Auteur : Anarchriz/DREAD Date de Sortie : le 29 avril 1999 (dernière modification le 30 avril 1999)

Difficulté : débutant à avancé

Cible : CRC algorithme

Outils : QEdit 2.1 (le mieux!, Wordpad et quelque prog CRC)

CRCet comment Les modifier

Introduction

Cet essai va essayer de vous montrer comment se débarrasser des CRC.
Beaucoup de Coders/Reversers ne savent pas exactement comment travaillent les CRC et presque personne ne sait comment les modifier...
Tout d'abord, ce tutorial va vous apprendre comment calculer un CRC de manière générale.
Deuxièmement, la partie Reverse vous apprendra (principalement) comment modifier complètement les CRC-32.
Vous pourrez employer cela pour casser des certaines protections (Comme les anti-virus). Il semble y avoir des utilitaires qui puissent 'corriger' un CRCS pour vous, mais je doute qu'ils expliquent aussi comment ils s'y prennent.
J'aimerais vous avertir qu'il y a pas mal de maths employés dans cet essai. Cela ne fera de mal à personne, et permettra de mieux comprendre le Mécanisme de Reverse Engineering.

Partie 1 : comment calculer un CRC

Le Code de Redondance Cyclique ou CRC est fréquemment utilisé, et vous l'avez déjà rencontré, peut être même sans le savoir. Vous vous rappelez tous être tombé sur un message ennuyant " File corrupted " avec des fichiers RAR, ou d'autres compresseurs, suite à un mauvais téléchargement, ou à cause d'une de ces @#$% de disquettes...
Le CRC est une valeur calculée sur un morceau de données avant compression. Quand l'archiver déballe ce fichier, il lit la valeur du CRC précédemment écrit et contrôle la validité du fichier non décompressé en refaisant le même calcul. Quand les deux résultats correspondent, il y a de bonne chance que les fichiers soient identiques. Avec les CRC-32,Il y a un risque d'erreur de l'ordre de 1/2^32 que la vérification ne détecte pas un changement de données.
comment le calcul est-il fait ? et bien l'idée principale est de voir le fichier comme un grande série d'octets divisées par un nombre, et qui vous laissera un Reste, le CRC!
Vous avez toujours un reste (qui peut aussi être zéro), et toujours un diviseur(9/3=3 reste=0; (9+2)/3=3 reste=2) qui peut être sous la forme de X soustractions. Si vous voulez trouver la valeur d'origine il suffit de multiplier le diviseur (ou additionner X fois) et ajouter le reste.

Voici 2 exemples, le numéro 1 est une soustraction normale, 2&3 Sont particuliers :
-+

```
(1) 1101  (2) 1010   1010  (3) 0+0=0  0-0=0
    1010-     1111 + 1111-     0+1=1 *0-1=1
    -----     ----   ----      1+0=1  1-0=1
    0011      0101   0101     *1+1=0  1-1=0
```

Dans l'exemple (1), la deuxième colonne de droite équivaut à 0-1 =-1, donc un bit a été emprunté au bit d'a coté, et vous donnera cette soustraction (10+0)-1=1. (C'est comme une soustraction décimale classique sur le papier), Pour les cas particuliers (2&3), 1+1 aurait du normalement donner comme réponse 10, où '1' est la retenue qui transmet une valeur au bit

suivant. Cette valeur est oubliée. Le cas spécial 0-1 aurait normalement dû donner comme réponse '-1 '. Si vous avez des notions de programmation, cela ressemble, ou mieux, C'EST l'opération XOR. Regardez maintenant l'exemple d'une division:

Dans une arithmétique normale :

```
1001/1111000\1101 13                    9/120\13
        1001     -                        09 - |
        ----                              --   |
        1100                              30   |
        1001 -                            27 -
        ----                              --
        0110                               3 - > le reste
        0000 -
        ----
        1100
        1001 -
        ----
         011 - > 3, le reste
```

Dans l'arithmétique CRC :

```
1001/1111000\1110                       9/120\14 reste 6
       1001 -
       ----
       1100
       1001 -
       ----
       1010
       1001 -
       ----
       0110
       0000 -
       ----
        110 - > le reste( Exemple 3)
```

Le quotient d'une division n'est pas important et il est inutile de s'en souvenir, parce que ce n'est qu'un couple de octets inférieurs du bitstring que vous avez voulu calculer pour le CRC. Ce qui EST important, c'est le reste!

C'est la seule chose qui représente quelque chose d'important par rapport au fichier original. C'est le fondement du CRC!

Passant au calcul réel du CRC :

Pour exécuter un calcul CRC nous avons besoin de choisir un diviseur, nous l'appellerons le'poly' dorénavant. La largeur W d'un poly est la position du bit le plus élevé, donc la taille du poly 1001 est 3 et non pas 4.
Notez que le bit le plus élevé est toujours Un, quand vous devez choisir la taille d'un poly, vous avez seulement à choisir une valeur pour la partie basse des octets W. Si nous voulons calculer le CRC sur une bitstring, nous devons nous assurer que tous les octets sont traitées.
Il faut donc ajouter W zéro octets à la fin de la Bitstring.
Dans l'exemple 3, nous allons supposer que la bitstring était 1111.
Regardez un exemple un peu plus grand :

```
Le poly              = 10011, taille de W =4
Bitstring + W zéros = 110101101 + 0000
```

```
10011/1101011010000\110000101 (nous ne nous soucions pas du quotient)

              10011 |||||||| -
               -----||||||||
                10011|||||||
                10011||||||| -
                -----|||||||
                 00001||||||
                 00000|||||| -
                 -----||||||
                  00010|||||
                  00000||||| -
                  -----|||||
                   00101||||
                   00000|||| -
                   -----||||
                    01010|||
                    00000||| -
                    -----|||
                     10100||
                     10011|| -
                     -----||
                     01110 |
                     00000 | -
                      -----|
                       11100
                       10011-
                       -----
                        1111 - > le reste - > le CRC!( Exemple 4)
```

Il y a 2 choses importantes d'exposées ici :

1. Uniquement quand l'octet de poids fort est l'un des octets de la bitstring, nous le XORons avec le poly, autrement dit nous 'changeons' seulement la bitstring d'un bit sur la gauche.
2. L'effet du XORRING, quand il XORé avec les octets inférieurs W, est que le bit de poids fort donne toujours zéro.

Table-Driven Algorithm

Vous devez comprendre qu'un algorithme basé sur le calcul bitwise sera très lent et inefficace.
Il serait beaucoup plus efficace de le calculer sur une base "par octet".
Mais alors nous pouvons seulement accepter les poly's avec une taille d'un multiple de 8 (c'est un octet).
Voyons ca dans un exemple avec un poly d'une taille de 32 octets (W =32) :

```
        3   2   1   0    octet
        +---+---+---+---+
Pop < - |   |   |   |   | < - bitstring avec des W bits zéro ajoutés, cas 32
        +---+---+---+---+
        1 <---32 octets---> c'est le poly, 4*8 octets
```

(Figure 1)

C'est un registre que vous utilisez pour stocker le résultat provisoire du CRC, je l'appelle le registre CRC ou uniquement Registre dorénavant. Vous changez des octets de la bitstring du coté

droit, et d'autres du côté gauche. Quand un bit change du côté gauche, le registre entier est XORRED par les W octets bas du poly (dans le cas 32).
En fait, nous faisons exactement la même chose que dans la division ci-dessus.

Regardez l'exemple d'un CRC 8 octets avec 4 octets de changés immédiatement :

Le registre juste avant le changement : 10110100

4 octets sont (en partie supérieure) changés du côté gauche en modifiant 4 nouveaux Octets du côté droit.
Dans cet exemple 1011 est changé en 1101.
La situation est celle ci :

```
8 octets du registre CRC          : 01001101
4 octets supérieurs sont modifiés : 1011
Nous utilisons le poly            : 101011100, taille W=8
```

Nous calculons la nouvelle valeur du registre.

```
Top     Registre
----    --------
1011    01001101   le top-bit  et le registre
1010       11100 + (*1) du poly est XORRED sur le 3ème bit
----    --------
0001    10101101   résultat du XORRING
```

Nous avons toujours un 1 sur le bit 0 du top-bits :

```
0001    10101101   résultat précédent
   1    01011100 + (*2) le poly est XORRED sur le bit 0 du top octets
----    ---------
0000    11110001   résultat de seconde XORring
^^^^
```

Maintenant tout le top-bits est à 0, et nous n'avons plus besoin de XORer avec le poly pour cette séquence.
Vous obtenez la même valeur dans le registre si vous XORez (*1) avec (*2) et le résultat avec le registre. C'est à cause des propriétés du XOR:

(a XOR b) XOR c = a XOR (b XOR c)

```
1010      11100    poly sur le bit 3 du top-bits
   1   01011100 +  poly XORred sur le bit 0 du top-bits
----   --------
1011   10111100    (*3) résultat du XORRINGLe
```

Le résultat (*3)   est XORRED avec le registre

```
1011   10111100
1011   01001101 +  les octets supérieurs  et le registre
----   --------
0000   11110001
```

Vous voyez ? Le même résultat!

Maintenant (*3) est important, parce que le bit supérieur 1010 est toujours la valeur (*3)=10111100 (pour les octets inférieurs W=8).

Cela signifie que vous pouvez pré-calculer les valeurs Xorées pour chaque combinaison d'octets supérieurs. Notez que des octets supérieurs arrivent toujours à zéro après une itération, cela doit être parce que la combinaison du XORRING y mène.

Revenons à la figure 1.

Pour chaques valeurs du byte supérieur (8 octets) changés, nous pouvons pré calculer une valeur. Dans ce cas se serait une table de 256 (2^8) entrées de DWORD (32bit). (La table CRC-32 est dans l'annexe)
En pseudo-language notre algorithme est maintenant celui ci :

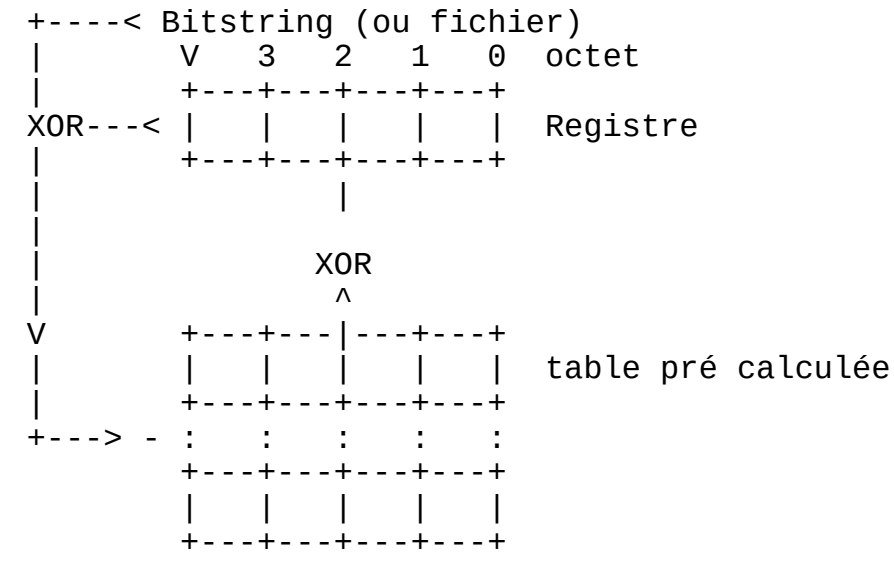
```
While (la bitsring n'est pas exhaustive)

Begin
   Top      = top_byte of register
   Register = Register shifted 8 octets left ORred with a new byte from string
   Register = Register XORred by value from precomputedTable at position Top
 End
```

L'Algorithme de la Table

L'algorithme proposé ci-dessus peut être optimisé. Les octets de la bitString n'ont pas besoin d'aller dans tout le registre avant qu'ils ne soient employés.
Avec ce nouvel algorithme nous pouvons directement XORé un octet d'une Bitstring avec l'octet changé du registre. Le résultat pointe vers une valeur dans la table pré-calculée qui sera XORRée avec le registre. Je ne sais pas exactement pourquoi cela donne le même résultat (ca doit avoir un rapport avec les propriétés de la fonction XOR), mais il a le Grand avantage de ne pas vous obliger à ajouter des octets nuls / octets de votre Bitstring.

Regardons cet algorithme :

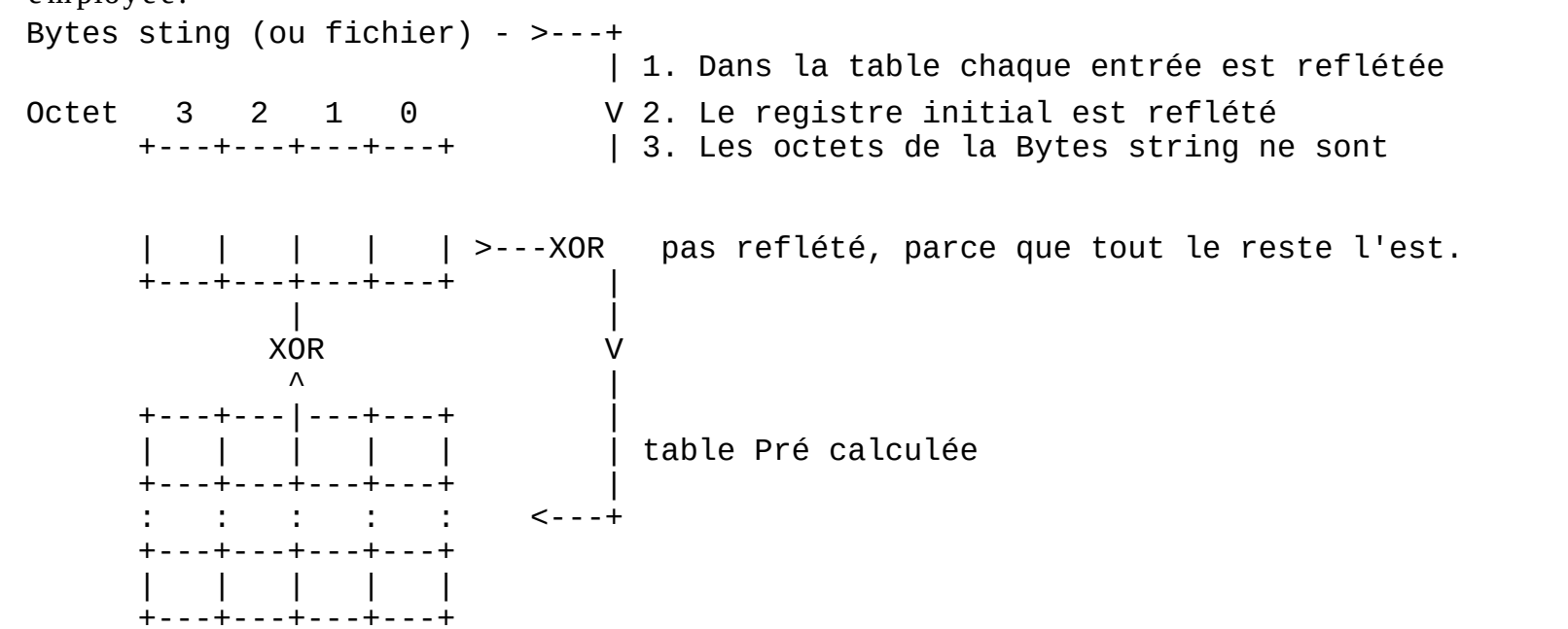


(Figurez 2)

The 'reflected' direct Table Algorithm

Pour compliqué un peu les choses, il y a une version 'reflet' de cet Algorithme. Une valeur / registre Reflected est un bit qui a pivoté autour de son centre. Par exemple 0111011001 est le reflet de 1001101110.
Ils ont inventé cela à cause de l'UART (le chip qui utilise le serial IO),qui envoie chaque octet avec le bit le moins significatif (le bit 0) d'abord et en dernier le bit le plus significatif (le bit 7),

c'est le contraire de la situation normale. Au lieu de refléter chaque octet avant de le traiter, chaque contraire est reflété. Un des avantages est d'obtenir un code plus compact lors de la mise en Suvre. Ainsi, dans le calcul de la table, les octets sont changés par la droite et le poly est reflected. Dans le calcul du CRC le registre est changé à droite et (bien sûr) la table reflétée est employée.



(Figurez 3)

Notre algorithme est maintenant :

1. Changer le registre directement par un octet
2. XORer l'octet supérieur par un nouvel octet de la Bytes string
   Pour pointer vers un index dans la table ([0,255])
3. XORer la valeur de la table dans le registre
4. Goto 1 s'il y a plus d'octets à traiter

Quelques exemples en Assembleur
voici la norme complète CRC-32 :

```
Nom             : "CRC-32"
Taille          : 32
Poly            : 04C11DB7
Valeur initiale : FFFFFFFF
Reflet          : Vrai
XORé avec       : FFFFFFFF
```

En prime, pour les gens curieux, voici la norme CRC-16 :

```
Nom             : "CRC-16"
Taille          : 16
Poly            : 8005
Valeur initiale : 0000
Reflet          : Vrai
XORé avec       : 0000
```

'XORé avec' est la valeur qui est XORRée avec la valeur finale du registre avant obtention (comme réponse) du CRC final.
Il y a aussi des 'reversed CRC Poly's' mais ils ne sont pas appropriés pour ce tutorial.
Regardez mes références si vous voulez en savoir plus.

Pour la mise en Suvre en assembleur, j'emploie du code en 32 bits et du codes 16 bits...
Donc vous verrez quelques mélanges entre les deux codes...

Il est facile de convertir cela pour transformer le code en 32 bits.

Notez que la partie assembleur est entièrement écrite pour fonctionner correctement en Java ou en C dont il est tiré.
Ok. Voici la routine assembleur pour le calcul de la table CRC-32 :

```
      Xor ebx, ebx     ; ebx=0, parce qu'il sera employé comme pointeur
InitTableLoop :
      Xor eax, eax     ; eax=0 pour nouvelle entrée
      Mov al, bl       ; partie basses 8 octets d'ebx est copiée dans partie
                       ; basses 8 octets d'eax
; Generate Entry
      Xor cx, cx
EntryLoop :
      test eax, 1
      Jz no_topbit
      Shr eax, 1
      Xor eax, poly
      Jmp entrygoon
No_topbit :
      Shr eax, 1
Entrygoon :
      Inc cx
      test cx, 8
      Jz entryLoop
      Mov dword ptr [ebx*4 + crctable], eax
      Inc bx
      test bx, 256
      Jz InitTableLoop
```

Notes :
- crctable est un tableau de 256 DWORD
- Eax est changé à droite parce que le CRC-32 emploie l'Algorithme reflété

- donc la partie basse 8 octets est traitée...

Dans du Java ou du C (int est en 32 bits) :

```
for (int cx=0; cx<8; cx++)
{ if (eax&&0x1)
{ eax>>=1; eax^=poly;
 }
else eax>>=1;
}
crctable[bx]=eax;
}
```

La mise en Suvre pour le calcul du CRC-32 utilise la table :

```
computeLoop:
   xor ebx, ebx
   xor al, [si]
   mov bl, al
   shr eax, 8
```

```
  xor eax, dword ptr[4*ebx+crctable]
  inc si
  loop computeLoop
  xor eax, 0FFFFFFFFh
```

Notes :
- ds:si pointe vers le buffer où les octets sont traités
- Cx contient le nombre d'octets à traiter
- Eax contient le CRC en cours
- Crctable est la table calculée avec le code ci-dessus
- La valeur initiale du CRC est dans le cas d'un CRC-32 : FFFFFFFF
- Après le calcul complet, le CRC est XORRED avec : FFFFFFFF

En Java ou en C on obtient ceci :

```
for (int cx=0 ; cx >=8 ;
eax^=crcTable[ebx];
}
eax^=0xFFFFFFFF;
```

Nous voici arrivé à la fin de la première partie : si vous voulez en savoir encore plus sur les CRC, je vous suggère de lire la documentation que vous trouverez dans les liens à la fin de ce texte.

Ok.
Passons à la partie intéressante de ce document : Reverser un CRC-32 :

Partie II: Reversing CRC

Alors que je cherchais une façon de modifier un CRC...
J'ai été coincé plusieurs fois !
J'ai essayé de 'désactiver' le CRC en pensant que l'ordre des octets n'avait pas d'importance quel que soit l'octet qui arrivait derrière...
et je me suis rendu compte que je ne pourrait jamais travailler ainsi, parce que l'algorithme CRC est construit de telle façon qu'il se fiche de l'octet que vous changez
Je me suis rendu compte que je pourrais seulement 'corriger' le CRC après avoir modifié les octets qui m'intéressaient. Donc il fallait que je transforme le CRC en l'a valeur qui m'intéressait.

Voici mon idée :

```
Série d'octets : 01234567890123456789012345678901234567890123456789012

Vous voulez intervertir   ^        et       ^
```

De la position 9 à 26.

Nous avons aussi besoin de 4 octets supplémentaires (avant la position 30) pour la séquence d'octets qui changeront la valeur d'origine du CRC après que les octets auront été modifiés.
Quand vous calculez le CRC-32 il n'y a pas de problème jusqu'à l'octet en 9ème position, ensuite la série d'octets patchés va faire changer radicalement le CRC. Même au-delà de la position 26, où les octets ne sont pas changés, vous n'arrivez jamais à retrouver les valeurs du CRC original.

En lisant le reste de cet essai, vous comprendrez pourquoi.

En fait il va falloir que vous préserviez le CRC quand vous patcherez une partie du code:

1. Calcul du CRC jusqu'à la position 9, et sauvegarde de cette valeur.
2. Continuer le calcul jusqu'à la position 27 et les 4 extra bytes, sauver le résultat.
3. Utiliser la valeur de 1 pour calculer le CRC des " nouveaux " octets et des 4 extra bytes (ca

devrait donner 27-9+4=22 bytes) et sauver la valeur obtenue.
4. maintenant, nous avons la "nouvelle" valeur du CRC, mais nous voulons que le CRC devienne "l'ancienne" valeur du CRC. Nous allons utiliser "the reverse algorithm" pour calculer les 4 extra bytes.

Reversing CRC-16

Je pense que, pour rendre les choses plus faciles, vous devez calculer en premier un CRC-16. Nous nous situons, d'une certaine manière, après le code patché, là ou nous voulons changer le CRC. Nous connaissons le CRC original (calculé avant le patchage des data) et l'état actuel du registre CRC. Nous voulons calculer les deux bytestring qui modifie le CRC.

En premier, nous allons calculer 'normalement' le CRC avec les 2 octets inconnus appelé X et Y, pour le registre je vais prendre a1 a0, la seul non-variable etant zéro (00).

Regardez à nouveau notre dernier algorithme CRC, figure 3, pour mieux comprendre ce que je vais faire.
Ok, nous y sommes :

Prenez les 2-bytestring 'X Y'. Les Bytes sont traités à partir du coté gauche.
Prenez comme registre a1 a0.
Pour une opération XOR, j'écrirai '+' (comme dans le tutorial sur le CRC)

Traitement du premier byte, X:

```
a0+X            c'est le calcul du bit supérieur (topbyte) (1)
b1 b0           séquence dans la table où pointe le topbyte points à
00 a1           la droite du registre
00+b1 a1+b0     2 précédentes lignes XORred avec les autres
```

Le nouveau registre devient: (b1) (a1+b0)

Traitement du second byte, Y:

```
(a1+b0)+Y       c'est le calcul du bit supérieur topbyte (2)
c1 c0           séquence dans la table où pointe le topbyte points à
00 b1           la droite du registre
00+c1 b1+c0     2 précédentes lignes XORred avec les autres
```

Le registre final est: (c1) (b1+c0)

Je vais vous montrer un petite variante:

```
a0 + X      =(1)  points to b1 b0 in table
a1 + b0 + Y =(2)  points to c1 c0 in table
     b1 + c0=d0   new low byte of register
          c1=d1   new high byte of register
   (1)      (2)
```

Ne soyez pas inquiet, un exemple plus concert arrive ensuite.

Vous voulez que le registre soit égal à d1 d0 (le CRC original) et vous connaissez la valeur du registre avant la transformation (a1 a0)...
Quels sont les 2 octets, ou quels X et Y, devez vous prendre pour le calcul du CRC ?
Ok. Nous allons travailler de l'arrière vers l'avant. d0 doit devenir b1+c0 et d1 doit devenir c1.
Mais comment diable !, je vous entends dire, pouvez vous savoir la valeur de l'octet b1 et c0 ? ? ?

Dois je vous rappelez la Table?
Vous pouvez consulter la valeur du WORD C0 C1 dans la Table parce que vous connaissez C1. Donc vous besoin de faire une routine 'de consultation'. Si vous avez trouvé la valeur, soyez sur de vous rappelez l'index de cette valeur parce que c'est le moyen de trouver les deux topbytes inconnus, par exemple.(1)&(2)!
Si maintenant vous avez trouvé c1 c0, comment obtenir b1 b0 ?
Si b1+c0=d0 alors b1=d0+c0!

Utilisez votre routine de consultation pour trouver la valeur de b1 b0. Maintenant nous connaissons tout ce qui nous est nécessaire pour calculer X et Y!

```
a1+b0+Y=(2) donc Y=a1+b0+(2)

a0+X=(1)    donc X=a0+(1)
```

Non-variable example for CRC-16

Regardons un exemple avec de vraies valeurs:

- registre avant: (a1=)DE (a0=)AD

- registre désiré: (d1=)12 (d0=)34

Regardez l'entrée commençant par 12 dans la table CRC-16 table de l'annexe.

- C'est l'entrée 38h dont la valeur est 12C0. Essayons de trouver une autre entrée commençant par 12.
Vous ne pouvez pas en trouver parce que nous avons calculé chaque entrée pour chaque valeur possible du topbyte et qu'il n'y en a que 256.

Maintenant nous connaissons (2)= 38, c1= 12 et c0= C0, donc b1= C0+34=F4, regardons l'entrée de B1 commençant avec F4.

- c'est l'entrée 4Fh avec la valeur F441.

(1)= 4F, b1= F4 et b0= 41. Toute ces valeurs nécessaires vont permettre de calculer X and Y en faisant:

```
Y=a1+b0+(2)= DE+41+38 = A7
X=a0+(1)   = AD+4F    = E2
```

Conclusion: pour changer le registre CRC-16 de DEAD vers 1234, nous avons besoin des bytes

E2 A7 (dans cet ordre).

Vous voyez, pour reverser un CRC, vous devez calculer à l'envers, et vous rappeler des valeurs trouvées. Si vous programmez la table de consultation en assembleur, rappelez vous qu'INTEL sauve les valeurs à l'envers en Little-Endian format.

Passons au CRC-32

Reversing CRC-32

Le CRC-32 est aussi facile (ou difficile) que le CRC-16. Nous allons travailler avec 4 octets au lieu de deux.

```
Prenez 4-bytestring X  Y  Z  W  , Les octets sont pris à partir de la gauche.
Prenez comme registre  a3 a2 a1 a0
```

Notez que a3 est l'octet le plus significatif, et a0 le moins.

Traitement du premier octet, X:

```
a0+X                    this is the calculated topbyte (1)
b3    b2    b1    b0    sequence in table where the topbyte points at
00    a3    a2    a1    to right shifted register
00+b3 a3+b2 a2+b1 a1+b0 previous 2 lines XORred with eachother
```

Le nouveau registre donne: (b3) (a3+b2) (a2+b1) (a1+b0)

Traitement du second octete, Y:

```
(a1+b0)+Y                       this is the calculated topbyte (2)
c3    c2    c1       c0         sequence in table where the topbyte points at
00    b3    a3+b2    a2+b1      to right shifted register
00+c3 b3+c2 a3+b2+c1 a2+b1+c0   previous 2 lines XORred with eachother
```

Le nouveau registre donne: (c3) (b3+c2) (a3+b2+c1) (a2+b1+c0)

Traitement du troisième octet, Z:

```
(a2+b1+c0)+Z                    this is the calculated topbyte (3)
d3    d2    d1       d0         sequence in table where the topbyte points at
00    c3    b3+c2    a3+b2+c1    to right shifted register
00+d3 c3+d2 b3+c2+d1 a3+b2+c1+d0 previous 2 lines XORred with eachother
```

Le nouveau registre donne: (d3) (c3+d2) (b3+c2+d1) (a3+b2+c1+d0)

Traitement du quatrième octet, W:

```
(a3+b2+c1+d0)+W                 this is the calculated topbyte (4)
e3    e2    e1       e0         sequence in table where the topbyte points at
00    d3    c3+d2    b3+c2+d1    to right shifted register
00+e3 d3+e2 c3+d2+e1 b3+c2+d1+e0 previous 2 lines XORred with eachother
```

Le registre final est égal à: (e3) (d3+e2) (c3+d2+e1) (b3+c2+d1+e0)

Je vais vous montrer une méthode différente:

```
a0 + X                  =(1)  pointe vers  b3 b2 b1 b0  in table
a1 + b0 + Y             =(2)  pointe vers  c3 c2 c1 c0  in table
a2 + b1 + c0 + Z        =(3)  pointe vers  d3 d2 d1 d0  in table
a3 + b2 + c1 + d0 + W   =(4)  pointe vers  e4 e3 e2 e1  in table
     b3 + c2 + d1 + e0  =f0
          c3 + d2 + e1  =f1
               d3 + e2  =f2
                    e3  =f3
    (1)  (2)  (3)  (4)
```

(figure 4)

Voici un exemple en valeurs réelles:
(utilisez la table CRC-32 de l'index).

Prenez comme valeur CRC d'origine a3 a2 a1 a0 -> AB CD EF 66
Prenez comme valeur CRC modifié, f3 f2 f1 f0 -> 56 33 14 78 (valeurs recherchées)

```
First byte of entries            entry   value

e3=f3                      =56 -> 35h=(4) 56B3C423 for e3 e2 e1 e0
d3=f2+e2      =33+B3       =E6 -> 4Fh=(3) E6635C01 for d3 d2 d1 d0
c3=f1+e1+d2   =14+C4+63    =B3 -> F8h=(2) B3667A2E for c3 c2 c1 c0
b3=f0+e0+d1+c2=78+23+5C+66=61 -> DEh=(1) 616BFFD3 for b3 b2 b1 b0
```

Nous avons désormais toutes les valeurs nécessaires :

```
X=(1)+          a0=          DE+66=B8
Y=(2)+       b0+a1=       F8+D3+EF=C4
Z=(3)+   c0+b1+a2=    4F+2E+FF+CD=53
W=(4)+d0+c1+b2+a3=35+01+7A+6B+AB=8E
(calcul final)
```

Conclusion: pour changer le CRC-32 de ABCDEF66 vers 56331478 nous avons besoin de cette séquence d'octets: B8 C4 53 8E

The reverse Algorithm for CRC-32

Si vous regardez le by-hand computation de la séquence d'octets nécessaires, pour changer le registre CRC de a3 a2 a1 a0 vers f3 f2 f1 f0, il est difficile de transformer celui ci en un bel algorithme compact.

Regardez cette version étendue du calcul final:

```
                              Position

X =(1) +                 a0     0
Y =(2) +            b0 + a1     1
Z =(3) +       c0 + b1 + a2     2
W =(4) + d0 + c1 + b2 + a3     3
f0= e0 + d1 + c2 + b3          4
f1= e1 + d2 + c3               5
f2= e2 + d3                    6
f3= e3                         7
```

(figure 5)

C'est la même chose que pour la figure 4, seul quelques valeurs/octets changent. Ce schéma va nous aider à obtenir un algorithme plus compact.
Si nous prenons un buffer de 8 octets, c'est pour réserver 1 octet, pour chaque ligne que vous voyez dans la figure 5.
Les octets 0 à 3 sont remplis avec a0 à a3, les octets 4 à 7 sont remplis avec f0 à f3. Comme précédemment, nous utilisons le dernier octet e3, qui est égal à f3, pour regarder la valeur correspondante dans la table CRC.
Puis nous XORons cette valeur (e3 e2 e1 e0) sur la position 4 (comme dans la figure 5). Alors nous obtenons automatiquement la valeur de d3, parce nous avons déjà XORré f3 f2 f1 f0 avec e3 e2 e1 e0, et que f2+e2 = d3.
Comme nous connaissons maintenant la valeur (4), nous pouvons directement XORer cette valeur avec la position 3.

Maintenant, nous savons que d3 utilise ce résultat pour trouver les valeurs de d3 d2 d1 d0 et XOR sur la position précédante, qui est la position3 (regardez la figure!).
XOR le nombre trouvé (3) pour la valeur en position 2. Nous connaissons désormais c3parce que

nous avons les valeurs de f1+e1+d2=c3 sur la position 5.

Nous continuons jusqu'à ce que nous XORons b3 b2 b1 b0 sur la position 1. et voila!

Les octets 0 a 3 du buffer contiennent les octets X et W!

Récapitulons l'algorithme :

1. Dans un buffer de 8 octets, remplir la position 0 à 3 avec a0 à a3 (la valeur de départ du registre CRC), et la position 4 à 7 avec f0 à f3 (valeur de finale recherchée du registre CRC).

2. Prendre l'octet en position 7 et l'utiliser pour regarder la valeur complète.
3. XOR cette valeur (dword) sur la position 4
4. XOR le nombre d'entré (byte) sur la position 3
5. Répéter l'étape 2 & 3 trois fois de plus en décrémentant la position à chaque fois par un.

Source de l'Algorithme

Place aux codes. Ci dessous vous trouverez le source de l'algorithme pour le CRC-en en assembleur (ca n'est pas bien compliqué de l'adapter à d'autres langage et/ou aux CRC standards). Notez qu'en assembleur (sur les PC) les DWORD sont écrits et lu dans l'ordre inverse de leur apparition en mémoire.

```
crcBefore       dd (?)
wantedCrc       dd (?)
buffer          db 8 dup (?)

        mov     eax, dword ptr[crcBefore] ;/*
        mov     dword ptr[buffer], eax
        mov     eax, dword ptr[wantedCrc] ; Step 1
        mov     dword ptr[buffer+4], eax  ;*/
        mov     di, 4

 computeReverseLoop:

        mov     al, byte ptr[buffer+di+3] ;/*
        call   GetTableEntry              ; Step 2 */
        xor     dword ptr[buffer+di], eax ; Step 3
        xor     byte ptr[buffer+di-1], bl ; Step 4
        dec     di                        ;/*
        jnz    computeReverseLoop         ; Step 5 */
```

Notes:

- les registres eax, di et bx sont utilisés

Implementation of GetTableEntry

```
crctable        dd 256 dup (?)       ;doit être définie quelque part &
                                     ; et initialisée

        mov     bx, offset crctable-1

 getTableEntryLoop:

        add     bx, 4                ;pointe vers (crctable-1)+k*4 (k:1..256)
        cmp     [bx], al             ;doit tjs trouver la valeur quelque part
```

```
        jne     getTableEntryLoop
        sub     bx, 3
        mov     eax, [bx]
        sub     bx, offset crctable
        shr     bx, 2
        ret
```

En retour eax contient l'entrée de la table, bx le nombre d'entrée.

Well... vous êtes arrivé à la fin de cet essai. Si vous pouvez penser maintenant : Wow, tous ces programmes protégés par des CRC peuvent dire 'bye bye'.
Nope. C'est vraiment trop facile de faire un anti-anti-CRC. Pour réussir avec succès le reverse d'un CRC, vous aurez besoin de connaître précisément dans quelle partie du code le CRC est calculé, et quel est l'algorithme utilisé. Une simple contre-mesure est d'utiliser deux algorithmes différents ou une combinaison avec un autre data-protection algorithm.

Quoi qu'il en soit... J'espère que tout ce travail a été intéressant et que vous avez apprécié de le lire autant que j'ai pu en avoir à l'écrire.

Mes remerciements vont aux beta-testers Douby/DREAD et Knotty Dread pour les commentaires sur mon travail, et qui m'ont permis de l'améliorer!

Pour un patcher corrigeant un CRC-32 simple, visitez ma webpages:
http://surf.to/anarchriz -> Programming -> Projects

Pour plus d'infos sur DREAD visitez http://dread99.cjb.net

Si vous avez toujours des questions, vous pouvez m'écrire à anarchriz@hotmail.com, ou essayer les channels #DreaD, #Win32asm, #C.I.A et #Cracking4Newbies (dans cet ordre) sur EFnet (sur IRC).

CYA ALL! - Anarchriz

Index

CRC-16 Table

```
 00h   0000 C0C1 C181 0140 C301 03C0 0280 C241
 08h   C601 06C0 0780 C741 0500 C5C1 C481 0440
 10h   CC01 0CC0 0D80 CD41 0F00 CFC1 CE81 0E40
 18h   0A00 CAC1 CB81 0B40 C901 09C0 0880 C841

 20h   D801 18C0 1980 D941 1B00 DBC1 DA81 1A40
 28h   1E00 DEC1 DF81 1F40 DD01 1DC0 1C80 DC41
 30h   1400 D4C1 D581 1540 D701 17C0 1680 D641
 38h   D201 12C0 1380 D341 1100 D1C1 D081 1040

 40h   F001 30C0 3180 F141 3300 F3C1 F281 3240
 48h   3600 F6C1 F781 3740 F501 35C0 3480 F441
 50h   3C00 FCC1 FD81 3D40 FF01 3FC0 3E80 FE41
 58h   FA01 3AC0 3B80 FB41 3900 F9C1 F881 3840

 60h   2800 E8C1 E981 2940 EB01 2BC0 2A80 EA41
 68h   EE01 2EC0 2F80 EF41 2D00 EDC1 EC81 2C40
 70h   E401 24C0 2580 E541 2700 E7C1 E681 2640
 78h   2200 E2C1 E381 2340 E101 21C0 2080 E041
```

```
 80h   A001 60C0 6180 A141 6300 A3C1 A281 6240
 88h   6600 A6C1 A781 6740 A501 65C0 6480 A441
 90h   6C00 ACC1 AD81 6D40 AF01 6FC0 6E80 AE41
 98h   AA01 6AC0 6B80 AB41 6900 A9C1 A881 6840

 A0h   7800 B8C1 B981 7940 BB01 7BC0 7A80 BA41
 A8h   BE01 7EC0 7F80 BF41 7D00 BDC1 BC81 7C40
 B0h   B401 74C0 7580 B541 7700 B7C1 B681 7640
 B8h   7200 B2C1 B381 7340 B101 71C0 7080 B041

 C0h   5000 90C1 9181 5140 9301 53C0 5280 9241
 C8h   9601 56C0 5780 9741 5500 95C1 9481 5440
 D0h   9C01 5CC0 5D80 9D41 5F00 9FC1 9E81 5E40
 D8h   5A00 9AC1 9B81 5B40 9901 59C0 5880 9841

 E0h   8801 48C0 4980 8941 4B00 8BC1 8A81 4A40
 E8h   4E00 8EC1 8F81 4F40 8D01 4DC0 4C80 8C41
 F0h   4400 84C1 8581 4540 8701 47C0 4680 8641
 F8h   8201 42C0 4380 8341 4100 81C1 8081 4040
```

CRC-32 Table

```
 00h   00000000 77073096 EE0E612C 990951BA
 04h   076DC419 706AF48F E963A535 9E6495A3
 08h   0EDB8832 79DCB8A4 E0D5E91E 97D2D988
 0Ch   09B64C2B 7EB17CBD E7B82D07 90BF1D91

 10h   1DB71064 6AB020F2 F3B97148 84BE41DE
 14h   1ADAD47D 6DDDE4EB F4D4B551 83D385C7
 18h   136C9856 646BA8C0 FD62F97A 8A65C9EC
 1Ch   14015C4F 63066CD9 FA0F3D63 8D080DF5

 20h   3B6E20C8 4C69105E D56041E4 A2677172
 24h   3C03E4D1 4B04D447 D20D85FD A50AB56B
 28h   35B5A8FA 42B2986C DBBBC9D6 ACBCF940
 2Ch   32D86CE3 45DF5C75 DCD60DCF ABD13D59

 30h   26D930AC 51DE003A C8D75180 BFD06116
 34h   21B4F4B5 56B3C423 CFBA9599 B8BDA50F
 38h   2802B89E 5F058808 C60CD9B2 B10BE924
 3Ch   2F6F7C87 58684C11 C1611DAB B6662D3D

 40h   76DC4190 01DB7106 98D220BC EFD5102A
 44h   71B18589 06B6B51F 9FBFE4A5 E8B8D433
 48h   7807C9A2 0F00F934 9609A88E E10E9818
 4Ch   7F6A0DBB 086D3D2D 91646C97 E6635C01

 50h   6B6B51F4 1C6C6162 856530D8 F262004E
 54h   6C0695ED 1B01A57B 8208F4C1 F50FC457
 58h   65B0D9C6 12B7E950 8BBEB8EA FCB9887C
 5Ch   62DD1DDF 15DA2D49 8CD37CF3 FBD44C65

 60h   4DB26158 3AB551CE A3BC0074 D4BB30E2
 64h   4ADFA541 3DD895D7 A4D1C46D D3D6F4FB
```

```
 68h   4369E96A 346ED9FC AD678846 DA60B8D0
 6Ch   44042D73 33031DE5 AA0A4C5F DD0D7CC9

 70h   5005713C 270241AA BE0B1010 C90C2086
 74h   5768B525 206F85B3 B966D409 CE61E49F
 78h   5EDEF90E 29D9C998 B0D09822 C7D7A8B4
 7Ch   59B33D17 2EB40D81 B7BD5C3B C0BA6CAD

 80h   EDB88320 9ABFB3B6 03B6E20C 74B1D29A
 84h   EAD54739 9DD277AF 04DB2615 73DC1683
 88h   E3630B12 94643B84 0D6D6A3E 7A6A5AA8
 8Ch   E40ECF0B 9309FF9D 0A00AE27 7D079EB1

 90h   F00F9344 8708A3D2 1E01F268 6906C2FE
 94h   F762575D 806567CB 196C3671 6E6B06E7
 98h   FED41B76 89D32BE0 10DA7A5A 67DD4ACC
 9Ch   F9B9DF6F 8EBEEFF9 17B7BE43 60B08ED5

 A0h   D6D6A3E8 A1D1937E 38D8C2C4 4FDFF252
 A4h   D1BB67F1 A6BC5767 3FB506DD 48B2364B
 A8h   D80D2BDA AF0A1B4C 36034AF6 41047A60
 ACh   DF60EFC3 A867DF55 316E8EEF 4669BE79

 B0h   CB61B38C BC66831A 256FD2A0 5268E236
 B4h   CC0C7795 BB0B4703 220216B9 5505262F
 B8h   C5BA3BBE B2BD0B28 2BB45A92 5CB36A04
 BCh   C2D7FFA7 B5D0CF31 2CD99E8B 5BDEAE1D

 C0h   9B64C2B0 EC63F226 756AA39C 026D930A
 C4h   9C0906A9 EB0E363F 72076785 05005713
 C8h   95BF4A82 E2B87A14 7BB12BAE 0CB61B38
 CCh   92D28E9B E5D5BE0D 7CDCEFB7 0BDBDF21

 D0h   86D3D2D4 F1D4E242 68DDB3F8 1FDA836E
 D4h   81BE16CD F6B9265B 6FB077E1 18B74777
 D8h   88085AE6 FF0F6A70 66063BCA 11010B5C
 DCh   8F659EFF F862AE69 616BFFD3 166CCF45

 E0h   A00AE278 D70DD2EE 4E048354 3903B3C2
 E4h   A7672661 D06016F7 4969474D 3E6E77DB
 E8h   AED16A4A D9D65ADC 40DF0B66 37D83BF0
 ECh   A9BCAE53 DEBB9EC5 47B2CF7F 30B5FFE9

 F0h   BDBDF21C CABAC28A 53B39330 24B4A3A6
 F4h   BAD03605 CDD70693 54DE5729 23D967BF
 F8h   B3667A2E C4614AB8 5D681B02 2A6F2B94
 FCh   B40BBE37 C30C8EA1 5A05DF1B 2D02EF8D
```

Références

Un guide indolore sur les algorithmes de détection d'erreur CRC :

Url : ftp://ftp.adelaide.edu.au/pub/rocksoft/crc_v3.txt

J'ai aussi employé le source aléatoire d'un algorithme CRC-32 pour mieux comprendre l'algorithme :

cherchez 'CRC.ZIP' ou 'CRC.EXE' ou quelque chose de ce genre avec un ftpsearch (http://ftpsearch.lycos.com?form=advanced)

(c) 1998,1999 par Anarchriz(C'est VRAIMENT la dernière ligne :)

Translated by christal. Septembre 2000