

S'attaquer au MD5 et SHA1

Quelles sont leurs vulnérabilités et comment s'en protéger ?

Pierre ESPENAN
Thomas DONNINGER
Quentin VALDEYRON

- Juillet 2010 -

Quel est le niveau de sécurité de MD5 et de SHA0/1, deux des algorithmes de hachage les plus utilisés au monde ? Quelles en sont leurs failles ? Comment les exploiter ? Comment s'en protéger ?

MD5 et SHA1, les deux algorithmes de hachage aujourd'hui les plus utilisés au monde. Leur utilité n'est plus à démontrer : elles permettent d'identifier des données de façon simple et rapide (hachage de mot de passe, de fichiers,...).

MD5, pour Message Digest 5, a été créé par le cryptologue Ronald Rivest (professeur au MIT et co-inventeur du RSA). Il date de 1991. La première faille est découverte en 1993, soit 2 ans après la mise au point de l'algorithme. Cette faille se confirme en 1996 et son exploitation est finalisée en 2004.

Quand à SHA1, pour Secure Hash Algorithm version 1, a été conçu par la NSA elle-même (National Security Agency - USA) en 1995. C'est une amélioration du SHA0, considéré comme faillible. Après la concrétisation des vulnérabilités du SHA0 en 2004, c'est en 2005 que les premières failles du SHA1 sont révélées.

Ils prônent aujourd'hui comme les deux algorithmes de hachage les plus répandus et les plus utilisés, autant par le grand public que par les entreprises. Ils sont pourtant vulnérables, bien souvent exploitables avec des moyens se limitant à un simple ordinateur personnel. Des failles donc accessibles par le commun des mortels, qui rendent d'autant plus dangereuse l'utilisation de ces algorithmes aujourd'hui vieillissants et obsolètes.

I – Fonctionnement

1 – MD5

A – Introduction

B – Algorithme

2 – SHA-0/1

A – Introduction

B – Algorithme

II – Attaques

1 – Les collisions

A – Introduction

B – Historique

a/ MD5

b/ SHA-1

C – Le principe

2 – Les pre-image

3 – Les rainbow tables

III – Exploitation

1 – Les collisions

2 – Les pre-image

3 – Les rainbow tables

IV - Protection

1 – Utiliser des algorithmes récents

2 – Utiliser des salts

V – Conclusion

I – Fonctionnement

1 – MD5

A - Introduction

MD5 est une fonction de hachage inventée par Ronald Rivest en 1991. Une fonction de hachage permet de calculer une empreinte de toute donnée numérique (allant d'une simple chaîne de caractères à un fichier de plusieurs giga octets). L'empreinte générée est d'une longueur de 128 octets (soit 32 caractères). MD5 est sensée être irréversible, c'est-à-dire qu'il est impossible de retrouver la séquence originelle d'après l'empreinte produite. Cette dernière est aussi unique ; une chaîne de caractère ne possède qu'une seule empreinte MD5.

Une fonction de hachage tel que MD5 a plusieurs utilités :

Elle permet par exemple de comparer un mot de passe entré par un utilisateur à un mot de passe stocké dans une base de données sans avoir à le faire transiter en clair. Il est donc transmis haché en MD5 puis comparé à l'empreinte dans la base de données. Ainsi, à aucun moment le mot de passe n'est transmis ou stocké en clair.

Une autre utilisation de MD5 consiste à publier l'empreinte d'un logiciel ou d'un fichier. Ainsi, lorsqu'un utilisateur le télécharge, il peut comparer l'empreinte fournie par l'éditeur et l'empreinte qu'il obtient sur le fichier téléchargé. Ceci afin de prévenir par exemple des corruptions lors du téléchargement ou de l'inclusion de virus au sein du fichier.

D'autres utilisations plus avancées consistent à vérifier l'intégrité d'un message de son point d'envoi jusqu'au destinataire, ou encore à contrôler l'authenticité de certificats SSL sur les sites sécurisés (HTTPS).

B – Algorithme

MD5 est inspiré de son ancêtre MD4. La structure de l'algorithme de MD5 repose sur une construction dite de « Merkle-Damgård » :

- 1 – Le fichier originel est découpé en blocs de taille égale. La taille de ces blocs est arbitraire.
- 2 – Chaque bloc est passé en entrée d'une fonction de compression, ainsi qu'un « vecteur ».
- 3 – Le résultat de la fonction de compression « n - 1 » sert de vecteur pour la fonction « n »

Pour le premier bloc, aucun vecteur n'existe. Un « vecteur d'initialisation » défini par le standard MD5 est donc utilisé.

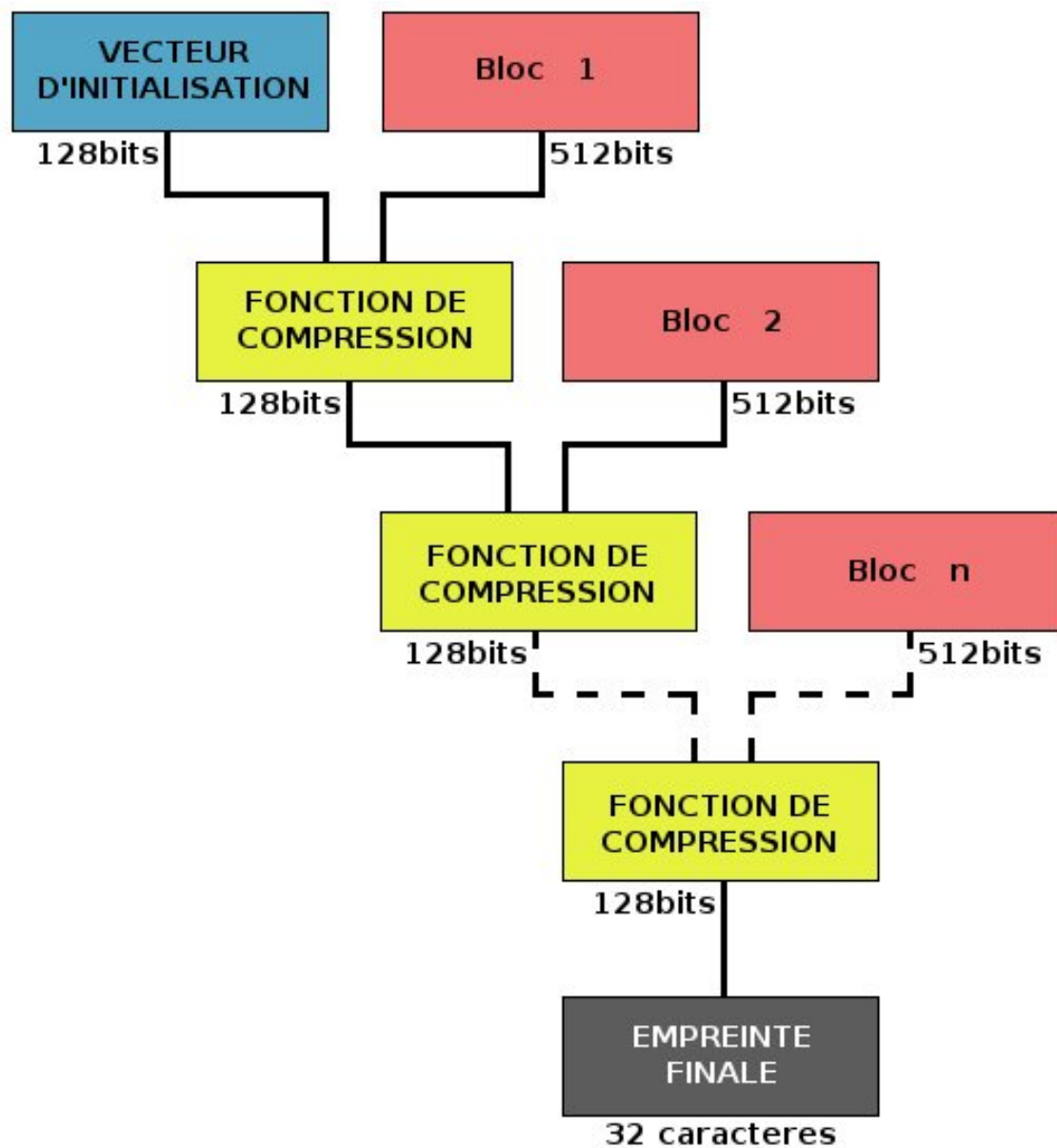
MD5 découpe le fichier en blocs de 512 bits. Chacun de ces blocs est donc passé en entrée d'une fonction de compression, en plus du résultat de la fonction de compression précédente.

Si la taille du fichier n'est pas un multiple de 512, le dernier bloc est complété par un bit égal à '1' et une série de bits égaux à '0' suffisamment grande pour atteindre le multiple suivant de 512 bits moins 64 bits. Sur ces derniers 64 bits sont écrits la taille du fichier d'origine.

De cette façon, certains blocs, comme les chaînes « chaine10 » et « chaine1 » sont reçues par la fonction de compression différemment :

- « chaine1010000000...000*tailleDuFichier* »
- « chaine1100000000...000*tailleDuFichier* »

Figure 1 – Schéma de fonctionnement de MD5



Ce schéma illustre les grandes lignes du fonctionnement de MD5 :

- 1 – Le découpage du fichier en n blocs de 512 bits
- 2 – L'utilisation d'un vecteur d'initialisation pour le premier bloc
- 3 – La création d'empreintes intermédiaires
- 4 – Le résultat final

La compression effectuée est destructive. C'est-à-dire que contrairement aux compressions habituelles, des données seront perdues, et l'information d'origine ne peut pas être retrouvée. Pour les algorithmes de hachage basés sur la construction de « Merkle-Damgård » comme MD5, leur résistance à la cryptanalyse dépend directement de la qualité des fonctions de compression.

2 – SHA-0/1

A - Introduction

SHA-0 est créé par la NSA en 1993. Suite à de premières rapides découvertes de vulnérabilités dans cette fonction, la NSA publie SHA-1 en 1995, très similaire mais complexifié. Les utilisations des fonctions de la famille SHA sont les mêmes que pour MD5.

B - Algorithme

SHA-1 et MD5 se ressemblent énormément avec cependant quelques améliorations pour SHA-1 :

- L'empreinte finale fait 160 bits (au lieu de 128 bits pour MD5)
- Les fonctions de compression sont améliorées
- De nouvelles fonctions de compression apparaissent

SHA-1 est lui aussi basé sur la construction de type « Merkle-Damgård ». Chaque donnée à hacher est découpée en blocs de 512 bits. Les fonctions de compressions sont capables de recevoir en entrée des vecteurs de 160 bits et de créer des empreintes intermédiaires de 160 bits.

II – Attaques

1 – Les collisions

A – Introduction

MD5 et SHA-1 sont des fonctions surjectives. Mathématiquement parlant, une fonction surjective est une fonction dont l'ensemble de définition ne contient pas tout l'ensemble de départ. MD5 produit une empreinte de 128 bits soit 32 caractères, quant à SHA-1 c'est une empreinte de 160 bits soit 40 caractères.

Par exemple, la chaîne « Toto va au zoo voir des lions, des girafes et des zèbres » est hachée de cette façon :

- d85395e8aa3f1b2cca3d858e35f88c19 (MD5)
- f6e2f81e8801a2d1a866eaf96a349a597b40f462 (SHA-1)

La chaîne de départ fait pourtant 56 caractères : le résultat de ces fonctions est plus court que l'entrée. Il n'existe pas une infinité d'empreintes de 32 ou 40 caractères de long mais il existe bien une infinité de possibilités de fichiers à hacher.

Il existe donc, en toute logique, des situations où deux fichiers possèdent la même empreinte. C'est une collision.

L'attaquant est en mesure de trouver deux messages différents (X , X') tels que $MD5(X) = MD5(X')$.

B – Historique

a/ MD5

1992 : Publication

1995 : De premières collisions sont découvertes dans certaines fonctions de compression de MD5

2005 : Des chercheurs réussissent à créer plusieurs document intelligibles possédants les mêmes empreintes.

2008 (Dec.) : Un groupe de chercheurs dévoilent une Autorité de Certification capable de générer des certificats SSL qui paraissent valides aux yeux des navigateurs Web (Les chercheurs ont utilisé un réseau de 200 PS3 pour mener à bien leur attaque)

Aujourd'hui : MD5 n'est plus considéré comme un algorithme résistant à la recherche de collisions.

b/ SHA-1

1995 : Publication

2005 (Jan.) : Première attaque sur une version réduite de SHA-1, demandant un calcul de 2^{80} opérations (Vincent Rijmen, Elisabeth Oswald)

2005 (Fev.) : Attaque sur la version complète de SHA-1 demandant un calcul de 2^{69} opérations (Xiaoyun Wang, Yiqun Lisa Yin, Bayarjargal, Hongbo Yu)

2005 (Aout) : L'attaque précédente est améliorée pour obtenir un calcul de 2^{63} opérations

Aujourd'hui : SHA-1 n'est plus considéré comme un algorithme résistant à la recherche de collisions.

C – Le principe

La création de deux fichiers ayant la même empreinte est relativement simple. Nul besoin de calculs lourds pour trouver l'empreinte commune : Il suffit de connaître une paire de « vecteurs » présentant une collision. Un « vecteur » est un ensemble de données destinées à être hachées.

Soit deux vecteurs X et Y présentant la même empreinte : $MD5(X) = MD5(y)$.

Soit le vecteur Q le programme dont nous voulons créer deux version différentes mais de même empreinte. Q doit contenir à la fois le code du programme sain et le code du programme malfaisant.

Il suffit de concaténer X et Q ainsi que Y et Q pour obtenir deux vecteurs X+Q et Y+Q présentant une collision : $MD5(X+Q) = MD5(Y+Q)$.

Rapportez-vous à la partie « Exploitation » pour voir des exemples concrets d'utilisation d'une telle attaque.

Afin de générer une collision il est courant d'utiliser une attaque basée sur le paradoxe des anniversaires (Birthday Attack). Cette attaque réduit considérablement le nombre d'opérations nécessaires avant de trouver une collision.

Voici le principe du paradoxe des anniversaires :

Combien faut-il de personnes dans une pièce, pour que la probabilité dépasse 50% pour que deux personnes au moins soient nées le même jour de l'année ?

Le résultat choque l'intuition (d'où le paradoxe) : il suffit d'un groupe de 23 personnes.

Le calcul est simple :

Soit E un ensemble fini de k éléments

On tire uniformément n éléments dans E

La probabilité $p(n)$ que, parmi n éléments de E , deux éléments au moins soient identiques, vaut :

$$p(n) = 1 - \frac{|E|!}{(|E| - n)!} \cdot \frac{1}{|E|^n}$$

Ainsi, $p(n)$ dépasse 50% lorsque n équivaut à 23.

De la même façon, on a une probabilité de 50% de trouver une collision lorsque $n = \sqrt{k}$.

Pour faire le parallèle avec MD5 :

MD5 génère une empreinte de 128 bits. Il y a donc $k = 2^{128}$ empreintes possibles en sortie.

On a donc une chance sur deux de trouver une collision en ne calculant plus que $\sqrt{k}$ empreintes, soit $2^{(128/2)}$ et non 2^{128} , soit 18 milliards de milliards fois moins de calculs...

2 – Les pre-image

La résistance à la recherche de pre-image et de pre-image secondaire constitue un critère de sécurité important pour un algorithme de hachage. Qu'est-ce dont qu'une pre-image ?

Recherche de pre-image

Considérons une empreinte Y donnée.

L'attaquant ne doit pas être en mesure de trouver un vecteur X tel que $\text{MD5}(X) = Y$.

(à moins de calculer l'ensemble des empreintes générables par la fonction de hachage, c'est-à-dire 2^{128} pour le MD5 et 2^{160} pour le SHA-1)

L'intérêt de cette recherche réside dans la possibilité de retrouver à partir de l'empreinte le vecteur d'origine.

Recherche de pre-image secondaire

Considérons un vecteur X donné et son empreinte correspondante Y donnée : $\text{MD5}(X) = Y$.

L'attaquant ne doit pas être en mesure de trouver un nouveau vecteur $X' \neq X$ tel que $\text{MD5}(X') = Y$.

(à moins de calculer l'ensemble des empreintes générables par la fonction de hachage, c'est-à-dire 2^{128} pour le MD5 et 2^{160} pour le SHA-1)

L'intérêt de cette recherche réside dans la possibilité de trouver un nouveau vecteur possédant la même empreinte que le premier.

MD5 et SHA-1 possèdent encore aujourd'hui une certaine résistance à la recherche de pre-image primaire et secondaire. En effet, la seule étude publiée en Avril 2009 expose une attaque

théorique permettant de trouver une pre-image complète dans l'algorithme MD5. Au vu de la puissance de calcul nécessaire pour mener à bien cette attaque ($2^{123,4}$ calculs), cette attaque est restée à l'état théorique et n'a jamais été mise en pratique. À l'heure actuelle, aucune attaque de ce genre n'a été publiée pour le SHA-1.

3 – Les rainbow-tables

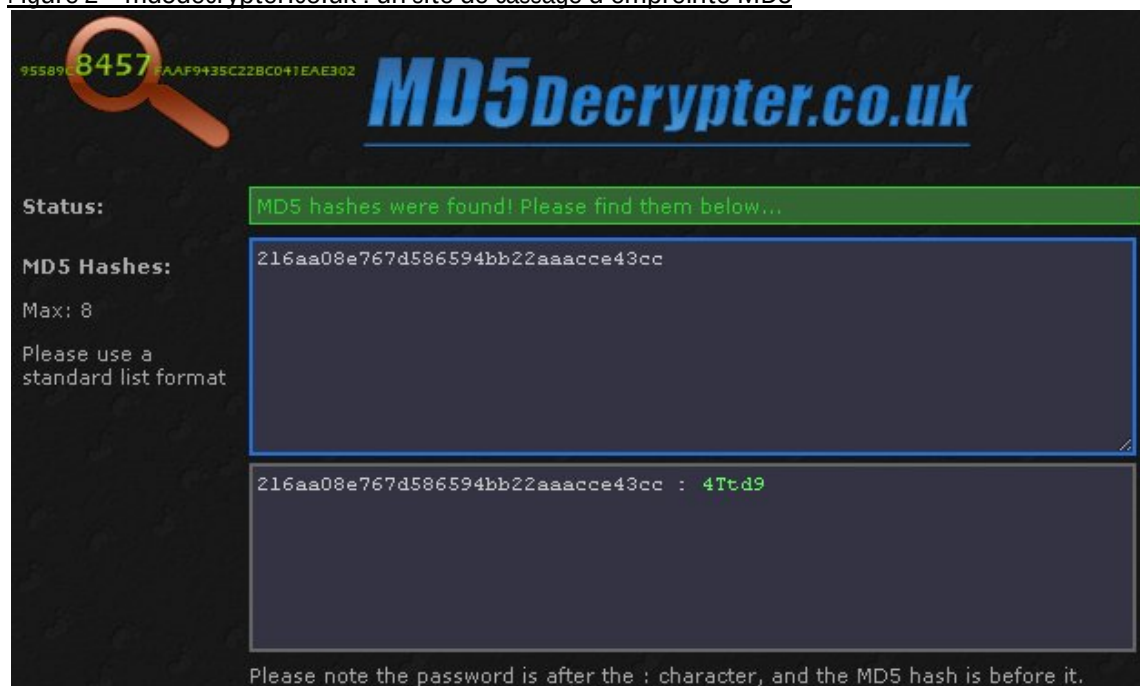
Les rainbow tables sont de simples « bases de données » qui contiennent uniquement des correspondances entre de très grandes quantités de vecteurs et leur empreinte. Ces tables sont utilisées pour des attaques de type « brute-force » sur une empreinte déjà existante (recherche de pre-image secondaire par brute-force).

Chaque empreinte de la contenue dans la rainbow table est comparée à l'empreinte que l'on cherche à casser, en espérant trouver une correspondance. Le brute-force basique consisterait à calculer une à une l'empreinte de chaque possibilité. L'attaque par rainbow table est donc un brute-force un peu plus intelligent. En effet les empreintes sont générées à l'avance en très grande quantité, ce qui rend ensuite la recherche bien plus rapide puisqu'aucun calcul n'est nécessaire. Le temps de cassage est réduit au seul temps de comparaison des empreintes.

Aujourd'hui des rainbow tables de plus de 1To proposent quelques 3500 millions d'empreintes pré calculée. Même si ce chiffre peut paraître impressionnant, il faut garder à l'esprit qu'il ne représente qu'à peine plus de $1 \cdot 10^{(-29)}$ % du nombre total d'empreintes possibles en MD5. Cela dit, les rainbow tables sont surtout utilisées pour casser des empreintes de mots de passe. Ceux-ci étant plus courts que des fichiers classiques, une rainbow table de cette ampleur en contient en grande quantité.

Il existe sur Internet une multitude de sites proposant des téléchargements gratuits de rainbow tables, voire même d'accès direct via un simple formulaire afin de tenter de casser une empreinte à l'occasion :

Figure 2 – md5decrypter.co.uk : un site de cassage d'empreinte MD5



III – Exploitations

1 – Les collisions

Dans quels cas peut-on exploiter les collisions MD5 ?

Un premier exemple d'utilisation serait la diffusion de logiciels malveillants. La première étape consiste à publier (via son site web, des réseaux torrent, ...) un logiciel ordinaire. Après une certaine période, un deuxième exécutable est créé, possédant la même empreinte que le premier exécutable. Cette fois-ci, l'exécutable est un logiciel malveillant.

L'utilisateur ne pouvant rien remarquer, se fie au contrôle d'intégrité MD5 et pense être en présence d'un logiciel sûr.

Voici un exemple de logiciel banal :

```
#include <iostream>

using namespace std;

int main()
{
    cout << "Hello world ! I'm a nice software !\n" << endl;
    return 0;
}
```

Et un exemple de logiciel malveillant :

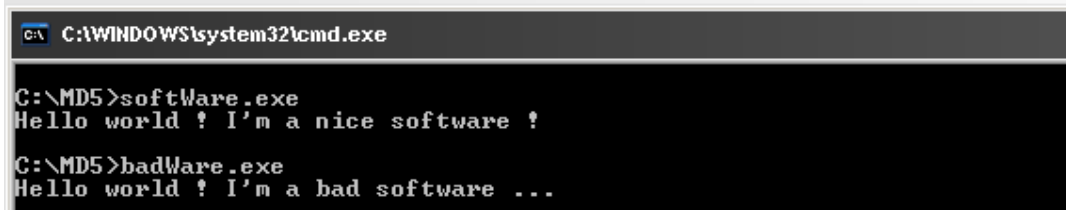
```
#include <iostream>

using namespace std;

int main()
{
    cout << "Hello world ! I'm a bad software :D\n" << endl;
    return 0;
}
```

Ces deux logiciels sont compilés en tant que softWare.exe et badWare.exe

Figure 3 – Exécution de softWare.exe et de badWare.exe



Vient ensuite le code du générateur de collision. Il utilise une paire de vecteurs possédants la même empreinte (`static byte[] vec1` & `static byte[] vec2`), auquel il concatène les programmes softWare.exe et badWare.exe

Voici le code :

```
using System;
using System.IO;
using System.Security.Cryptography;
using System.Text;
using System.Threading;

namespace Md5Generate
{
    class Generator
    {
        /// Point d'entrée de l'application
        [STAThread]
        static void Main(string[] args)
        {
            if (args.Length != 2)
            {
                Console.WriteLine("Usage: md5clone good_file evil_file");
                Environment.Exit(-1);
            }

            byte[] software = ReadBinaryFile(args[0]);
            byte[] badWare = ReadBinaryFile(args[1]);

            WriteBinary("software.bin", vec1, software, badWare);
            WriteBinary("badware.bin", vec2, software, badWare);
        }

        /// Ecriture du fichier binaire.
        /// Le fichier est une concaténation du préfixe, de software et
de badWare
        private static void WriteBinary (string sOutFileName, byte[]
prefix, byte[] software, byte[] badWare)
        {
            using (FileStream fs = File.OpenWrite (sOutFileName))
            {
                using (BinaryWriter writer = new BinaryWriter(fs))
                {
                    writer.Write(prefix);
                    writer.Write(software.Length);
                    writer.Write(badWare.Length);
                    fs.Write(software, 0, software.Length);
                    fs.Write(badWare, 0, badWare.Length);
                    fs.Close();
                }
            }
        }

        /// Read a file and return the data in binary format
        private static byte[] ReadBinaryFile (string sFileName)
        {
            byte[] data;
            using (FileStream fs = File.OpenRead (sFileName))
            {
                data = new byte[fs.Length];
                fs.Position = 0;
                fs.Read(data, 0, data.Length);
            }
            return data;
        }
    }
}
```

```

    /// Dump binary data in hexadecimal
    public static string ToHexa (byte[] data)
    {
        StringBuilder sb = new StringBuilder();
        for (int i = 0; i < data.Length; i++)
            sb.AppendFormat("{0:X2}", data[i]);
        return sb.ToString ();
    }

    /// Premier vecteur préfixe
    static byte[] vec1 =
    {
        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
    };

    /// Deuxième vecteur préfixe
    static byte[] vec2 =
    {
        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, /**
        flag byte*/0xab, 0x6f, 0xf7, 0x2a, 0x70
    };
}

```

Suite à l'exécution du générateur, on obtient 2 fichiers différents *software.bin* et *badware.bin*. Ils possèdent tous les deux la même empreinte MD5.

Figure 4 – *software.bin* et *badware.bin* possèdent la même empreinte MD5

```

C:\MD5>exf software.bin badware.bin
; Checksums generated by ExactFile Console Application 1.0.1.6 BETA
; www.ExactFile.com
; 31/07/2010 19:29:11
;
0f0bf4fb3c0f530767a5f317a9b84491 *software.bin
0f0bf4fb3c0f530767a5f317a9b84491 *badware.bin
; 2 files sucessfully hashed.

```

Il est maintenant nécessaire d'écrire l'extracteur qui nous permettra d'extraire un exécutable depuis le fichier binaire .bin :

```
using System;
using System.IO;

namespace Md5Extractor
{
    class Extractor
    {
        /// Le point d'entrée de l'application.
        [STAThread]
        static void Main(string[] args)
        {
            if (args.Length == 0)
            {
                Console.WriteLine("Usage: Md5Extractor.exe file.bin output_file.exe");
                Environment.Exit(-1);
            }
            ExtractFile(args[0], args[1]);

            private static void ExtractFile (string sfilename, string
soutfile)
            {
                using (BinaryReader reader = new
BinaryReader(File.OpenRead (sfilename)))
                {
                    byte[] vec = reader.ReadBytes (128);
                    int goodSize = reader.ReadInt32 ();
                    int evilSize = reader.ReadInt32 ();
                    /// Ouvre le badWare
                    if (vec[123] == 0xab)
                    {
                        reader.ReadBytes (goodSize);
                        byte[] evil = reader.ReadBytes (evilSize);
                        using (BinaryWriter writer = new
BinaryWriter(File.OpenWrite (soutfile)))
                        {
                            writer.Write (evil);
                            writer.Close ();
                        }
                    }
                    /// Ou alors, ouvre le softWare
                    else
                    {
                        byte[] good = reader.ReadBytes (goodSize);
                        using (BinaryWriter writer = new
BinaryWriter(File.OpenWrite (soutfile)))
                        {
                            writer.Write (good);
                            writer.Close ();
                        }
                    }
                    Console.WriteLine ("File written on {0}",
soutfile);
                }
            }
        }
    }
}
```

On exécute l'extracteur sur le premier fichier binaire software.bin :

Figure 5 – software.bin est extrait

```
C:\MD5>Md5Extractor.exe software.bin logiciel_public.exe
File written on logiciel_public.exe

C:\MD5>logiciel_public.exe
Hello world ! I'm a nice software !
```

Le logiciel légitime est bien extrait et est exécutable. Au tour maintenant de la nouvelle version du logiciel, badware.bin d'être extraite :

Figure 6 – badware.bin est extrait

```
C:\MD5>Md5Extractor.exe badware.bin logiciel_public.exe
File written on logiciel_public.exe

C:\MD5>logiciel_public.exe
Hello world ! I'm a bad software ...
```

Le logiciel se comporte donc de deux façons différentes, alors que chacune de ses 2 sources possèdent la même empreinte MD5 !

Un autre exemple d'utilisation malintentionnée des collisions MD5, est la falsification de certificats SSL. En Décembre 2008, deux chercheurs (Alex Sotirov & Jacob Appelbaum) annoncent avoir réussi à créer une fausse autorité de certifications de certificats SSL (Rogue CA). Concrètement, cette Rogue CA permet de créer des certificats pour site web protégés en SSL (HTTPS), possédants les mêmes empreintes que les certificats émis par les sociétés de confiances telles que VeriSign, RapidSSL, FreeSSL ou encore RSA Data Security. Plusieurs exploitations d'un tel travail sont possibles, comme déclarer un site de phishing sécurisé avec une url en https://, publier des applications Java dangereuses mais digne de confiance, collecter des données sur un réseau via une attaque de Man In The Middle, ...

Cela dit, le procédé de l'attaque en question n'a pas été révélé, et les chercheurs ont déclaré avoir eu recours à un cluster de 200 Playstation 3 pour fournir une puissance de calcul suffisante. Autant dire qu'une telle attaque n'est pas à la portée de tout le monde.

2 – Les pre-image

Comme vu précédemment, il est encore difficile d'exploiter la vulnérabilité de type pre-image pour le MD5 et encore impossible pour le SHA-1. Les puissances de calcul nécessaires sont trop importantes, même pour les supercalculateurs actuels.

Cependant, en attendant l'aboutissement de nouvelles recherches sur le sujet, le calcul de pre-images reste théoriquement possible. Cette faille est d'ailleurs bien plus critique que de simples collisions.

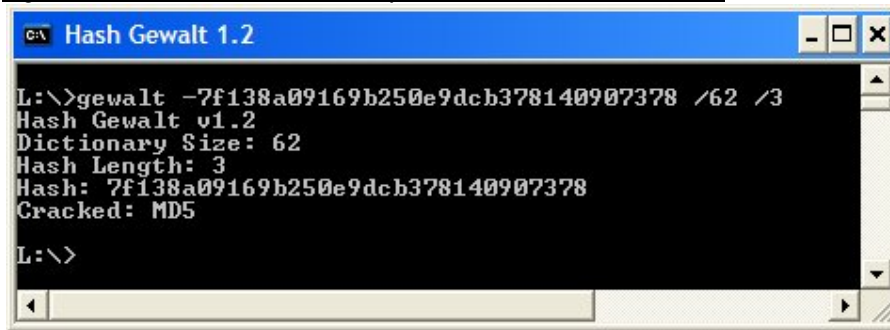
3 – Les rainbow tables

De nombreux logiciels permettent de parcourir des rainbow tables à la recherche de l'empreinte recherchée, et pour cause : ces logiciels sont relativement simples à développer. Cela dit, vu les temps calculs gargantuesques qui sont en jeux, mieux vaut préconiser les logiciels les plus performants.

John the Ripper est sans doute le plus répandu parmi les utilisateurs, de part sa réputation et son très grand panel de fonctionnalités.

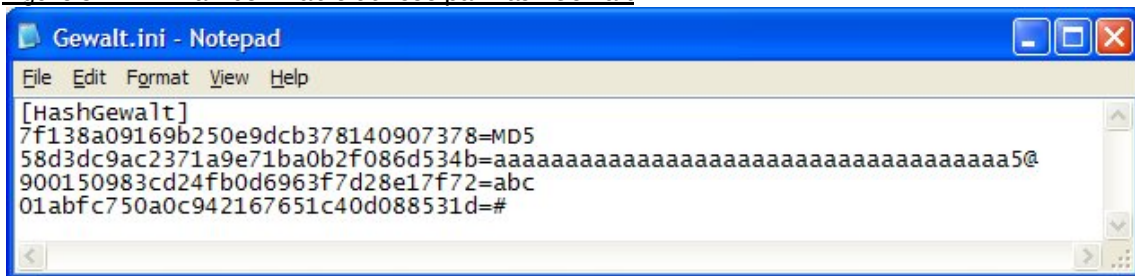
D'autres petits logiciels tel Hash Gewalt, moins connus, sont plus simples d'utilisation. C'est une bonne alternative pour s'initier à ce genre de techniques.

Figure 7 – Hash Gewalt casse l'empreinte de la chaîne « MD5 »



```
c:\ Hash Gewalt 1.2
L:\>gewalt -7f138a09169b250e9dcb378140907378 /62 /3
Hash Gewalt v1.2
Dictionary Size: 62
Hash Length: 3
Hash: 7f138a09169b250e9dcb378140907378
Cracked: MD5
L:\>
```

Figure 8 – Mini Rainbow Table utilisée par Hash Gewalt



```
Gewalt.ini - Notepad
File Edit Format View Help
[HashGewalt]
7f138a09169b250e9dcb378140907378=MD5
58d3dc9ac2371a9e71ba0b2f086d534b=aaaaaaaaaaaaaaaaaaaaaaaaaaaaa5@
900150983cd24fb0d6963f7d28e17f72=abc
01abfc750a0c942167651c40d088531d=#
```

IV – Protection

Il est heureusement possible de contrer les attaques dont sont victimes MD5 et SHA-1, et ce, de façon particulièrement efficace et simple.

1 – Utiliser des algorithmes récents

Il existe aujourd'hui une panoplie d'algorithmes fiables, en particulier les évolutions de SHA-1, appartenant à la sous-famille SHA-2 (SHA-224, SHA-256, SHA-384, SHA-512, ...) ou d'autres comme Whirlpool, RIPEMD-256 (et supérieur).

Ces algorithmes là ne connaissent à l'heure actuelle pas la moindre faille. Les cryptanalyses montrent qu'ils sont beaucoup moins sensibles aux collisions que leurs prédécesseurs.

Et enfin ils peuvent générer une quantité d'empreintes trop importante pour être à la portée d'une attaque par bruteforce, via les moyens techniques modernes. Cela complexifie aussi l'utilisation de rainbow tables (tables plus lourdes, plus longues à générer, ...)

Il ne sert à rien de s'acharner à vouloir corriger les erreurs du passé, mettez à jour !

2 – Utiliser des salts

Il existera toujours des utilisateurs qui auront pour mot de passe « toto », « 1234 », « abc », « medor »... Malheureusement, quel que soit l'algorithme utilisé, ce genre de mot de passes est très vulnérable aux rainbow tables. C'est pourquoi il est préconiser d'utiliser des « salts » :

Lors du stockage du mot de passe dans la base de données, il va être concaténé avec une chaîne de caractère, choisie arbitrairement, appelée un salt (par analogie au petit grain de sel/sable qui vient se glisser dans le système). Plus le salt est complexe, plus les chances d'éviter une attaque par rainbow tables est faible.

Par exemple, le mot de passe « 1234 » concaténé à un salt « Q^?%Yr; » ne se retrouvera sûrement pas dans une rainbow table.

Le concept peut encore être poussé pour éviter d'obtenir plusieurs fois la même empreinte dans une même base de données. Ainsi, un mot de passe concaténé à un salt arbitraire et au login du membre lui donnera un caractère unique :

« Q^?%Yr; » . « 1234 ». « login »

Les salts empêchent aussi une personne, mal intentionnée, qui a un accès à la base d'ajouter de nouveaux utilisateurs sans connaître le salt. En effet, si un nouveau compte est créé en encodant le mot de passe sans le salt, l'empreinte stockée ne correspondra pas avec l'empreinte du mot de passe utilisé lors de la connexion, concaténé au salt !

V – Conclusion

Pour conclure, il est aujourd'hui communément admis que les fonctions de hachage MD5 et SHA-1 sont obsolètes. Elles sont déconseillées d'utilisation au profit de fonctions plus récentes et plus performantes en particulier dans la famille SHA.

Nombreux sont les systèmes utilisant encore ces fonctions caduques, et ceci en mettant en péril leur propre sécurité ainsi que la sécurité de leurs utilisateurs.