - ((**C_2** ^ **M_1**) + (**M_3_s** ^ **0x693FA863**)) ^ (((**M_3_s**>>3) ^(**M_1**<<4)) + ((**M_3_s**<<2) ^(**M_1**>>5)))

==> Pour calculer **M_2_s**, besoin au préalable de **M_3_s**

Pour M_3_s:

M_3 = **M_3_s** + ((**C_3** ^ **M_2**) + (**M_4_s** ^ **0x693FA863**)) ^ (((**M_4_s**>>3) ^ (**M_2**<<4)) + ((**M_4_s**<<2) ^ (**M_2**>>5)))
d'où
M_3_s = **M_3** - ((**C_3** ^ **M_2**) + (**M_4_s** ^ **0x693FA863**)) ^ (((**M_4_s**>>3) ^ (**M_2**<<4)) + ((**M_4_s**<<2) ^ (**M_2**>>5)))

==> Pour calculer **M_3_s**, besoin au préalable de **M_4_s**

Pour M_4_s:

M_4 = [0x326AE0 + 24h] + M_4_s
d'où
M_4_s = M_4 - [0x326AE0 + 24h]

[0x326AE0 + 24h] est une zone mémoire contenant une valeur calculée juste avant le calcul de **M_4**:

[326AE0 + 24h] = ((**M_3** ^ **C_4**) + (**M_1** ^ **0x693FA863**)) ^ ((**M_1** >>3) ^ (**M_3** << 4) + (**M_1**<<2) ^ (**M_3** >> 5))

On en déduit donc l'algorithme suivant pour passer du tour 19 au tour 18 :

on calcule :
[326AE0 + 24h] = ((**M_3 ^ C_4**) + (**M_1 ^ 0x693FA863**)) ^ ((**M_1** >>3) ^ (**M_3** << 4) + (**M_1**<<2) ^ (**M_3**>> 5))
M_4_s = **M_4 - [0x326AE0 + 24h]**

on calcule :
TMP = ((**C_3 ^ M_2**) + (**M_4_s ^ 0x693FA863**)) ^ (((**M_4_s**>>3) ^ (**M_2**<<4)) + ((**M_4_s**<<2) ^ (**M_2**>>5)))
M_3_s = **M_3 - TMP**

on calcule :
TMP = ((**C_2 ^ M_1**) + (**M_3_s ^ 0x693FA863**)) ^ (((**M_3_s**>>3) ^(**M_1**<<4)) + ((**M_3_s**<<2) ^(**M_1**>>5)))
M_2_s = **M_2 - TMP**

on calcule :
TMP = ((**C_1 ^ M_4_s**) + (**M_2_s ^ 0x693FA863**)) ^ ((**M_2_s**>>3) ^(**M_4_s**<<4)) + ((**M_2_s**<<2) ^(**M_4_s**>>5))
M_1_s = **M_1 - TMP**

Après vérification (écriture des valeurs trouvées dans la mémoire à **0x6000X**, le programme m'affiche bien “excellent!” à la fin), l'algorithme est validé.

Par contre, en le lancant 19 fois consécutives pour calculer les valeurs de M_1 _ M_4 à chaque tour, le résultat final n'est pas bon.
En investiguant un peu, je remarque les choses suivantes :
dans l'algorithme que je viens d'écrire, j'ai:
M_1 = **M_1_s** + ((**C_1** ^ **M_4_s**) + (**M_2_s** ^ **0x693FA863**)) ^ ((**M_2_s**>>3) ^(**M_4_s**<<4)) + ((**M_2_s**<<2) ^(**M_4_s**>>5))

et dans l'algorithme que j'ai reversé dans "**premier essai**", j'ai :

M_1 = **M_1_s** + (((**C_2** ^ **M_4_s**)+ (**M_2_s** ^ **0x7F434625**)) ^ ((**M_2_s**>>3)^(**M_4_s**<<4)+(**M_2_s**<<2)^(**M_4_s**>>5)))

On voit donc que, selon le tour dans lequel on est, la constante (**0x693FA863 ou 0x7F434625**) et la clé (**C_1** ou **C_2**) utilisées sont différentes.
En repassant dans chaque tour de l'algorithe, on trouve facilement la constante et la clé utilisée pour chaque (voir **solution-tours.xls**).

A partir de là, il suffit de faire des permutations à chaque tour de l'algorithme (pour utiliser le bon **C_X** et la bonne constante) pour retrouver les valeurs initiales de **M_1** à **M_4**, c'est à dire la passphrase (voir **main.c**).

On trouve donc la passphrase : **Z7aw?gW\=HL|Np_9**