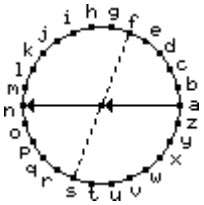
# Cryptographie sur 48

## *Espion dans l'âme, vos messages sont classés secret-défense. Un seul remède : le codage des données...*

### • Sacré Jules ! •

Tout a commencé à l'époque du bon vieux Jules César avec l'invention du code dit "de César". Le but étant de coder un message pour qu'il ne soit pas compris par l'ennemi. Classique. Ce code très simple est basé sur l'alphabet schématisé sous forme d'une roue de 26 caractères. Comme on peut le voir sur l'illustration ci-contre, la lettre **a** est convertie en lettre **n**, la lettre **f** est convertie en lettre **s**, etc. On remarque que si l'on fait le tour cette roue, la lettre **n** sera convertie en lettre **a** et la lettre **s** en lettre **f**. On en conclut aisément que ce code est facilement réversible, c'est à dire que le codeur et le décodeur seront en définitif le même programme. Comme exemple de message, prenons **LONGUE VIE A TEAM PALMTOPS** qui sera quasi immédiatement transformé en cet amalgame de caractères : **YBATHR IVR N GRNZ CNYZGBCF**. On repasse ce même message dans notre codeur et on retrouve le message d'origine. Le programme présenté (et non optimisé) se limite donc au code de César, soient aux 26 lettres de l'alphabet.



Le code de César

**'CESAR'** *(207 octets - CRC # BD7Dh)*

```
« DUP SIZE ¨ MESSAGE TAILLE
  « "" 1 TAILLE FOR BOUCLE
       MESSAGE BOUCLE DUP SUB NUM
       IF DUP DUP 65 Š SWAP 90 ‰ AND
       THEN 65 - 13 + 26 MOD 65 +
       END CHR +
    NEXT »
»
```

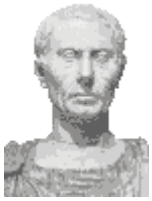
Dans le code ASCII, les lettres de l'alphabet sont représentées par les caractères n°65 à 90, c'est pourquoi on retranche 65 à la valeur de chaque lettre avant de lui appliquer cette petite formule : MOD(NUM(lettre)+13,26), c'est à dire qu'on garde juste le reste de la division avec 26 afin d'éviter les dépassements.

Si l'on généralise le code de César à l'ensemble de la table ASCII, qui comprend 256 caractères, alors le caractère n°0 sera transformé en caractère n°128, le n°1 en n°129, etc. Le programme en devient alors très allégé pour donner ceci :

**'ASCII'** *(132,5 octets - CRC # 9D4Ah)*

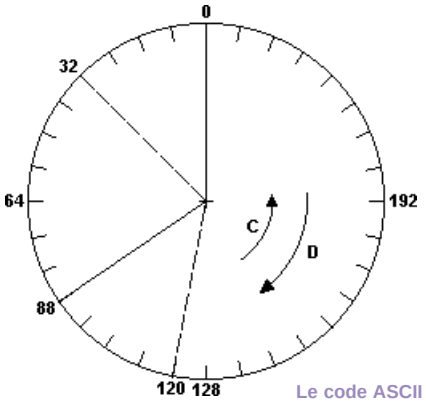
```
« DUP SIZE ¨ MESSAGE TAILLE
  « "" 1 TAILLE FOR BOUCLE
       MESSAGE BOUCLE DUP SUB NUM
       128 + 256 MOD CHR +
    NEXT »
»
```

Ainsi, notre message de départ sera transformé en quelque chose d'encore plus indigeste qu'il est absolument inutile de montrer. Et comme pour le programme 'CESAR', celui-ci est aussi réversible, c'est à dire que le codeur et le décodeur ne font qu'un seul et même programme.


Jules, en 8 niveaux de gris

### • Jules piraté... •

Certains malins ont jugé le précédent code trop simple à décrypter et ont choisi de réaliser un décalage de **n** caractères, ce qui implique deux programmes : un codeur et un décodeur. Le principe se complique un peu sans toutefois être difficile. Si l'on se base sur le code ASCII, le codage se fera selon la formule MOD(NUM(lettre)+n,256) et le décodage selon la formule inverse MOD(NUM(lettre)+256-n,256) et le tour est joué. Si on prend la seconde roue illustrée ci-dessus, on s'aperçoit que le caractère n°0 devient le caractère n°88 et que le caractère n°32 devient le caractère n°120. On comprend facilement que le décalage est de 88 caractères. Pour le codage du 1er exemple MOD(0+88,256)=88, et pour le décodage MOD(88+256-88,256)=0. Pour le codage du 2nd exemple, MOD(32+88,256)=120 pour le codage, et MOD(120+256-88,256)=32 pour le décodage. Le programme qui réalisera le codage prend donc en entrée le message et la valeur du décalage sur respectivement les niveaux 2 et 1 de la pile. Le programme de décodage demande la même chose, soient le message codé au niveau 2 de la pile et la valeur du décalage sur le niveau 1.



Le code ASCII

**'CODEUR'** *(101,5 octets - CRC # 4B10h)*

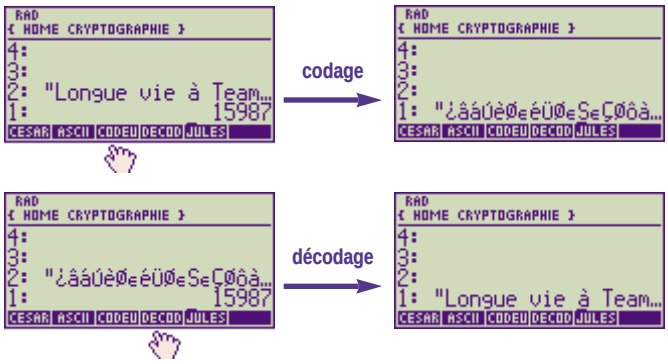
```
« SWAP DUP SIZE ¨ D M T
  « "" 1 T FOR B
       M B DUP SUB NUM
       D + 256 MOD CHR +
    NEXT »
»
```

**'DECODEUR'** *(114,5 octets - CRC # 3E3Eh)*

```
« SWAP DUP SIZE ¨ D M T
  « "" 1 T FOR B
       M B DUP SUB NUM
       256 + D - 256 MOD CHR +
    NEXT »
»
```

Prenons le même exemple de message que dans le 1er paragraphe et appliquons les deux programmes précédemment créés. Dans un premier temps, on pose les

objets demandés sur la pile et ensuite on lance le codage. L'opération inverse (le décodage) consiste à donner le même décalage sous peine d'obtenir un message toujours cryptés.




Il est bien évident que dans cet exemple il est plus difficile de trouver le bon décalage pour quiconque veut pirater votre message : il aurait juste 256 combinaisons possibles maximum pour le découvrir, ce qui rend cette codification facilement piratable.

### • Un peu de bidouille •

Le poids de deux programmes pour coder et décoder est assez lourd, alors passons à de nouvelles notions qui travaillent directement sur les octets pour n'avoir qu'un seul programme qui assume les deux opérations. Le plus simple est d'utiliser l'instruction NOT qui réalise de réelles prouesses. Le mieux est d'observer cet exemple de transformation : prenons la lettre **A** (codée 65 en ASCII) et infligeons lui un décalage de 76 modulo 256 avant d'utiliser la fonction NOT. Résultat : MOD(65+76,256)=141 (qui correspond au caractère ¨), et suite à un **NOT** notre lettre se retrouve tranformée en **r**. Voyons graphiquement comment tout ça réagit et profitons-en pour voir le décodage :

"**A**" → 65
+76
=141 → "¨" → **NOT** → "**r**"

"**r**" → 114
+ 76
=190 → "¾" → **NOT** → "**A**"

Il faudra donc réaliser cette opération sur chaque caractère. Dans notre programme, il est possible de réaliser d'abord l'ensemble des décalages et d'appliquer ensuite l'opérateur booléen NOT. Vous vous apercevrez que par rapport au programme 'CODEUR', seule la fonction NOT à été ajoutée à la fin du programme. Par contre le résultat d'un tel codage est bien plus compliqué à analyser pour le pirateur moyen surtout s'il ne connait pas son mode de codage.

**'CDC'** *(104 octets - CRC # 8D0Bh)*

```
« SWAP DUP SIZE ¨ D M T
  « "" 1 T FOR B
       M B DUP SUB NUM
       D + 256 MOD CHR +
    NEXT NOT »
»
```

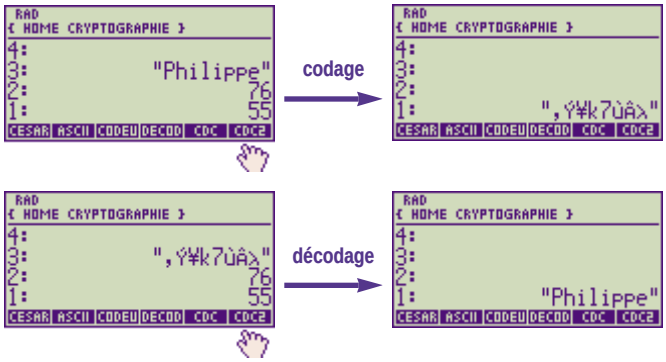
### • Plus vicieux •

Et si on réalisait un décalage dans le décalage ! Explications : si à chaque décalage on incrémente sa valeur d'un autre décalage, alors on aurait cette fois 256^2 possibilités maximum pour trouver la combinaison du message. Ce qui aurait pour but de voir une lettre codée de plusieurs façons dans un même message. Le tout est bouclé par un NOT, toujours pour n'avoir qu'un seul programme.

**'CDC2'** *(135 octets - CRC # 8981h)*

```
« ROT DUP SIZE ¨ D1 D2 M T
  « "" 1 T FOR B
       M B DUP SUB NUM
       D1 D2 + DUP 'D1' STO +
       256 MOD CHR +
    NEXT NOT »
»
```

Comme exemple, prenons un message court avec plusieurs fois la même lettre, avec un premier décalage de 76 et un second de 55 unités. On observe que les **i** et les **p** sont codés différemment, ceci grâce au second décalage.




L'inconvénient de ce genre de codage est qu'on ne peut pas concaténer deux (ou plusieurs) messages codés, car mêmes si les valeurs de décalages sont bonnes, seul la première partie (qui correspond au premier message) sera décodé.

Pour les passionnés d'assembleur, voici une source qui réalise la même opération que le programme CDC2, mais plus court sous tous rapports (vitesse et taille du programme). Voici tout d'abord le morceau de préparation réalisé en externals :

```
{ # 18A68h   Regarde la présence de 3 objets sur le niveau 1 de la pile,
  # 18FB2h   Regarde le type des objets :  si ce sont une chaîne de
  785        caractères et deux nombres réels, on passe à la suite,
  { # 18CEAh      Convertit un réel en entier système,
    # 3223h       Instruction SWAP,
    # 18CEAh # 3223h
    # 3295h       Instruction ROT,
    # 6657h       Instruction NEWOB,
    # 60FACh      Instruction 3 ROLLD,
    Code          Le programme principal,
    # 3258h } }   Efface les décalages par un DROP2.
```

Pour couronner le tout, la source assembleur est d'une extrême simplicité, même pour un novice en la matière. Dans un premier temps, on s'occupe d'initialiser le programme en sauvant les registres et pointeurs, puis de calculer les boucles et décalages nécessaires au bon fonctionnement du programme final. Mais une bonne source vaut mieux qu'un long discours :

```
"               A=A-5 A         *BOUCLE
GOSBVL 0679B    ASRB A          C=C+D B
A=DAT1 A        ?A=0 A          A=DAT0 B
D0=A            GOYES FIN       A=A+C B
D0=D0+ 5        A=A-1 A         A=-A-1 B
C=DAT0 B        B=A A           DAT0=A B
D=C B           D0=D0+ 5        D0=D0+ 2
D1=D1+ 10       D1=D1- 5        B=B-1 A
A=DAT1 A        C=DAT1 A        GONC BOUCLE
D0=A            D1=C            *FIN
D0=D0+ 5        D1=D1+ 5        GOVLNG 05143
A=DAT0 A        C=DAT1 B        @"
```

Pour ceux qui n'ont pas de PC pour charger les programmes, voici la source hexadécimale qu'il faut saisir dans une chaîne de caractères sans espace ni saut de ligne, et l'assembler avec GASS (proposé juste après) :

```
D9D20 86A81 2BF81 11920 11300 D9D20
AEC81 32230 AEC81 32230 59230 75660
CAF06 CCD20 E6000 8FB97 60143 13016
414EA E7179 14313 01641 42818 F8481
9F08A 8F2CC D8164 1C414 71351 7414F
A6B14 AA6AB EC148 161CD 5BE8D 34150
85230 B2130 B2130
```

**'GASS'** *(86,5 octets - CRC # 1DB3h)*

```
« "GROB 8 " OVER SIZE 2 / + " " + SWAP
+ OBJ¨ # 4017h SYSEVAL # 56B6h SYSEVAL
DROP NEWOB »
```

Le programme final assemblé sera nommé **'CDC2LM'** et doit avoir une taille de **97,5** octets et un CRC égal à **# 5123h**. L'avantage de la version assembleur est qu'elle est environ 3500 fois plus rapide que la version RPL : un message de 10000 caractères est codé en 15 minutes 34 secondes 39 centièmes en RPL, alors que la version assembleur met 43 centièmes de seconde ! Fou, non ?

### • Le top ! •

Depuis le début de cet article, nous avons toujours effectué des décalages à partir d'entiers préalablement posés sur la pile. L'idéal serait de coder un message à partir d'un mot de passe et de décoder le message codé avec le même mot de passe qu'on appellera ici **clef**. Imaginons simplement que la machine demande le message à coder au niveau 2 de la pile et la clef (de taille quelconque) au niveau 1 sous forme de chaîne de caractères. Le principe ressemble d'assez près au précédent programme au niveau algorithme. La seule différence réside dans le fait qu'on prend le code ASCII du premier caractère de la clef et qu'on l'ajoute au premier caractère du message, idem pour le second caractère, et ainsi de suite... Cette fois, nous allons passer directement à la version assembleur en commençant par les quelques externals qui s'imposent (sans commentaire), suivis par la source et le code hexadécimal :

```
{ # 18A8Dh # 18FB2h 51 { # 3223h # 6657h
# 3223h Code # 3244h } }
```

```
"               D1=D1+ 5        B=B-1 A
GOSBVL 0679B    C=DAT1 A        GONC SUITE
A=DAT1 A        C=C-5 A         C=R4 A
D0=A            CSRB A          B=C A
D0=D0+ 5        ?C=0 A          CD0EX
A=DAT0 A        GOYES FIN       C=C-B A
A=A-5 A         C=C-1 A         C=C-B A
ASRB A          D=C A           C=C-2 A
?A=0 A          D1=D1+ 5        D0=C
GOYES FIN       *BOUCLE         *SUITE
A=A-1 A         A=DAT0 B        D=D-1 A
R4=A A          C=DAT1 B        GONC BOUCLE
D0=D0+ 5        A=A+C B         *FIN
B=A A           A=-A-1 B        GOVLNG 05143
D1=D1+ 5        DAT1=A B        @"
C=DAT1 A        D0=D0+ 2
D1=C            D1=D1+ 2
```

```
D9D20 D8A81 2BF81 11920 33000 D9D20
32230 75660 32230 CCD20 99000 8FB97
60143 13016 41428 18F84 819F0 8A8C6
CC81A F0416 4D817 41471 35174 14781
8FA48 19F28 AA04C ED717 414A1 4FA6A
BEC14 91611 71CD5 A181A F1CD5 136E9
E9818 FA113 4CF5B C8D34 15044 230B2
130B2 130
```

Le programme final assemblé sera nommé **'CDCK'** et doit avoir une taille de **109** octets et un CRC égal à **# 21FDh**.

### • Les programmes •

Retrouvez l'ensemble des programmes de cet article sur le CD-Rom avec en bonus le portrait de Jules César en 8 niveaux de gris.

### • Le mot de la fin •

Dans ces trois pages nous avons aperçu plusieurs méthodes pour ne coder que des messages, alors qu'il serait aussi intéressant de coder des fichiers ou des objets de n'importe quelle nature. Alors comme vous avez désormais toutes les bases sur le sujet, je vous laisse réfléchir un peu...

Philippe Pamart