# Exploitation des collisions MD5

Ghosts In The Stack
http://www.ghostsinthestack.org

TranceFusion

**Résumé**

Le MD5 est un algorithme de hachage bien connu qui génère un hash de taille fixe, quelque soit la taille des données originales. Il est donc normal qu'il existe des collisions, c'est à dire des données distinctes mais qui produisent le même hash en sortie. Cet article explique comment exploiter ces collisions, en générant deux fichiers packés de même taille et de même hash MD5, mais ayant des comportement tout à fait différents lorsqu'on les extraits. Les applications sont considérables...

## Table des matières



# 1 Principe

## 1.1 La théorie

Comme tout algorithme de hashage, le MD5 peut être victime de collisions. En effet, cegenre d'algorithme retourne une chaîne d'octets (hash) de taille fixe, même si le fichier original est plus long; il est donc surjectif. Il est donc théoriquement possible de trouver deux fichiers possédant des contenus différents mais qui retournent le même hash une fois passés à la moulinette de l'algorithme. Dans un premier temps, la question est : comment trouver ces collisions ?

En fait, des chercheurs se sont déjà posés la question bien avant la rédaction de cet article... et en ont déjà trouvé. Nous allons voir par la suite qu'il suffit d'en connare une (c'est à dire un couple de données ayant le même hash) pour produire autant que l'on veut. La recherche de collisions, longue et fastidieuse, nous sera donc épargnée.

Dans la suite, et pour simplifier, nous appelerons *vecteur* une suite d'octets. Nous noterons ces octets en héxadécimal. Les opérations sur les vecteurs sont les opérations habituelles, mais nous en retiendrons ici deux seulement :

- +, qui désigne la concaténation de deux vecteurs. Exemple : si v1 = (01, 2d, f2) et v2 = (05) alors v1 + v2 = (01, 2d, f2, 05) , et v2 + v1 = (05, 01, 2d, f2). Comme vous le constatez, v1 + v2 != v2 + v1 ; le + est une opération dite "non commutative".
- **md5(v)**, où v désigne un vecteur quelconque. Cette opération est simplement la fonction de hashage de l'algorithme MD5, et retourne un autre vecteur qui est le hash proprement dit.

Le principe de base sur lequel cet article se base est le suivant, et il est important qu'il soit bien compris :

**Si x, y et q sont trois vecteurs quelconques, et si md5(x) = md5(y), alors md5(x+q) = md5(y+q).**

Autrement dit, si nous avons deux vecteurs x et y qui ont le même hash md5, et si nous ajouttons à chacun de ces vecteurs un troisième vecteur q, alors les hash md5 des deux vecteurs résultats resteront égaux ! Il suffit donc d'avoir x et y, et nous serons capables de produire autant de collisions que nous voudrons.

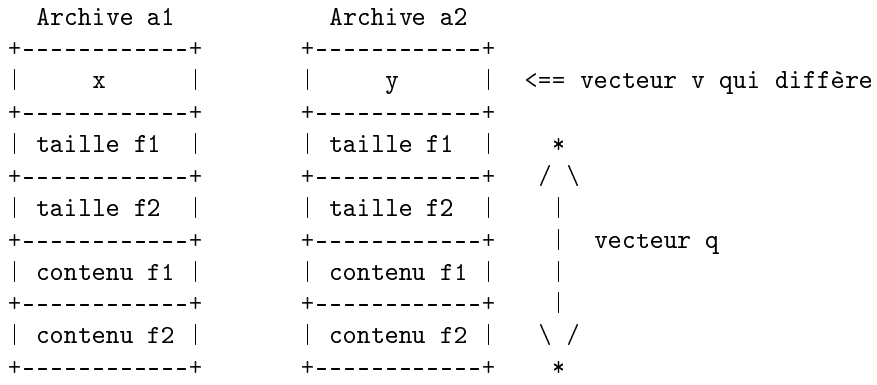
**Attention**, comme nous l'avons vu, le + n'est pas commutatif. Ainsi, la relation : "si md5(x) = md5(y) alors md5(q+x) = md5(q+y)" est **FAUSSE** ! x et y doivent impérativement être à gauche du +, donc en tête des vecteurs.

## 1.2 La pratique

Ce que l'on aimerait faire avec ces collisions serait de produire deux fichiers exécutables de même hash, masi ayant des comportements différents. Cependant, comme nous l'avons vu, la relation de base impose que les deux vecteurs x et y de même hash doivent en tête des fichiers. Et normalement, en tête d'un fichier exécutable, on trouve le header... L'idéal serait donc d'avoir deux headers différents de même hash. Malheureusement, c'est difficile à obtenir. Nous allons donc procéder différemment, et utiliser un format spécial (une sorte d'archive) que nous allons créer. Nous allons produire deux archives et dans chacune nous placerons en tête un des vecteurs x et y de même hash. Ensuite, le contenu restant des deux archives devra être identique.

Nous allons coder deux programmes. Tout d'abord un packer qui construira les deux archives (de même hash) en fonction de deux fichiers f1 et f2. Ces deux fichiers n'ont pas le même hash, et ont des comportement différents. Nous pouvons supposer par exemple que f1 est un fichier bienveillant et que f2 est malveillant :) ... Ensuite, nous allons concevoir un unpacker qui servira à extraire les fichiers des archives. C'est au moment de l'extraction que le bon fichier (f1 ou f2) sera extrait, en fonction du vecteur de tête.

Voici le format d'archive que nous allons utiliser :




Nous pouvons bien vérifier que chaque archive est de la forme v + q avec v qui est soit x, soit y. Les deux archives ont donc le même hash. On notera aussi que les deux archives contient à la fois le code du fichier bienveillant et du malveillant. Le but du packer que nous allons coder sera de nous produire ces deux archives en fonction de f1 et f2 (et des vecteurs x et y).

L'unpacker devra extraire f1 OU f2 de chaque archive, selon qu'on l'utilise avec l'archive a1 ou a2. Ainsi, si on l'utilsie sur a1, il devra extraire f1, et si on l'applique à a1, il extraira f2. Comment va-t-il distinguer l'archive ? Simplement en regardant certains octets de x et y. En effet, ces deux vecteurs sont différents, donc ils diffèrent au moins sur un octet. Il suffit de regarder quel est cet octet afin de savoir quelle est l'archive (a1 ou a2) et donc quel fichier extraire !

**Note :** Il est important de bien comprendre que même si a1 et a2 sont de même hash, les fichiers f1 et f2 n'ont pas le même hash ! En effet, le but de la manoeuvre est d'obtenir deux fichiers (a1 et a2) de même hash, mais qui produisent des fichiers différents à l'extraction (f1 et f2).

## 2 Codage du packer

Concernant le but et le fonctionnement du packer, tout est dit dans le paragraphe précédent. Nous allons donc traduire tout cela en algorithme. J'en profite pour préciser que j'ai développé cet outil sous Linux, mais il doit normalement être compilable sous Windows, à quelques petites modifs près. Comme on peut s'y attendre, l'algorithme proprement dit est très simple. Le voici, en langage de description :

```
/* Cette procédure pack les fichiers "f1" et "f2" en deux archives "a1" et "a2" */


/* Variables */

//les vecteurs x et y (on ne précise pas leur contenu pour le moment)
buffers vect1, vect2;

//les buffers qui vont contenir les fichiers f1 et f2
buffers buffer1, buffer2;

//tailles de f1 et f2
entiers taille1, taille2;


/* Algorithme (très simple...) */
```

```
//Recupere le contenu et la tialle en lisant f1
buffer1 = lireFichier("f1");
taille1 = getTaille("f1");

//Idem pour f2
buffer2 = lireFichier("f2");
taille2 = getTaille("f2");


//On ecrit l'archive a1
ecrireFichier("a1", vect1, taille1, taille2, buffer1, buffer2);

//Idem pour a2
ecrireFichier("a2", vect2, taille1, taille2, buffer1, buffer2);

//C'est déjà fini !
```

Comme vous le voyez, c'est quasiment du C! Nous allons détailler par la suite les fonctions lireFichier(), getTaille() et ecrireFichier().

## 2.1 Librairie d'operations sur les fichiers

Comme nous avons deux outils à coder (le packer et l'unpacker), et que dans chaque nous allons user et abuser des fonctions de lecteure / ecriture de fichiers, j'ai jugé bon de regrouper les fonctions de manipulation de fichiers dans une bibliothèque à part, que j'ai nommée "fichiers_bib.c". Voici son code :

```
/* fichiers_bib.c */

#include <stdio.h>
#include <stdlib.h>
#include <stddef.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>


#include "fichiers_bib.h"

/* Retourne la taille d'un fichier prealablement ouvert */
int getTaille(char * nom){
    FILE * fichier;
    int posDebut;
    int ret;

    fichier = fopen(nom,"rb");

    if(!fichier){
        printf("Erreur : le fichier %s est introuvable ou inaccessible en lecture.\n",nom);
        exit(-1);
    }
```

```
    //Positionnement a la fin
    if(fseek(fichier, 0, SEEK_END) != 0)
      printf("Impossible de se positionner a la fin du fichier !\n");

    //Recuperation de l'offset
    ret = ftell(fichier);

    fclose(fichier);

    return ret;
}

/* Lit un nombre voulu d'octets dans un fichier*/
char * lireOctets(FILE * fichier, int nbOctets){
    char * buffer;
    buffer = (char *) malloc(nbOctets * sizeof(char));

    if(fread(buffer, sizeof(char), nbOctets,fichier) != nbOctets)
      printf("Erreur de lecture du fichier\n");

    return buffer;
}

/* Lit un entier dans un fichier */
int lireEntier(FILE * fichier){
    int entier;

    if(fread(&entier, sizeof(int), 1,fichier) != 1)
      printf("Erreur de lecture du fichier\n");

    return entier;
}

/* Retourne le contenu d'un fichier dans un buffer alloue dynamiquement */
char * lireFichier(char * nom){
    char * buffer;
    FILE * fichier;
    int taille;

    fichier = fopen(nom,"rb");

    if(!fichier){
        printf("Erreur : le fichier %s est introuvable ou inaccessible en lecture.\n",nom);
        exit(-1);
    }

    taille = getTaille(nom);
    buffer = lireOctets(fichier,taille);

    fclose(fichier);
```

```
    return buffer;
}

/* Ecrit un entier dans un fichier */
void ecrireEntier(FILE * fichier, int entier){

    if(fwrite(&entier, sizeof(int), 1,fichier) != 1)
      printf("Erreur d'ecriture du fichier\n");

}


/* Ecrit un nombre d'octets provenant d'un buffer dans un fichier*/
void ecrireOctets(FILE * fichier, char * buffer, int nbOctets){

    if(fwrite(buffer, sizeof(char), nbOctets,fichier) != nbOctets)
      printf("Erreur de lecture du fichier\n");

}
```

Je pense qu'elle est suffisamment commentée pour être compréhensible. Son header est le suivant :

```
/* fichiers_bib.h */

#ifndef FICHIERS_BIB_H
#define FICHIERS_BIB_H

int getTaille(char * nom);

char * lireOctets(FILE * fichier, int nbOctets);

int lireEntier(FILE * fichier);

char * lireFichier(char * nom);

void ecrireEntier(FILE * fichier, int entier);

void ecrireOctets(FILE * fichier, char * buffer, int nbOctets);

#endif
```

## 2.2 le packer

Attaquons-nous au packer ! Son code est à peu de chose près le même qu'en langage de description...

```
/* main.c */

#include <stdio.h>
#include <stdlib.h>
```

```
#include <stddef.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>

#include "header.h"
#include "fichiers_bib.h"



/* Affiche l'aide */
void usage(){
    printf("Utilisation :\nmd5pack fichiersource1 fichiersource2 fichiercible1 fichiercible2\n");
    exit(0);
}


/* Ecrit un fichier binaire archive */
void ecrireFichier(char * nom, char * prefixe, int tailleprefixe, int taille1, int taille2,
      char * buffer1, char * buffer2){
   FILE * fichierOut = fopen(nom,"wb");

   if(!fichierOut){
        printf("Fichier %s inaccessible en ecriture.\n",nom);
        exit(-1);
   }

    ecrireOctets(fichierOut, prefixe, tailleprefixe);
    ecrireEntier(fichierOut, taille1);
    ecrireEntier(fichierOut, taille2);
    ecrireOctets(fichierOut, buffer1, taille1);
    ecrireOctets(fichierOut, buffer2, taille2);
   fclose(fichierOut);
}


/* Programme principal */
int main(int ac, char *av[])
{
   char * buffer1;
   char * buffer2;

   int taille1;
   int taille2;


   if(ac != 5) usage();

   buffer1 = lireFichier(av[1]);
   taille1 = getTaille(av[1]);
   buffer2 = lireFichier(av[2]);
```

```
   taille2 = getTaille(av[2]);


    ecrireFichier(av[3],vect1, sizeof(vect1), taille1, taille2, buffer1, buffer2);
    printf("Fichier %s cree.\n",av[3]);

    ecrireFichier(av[4],vect2, sizeof(vect2), taille1, taille2, buffer1, buffer2);
    printf("Fichier %s cree.\n",av[4]);

   return 0;
}
```

Son header n'a rien d'exceptionnel ; il se contente de définir les deux fameux vecteurs x et y, ou plutôt vect1 et vect2

```
/* header.h */

#ifndef HEADER_H
#define HEADER_H

/* Les vecteurs */

char vect1[]  =
{
    0xd1, 0x31, 0xdd, 0x02, 0xc5, 0xe6
    , 0xee , 0xc4 , 0x69 , 0x3d, 0x9a , 0x06
    , 0x98 , 0xaf , 0xf9 , 0x5c , 0x2f , 0xca
    , 0xb5 , /**/0x87 , 0x12 , 0x46 , 0x7e
    , 0xab , 0x40 , 0x04 , 0x58 , 0x3e , 0xb8
    , 0xfb , 0x7f , 0x89 , 0x55 , 0xad
    , 0x34 , 0x06 , 0x09 , 0xf4 , 0xb3 , 0x02
    , 0x83 , 0xe4 , 0x88 , 0x83 , 0x25
    , 0x71 , 0x41 , 0x5a, 0x08 , 0x51 , 0x25
    , 0xe8 , 0xf7 , 0xcd , 0xc9 , 0x9f
    , 0xd9 , 0x1d , 0xbd , 0xf2 , 0x80 , 0x37
    , 0x3c , 0x5b , 0xd8 , 0x82 , 0x3e
    , 0x31 , 0x56 , 0x34 , 0x8f , 0x5b , 0xae
    , 0x6d , 0xac , 0xd4 , 0x36 , 0xc9
    , 0x19 , 0xc6 , 0xdd , 0x53 , 0xe2 , 0xb4
    , 0x87 , 0xda , 0x03 , 0xfd , 0x02
    , 0x39 , 0x63 , 0x06 , 0xd2 , 0x48 , 0xcd
    , 0xa0 , 0xe9 , 0x9f , 0x33 , 0x42
    , 0x0f , 0x57 , 0x7e , 0xe8 , 0xce , 0x54
    , 0xb6 , 0x70 , 0x80 , 0xa8 , 0x0d
    , 0x1e , 0xc6 , 0x98 , 0x21 , 0xbc , 0xb6
    , 0xa8 , 0x83 , 0x93 , 0x96 , 0xf9
    , 0x65 , 0x2b , 0x6f , 0xf7 , 0x2a , 0x70
};


char vect2[] =
```

```
{
   0xd1 , 0x31, 0xdd , 0x02 , 0xc5 , 0xe6
   , 0xee , 0xc4 , 0x69 , 0x3d , 0x9a , 0x06
   , 0x98 , 0xaf , 0xf9 , 0x5c, 0x2f , 0xca
   , 0xb5 , /**/0x07 , 0x12 , 0x46 , 0x7e
   , 0xab , 0x40 , 0x04 , 0x58 , 0x3e , 0xb8
   , 0xfb , 0x7f , 0x89 , 0x55 , 0xad
   , 0x34 , 0x06 , 0x09 , 0xf4 , 0xb3 , 0x02
   , 0x83 , 0xe4 , 0x88 , 0x83 , 0x25
   ,/**/ 0xf1 , 0x41 , 0x5a , 0x08 , 0x51 , 0x25
   , 0xe8 , 0xf7 , 0xcd , 0xc9 , 0x9f
   , 0xd9 , 0x1d , 0xbd , /**/0x72 , 0x80
   , 0x37 , 0x3c , 0x5b, 0xd8 , 0x82
   , 0x3e , 0x31 , 0x56 , 0x34 , 0x8f , 0x5b
   , 0xae , 0x6d , 0xac , 0xd4 , 0x36
   , 0xc9 , 0x19 , 0xc6 , 0xdd , 0x53 , 0xe2
   , /**/0x34 , 0x87 , 0xda , 0x03 , 0xfd
   , 0x02 , 0x39 , 0x63 , 0x06 , 0xd2 , 0x48
   , 0xcd , 0xa0, 0xe9 , 0x9f , 0x33
   , 0x42 , 0x0f , 0x57 , 0x7e , 0xe8 , 0xce
   , 0x54 , 0xb6 , 0x70 , 0x80 , /**/ 0x28
   , 0x0d , 0x1e, 0xc6 , 0x98 , 0x21 , 0xbc
   , 0xb6 , 0xa8 , 0x83 , 0x93 , 0x96
   , 0xf9 , 0x65 , /**/0xab
   , 0x6f , 0xf7 , 0x2a , 0x70
};




#endif
```

Les deux veceturs se ressemblent fortement ; les différences sont notées sur vect2 par le symbole /**/.

Pour compiler :

```
$ gcc -c main.c
$ gcc -c fichiers_bib.c
$ gcc -o md5pack main.o fichiers_bib.o
```

Maintenant, testons notre packer avec deux programmes, eux aussi en C :

```
/* gentil.c */

#include <stdlib.h>
#include <stdio.h>


int main(){
    printf("Je suis gentil !\n");
}
```

```
/* mechant.c */

#include <stdlib.h>
#include <stdio.h>


int main(){
    printf("Je suis mechant !\n");
}
```

Désolé pour le manque d'originalité... On compile :

```
$ gcc -o gentil ./gentil.c
$ ./gentil
Je suis gentil !
$ gcc -o mechant ./mechant.c
$ ./mechant
Je suis mechant !
```

Ok, les deux programmes marchent. On constate au passage qu'ils n'ont pas le même hash MD5 :

```
$ md5sum gentil
8bd94852729ce6332992578c35000367  gentil
$ md5sum mechant
f2b485d7ac8cb2792789e3ce4ff69b29  mechant
```

On les passe au packer fraichement créé :

```
$ ./md5pack
Utilisation :
md5pack fichiersource1 fichiersource2 fichiercible1 fichiercible2
$ ./md5pack gentil mechant gentil.bin mechant.bin
Fichier gentil.bin cree.
Fichier mechant.bin cree.
```

Moment de vérité ! Les deux archives ont-elle le même hash ?...

```
$ md5sum gentil.bin
4db54b9789b9163de00b42e3d3135d08  gentil.bin
$ md5sum mechant.bin
4db54b9789b9163de00b42e3d3135d08  mechant.bin
```

Eh oui ! Et en plus, elles ont la même taille !

```
$ ls -l | grep gentil.bin
-rwxrwxrwx 1 root root [b]13389[/b] 2007-04-02 13:17 gentil.bin
$ ls -l | grep mechant.bin
-rwxrwxrwx 1 root root [b]13389[/b] 2007-04-02 13:17 mechant.bin
```

Il est dont dur de les différencier. La seule solution pour le détecter serait de faire un diff :

```
$ diff gentil.bin mechant.bin
Les fichiers binaires gentil.bin et mechant.bin sont différents.
```

Notre packer fonctionne donc. Passons maintenant à l'unpacker.

# 3 Codage de l'unpacker

Le packer doit extraire f1 si l'archive qu'on lui passe est a1, et f2 si c'est a2. Comme nous l'avons dit il se base sur un octet qui diffère dans x et y (vect1 et vect2). L'algorithme est très simple, et très proche du C, donc nous ne verrons que le code final du programme :

```
/* main.c */

#include <stdio.h>
#include <stdlib.h>

#include "header.h"
#include "fichiers_bib.h"

void usage(){
    printf("Utilisation :\nmd5unpack archive.bin fichier\n");
    exit(0);
}


int main(int ac, char *av[])
{
    char * vect;

    int taille; //Taille finale de l'executable (gentil ou mechant ? on ne le sait pas encore...)
    int taille1; //Taille du gentil fichier
    int taille2; //Taille du mechant fichier

    char * buffer; //Le buffer contenant le fichier a extraire
    FILE * fichier;

    if(ac != 3) usage();

    //Ouverture du fichier en lecture
    fichier = fopen(av[1],"rb");
    if(!fichier){
        printf("Erreur : le fichier %s est introuvable ou inaccessible en lecture.\n",av[1]);
        exit(-1);
    }

    //Lecture du vecteur
    vect = lireOctets(fichier,sizeof(vect1));
```

```
    //Recuperation de la taille des fichiers interieurs a l'archive
    taille1 = lireEntier(fichier);
    taille2 = lireEntier(fichier);

    //On detecte quel fichier on doit extraire
    if(vect[123] == (char) 0xab){ //Si c'est le mechant...


        //On saute le gentil
        lireOctets(fichier, taille1);

        //On lit le mechant
        buffer = lireOctets(fichier, taille2);

        taille = taille2;

    } else { //Si c'est le gentil...

        //On lit le gentil
        buffer = lireOctets(fichier, taille1);

        taille = taille1;

    }

    fclose(fichier);

    //Ouverture en ecriture du fichier cible
    fichier = fopen(av[2], "wb");
    if(!fichier){
        printf("Erreur : le fichier %s est inaccessible en ecriture.\n",av[2]);
        exit(-1);
    }

    //Ecriture du fichier
    ecrireOctets(fichier, buffer, taille);

    printf("Fichier %s extrait.\n",av[2]);

    fclose(fichier);

   return 0;
}
```

La distinction des deux archives se fait comme on le voit sur le 124ème octet de vect1 et vect2. Grâce à cet octet, on extrait soit f1, soit f2.

Pour compiler, c'est la même chose :

```
$ gcc -c main.c
$ gcc -c fichiers_bib.c
```

```
$ gcc -o md5unpack main.o fichiers_bib.o
```

Maintenant, testions-le sur nos deux archives gentil.bin et mechant.bin :

```
$ md5sum gentil.bin
4db54b9789b9163de00b42e3d3135d08  gentil.bin
$ md5sum mechant.bin
4db54b9789b9163de00b42e3d3135d08  mechant.bin
$ ./md5unpack
Utilisation :
md5unpack archive.bin fichier
$ ./md5unpack gentil.bin monexe
Fichier monexe extrait.
$ md5sum monexe
8bd94852729ce6332992578c35000367  monexe
$ ./monexe
Je suis gentil !
$ ./md5unpack mechant.bin monexe
Fichier monexe extrait.
$ md5sum monexe
f2b485d7ac8cb2792789e3ce4ff69b29  monexe
$ ./monexe
Je suis mechant !
```

Et voila! Nous avons donc réussi à obtenir deux archives de même hash, de même taille, mais possédant des comportements différents quand on les extrait.

# 4 Application

## 4.1 Scénario

C'est bien beau tout ça, mais à quoi ça peut servir en pratique?

Réponse : plein de choses... Imaginons par exemple que vous possédez un site où vous publiez des applications que vous avez développées. Pour une raison x ou y, vous les diffusez non pas sous forme exécutable, mais sous forme packée. Vous diffusez donc le packer permettant de les décompresser sur votre site.

Imaginons que vous êtes malveillant. Vous souhaitez infecter les utilsiateurs avec un rootkits lambda (backdoor, keylogger, etc). Vous développez donc dans un premier temps votre rootkit. Vous concevez ensuite une application banale censée être utile (où du moins que les utilisateurs téléchargeront). Mais avant de la diffuser, vous la packez avec le rootkit, en utilisant md5pack. Vous obtenez donc deux archives ; une qui, une fois extraite, donnera l'application classique, et une qui donnera le rootkit. Vous diffusez dans un premier temps la première, et publisez aussi son hash md5 sur votre site. En effet cette pratique est courante et permet aux utilisateurs d'être sûrs que l'application est la bonne et n'a pas été corrompue.

Les utilisateurs la téléchargent, et voient qu'elle marche bien. Puis, lorsque vous êtes prets, vous échangez l'archive "gentille" sur votre site contre la "méchante". Les utilisateurs la téléchargent, et peuvent même vérifier sa taille et son hash md5. Tout semble normal et ils ne se douteront de rien. Pourant, quand ils la passeront a l'unpacker... c'est le rootkit qu'ils lanceront !

### 4.2 La fin du MD5 ?

Cette application peut donc faire pas mal de dégats. Mais il faut toutefois relativiser :

- Le MD5 reste un assez bon algorithme de hashage pour les mots de passe. En effet, la technique décrite dans cet article s'applique dans un cadre très particulier, et permet de faire des collisions uniquement à partir de vecteurs bien définis, par ajout d'octets. Il est quand même préférable d'utiliser SHA-1 (ou même SHA-256 :)) pour obtenir une sécurité supérieure, car ces algorothmes sont réputés plus fiables, et plus lents (donc plus lourds à casser par force brute).
- Comme je viens juste de le dire, cette technique s'applique à une situation très particulière, puisque les archives produites contiennent toutes les deux le code malveillant, ce qui implique la malveillance de celui qui les a diffusées. Cette technique ne permet pas, par exemple, de produire un exécutable différent mais de même hash qu'un autre exécutable 100% sûr, donc le hash est fixé à l'avance.

## Références

je tiens à préciser que je n'ai rien inventé dans cet article. Je me suis beaucoup basé sur ces références, en particulier sur la 1ere. Mon travail a consisté à adapter cet article au C, et au français...


- Exploiting MD5 collisions (in C#), Eduardo Diaz[1]
- MD5 to be considered harmful someday, Dan Kaminsky[2]


[1] `http://www.codeproject.com/dotnet/HackingMd5.asp`
[2] `http://www.doxpara.com/md5_someday.pdf`