Practical Attacks on Digital Signatures Using MD5 Message Digest

Ondrej Mikle

Department of Software Engineering at Faculty of Mathematics and Physics,
Charles University, Prague, Czech Republic
Ondrej.Mikle@Gmail.com

December 2, 2004

Abstract. We use the knowledge of the single MD5 collision published by Wang et al. [2] to show an example of a pair of binary self-extract packages with equal MD5 checksums, whereas resulting extracted contracts have fundamentally different meaning. Secondly, we demonstrate how an attacker could create custom pair of such packages containing files arbitrarily chosen by the attacker with equal MD5 sums where each of the package extracts different file. Once the algorithm for finding MD5 collisions is published, attack could be made even more effective as we explain further. Authors of [2] claim to know such algorithm for any MD5 initialization vector. A real-world scenario of such attack is outlined. Finally, we point out the consequences resulting from such attack for signature schemes based on MD5 message digest on an example using GPG.

Keywords: collision, hash function, MD5

1 Introduction

MD5 is very common hash function designed by Ronald Rivest [1]. Nowadays, it is being commonly used for file integrity checking and as a message digest in digital signature schemes.

During CRYPTO 2004 Conference, a team of Chinese cryptographers have shown that they know an algorithm for finding a pair of colliding messages, i.e. different messages with the same MD5 sum [2]. Mentioned algorithm is very fast, finding collision in about an hour, as the authors claim. This fact means that MD5 is no longer secure as a way for checking file (message) integrity, since two colliding files can be created while the hash remains unchanged.

In section 2 we will use the example pair of colliding strings provided by Chinese cryptographers in [2] to create a pair of meaningful files with identical hash, though with fundamentally different meaning. We show one specific example, section 3 describes a way of creating a pair of archives containing arbitrary files of attacker's choice with the same MD5 sum. We then clarify how publishing the algorithm opens a way for even more sophisticate attack (section 4). Last two sections mentioned go deeper into technical details explained using C++ code. Next, we demonstrate the impact on digital signatures on an example using GPG (section 5) and finally an example of real-world attack is illustrated in the sphere of software distribution (section 6). Last section sums up what was achieved (section 7).

An archive containing the executables, source code and data files mentioned in this paper can be downloaded at [3]. Archive description can be found in appendix A.

2 Working Example

We used self-extract archive for the purposes of this demonstration of the collisions. Self-extract archives are commonly being used (e.g. to pack larger files, for software distribution). Reasons for choosing self-extract archives are discussed in appendix B.

From the user's point of view the situation is: user receives two files - self-extract.exe and data.pak. He can check MD5 sum of both files. User runs self-extract.exe and the program using data.pak extracts the document itself - contract.pdf. Other user receives the same self-extract.exe, but different data.pak. Both data.pak files are created so that their MD5 sum is identical. Therefore, both users think that the contracts extracted are the same in both cases.

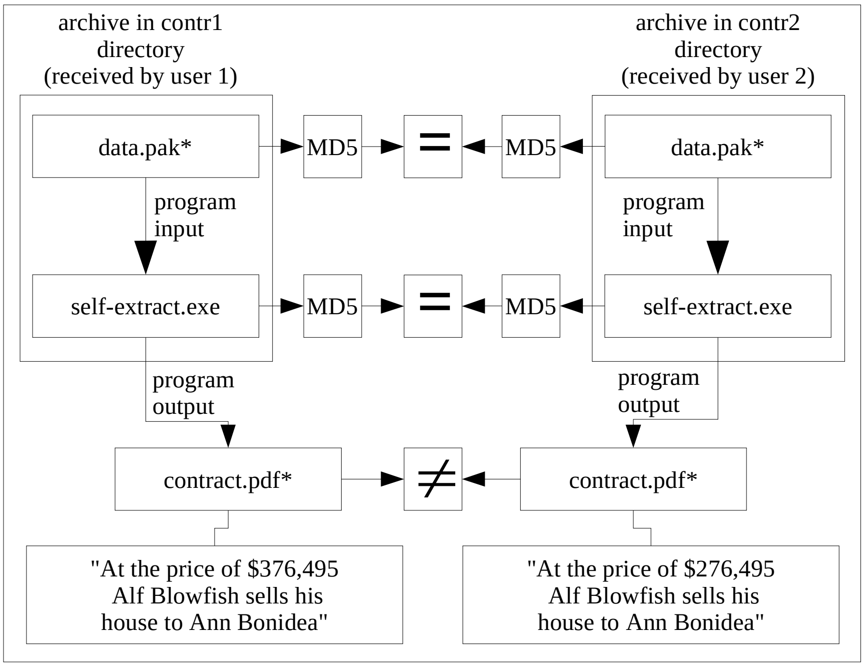
archive in contr1 directory (received by user 1)
archive in contr2 directory (received by user 2)
data.pak*
MD5
=
MD5
data.pak*
program input
program input
self-extract.exe
MD5
=
MD5
self-extract.exe
program output
program output
contract.pdf*
≠
contract.pdf*
"At the price of $376,495 Alf Blowfish sells his house to Ann Bonidea"
"At the price of $276,495 Alf Blowfish sells his house to Ann Bonidea"

Fig. 1. Both executables and data files have the the same MD5 hash, though running those programs result in different contracts extracted.
*) Files have identical names, but different contents

Now we will describe how to get such results using colliding strings from [2]. In the archive at homepage [3], there are two directories: contr1 and contr2. They represent files what one user and the other user receive, respectively. To check the program yourself, enter each of the directory and run self-extract.exe (on Windows platform) or compile and run self-extract using Makefile (on *nix platform).

Example self-extract.exe program reads 'data.pak' file, uses one specific bit in colliding block as an index to the table to decide which file to extract. Blocks causing collision are located at the beginning of each of the 'data.pak' files. Using one or other data.pak in the archive (in directories contr1 and contr2, see appendix A), you will get two completely different contracts extracted. As you can check yourself, both files, executable and the data file have the same MD5 sum - see figure 1 (example contracts are inspired by [6]).

The code is simple (and written in C++), it would be easy to port it to other system.

3 Tools for Creating Custom Pair of Archives

This section describes series of steps how an attacker can create a custom pair of self-extract archives where each extracts two different arbitrary files of attacker's choice. Created data.pak files have identical MD5 sum.

The archive at homepage [3] contains a program named create-package. For *nix platforms, it can be compiled from source using enclosed Makefile, for Windows platform, there is precompiled binary named create-package.exe. Usage of this program is:

```
create-package outfile infile1 infile2
```

Where: *outfile* tells how the extracted file should be named (after using self-extract program), *infile1* and *infile2* denote two arbitrary files to be stored in the package. This program then creates files data1.pak and data2.pak that should be put into contr1 and contr2 directory in the archive, respectively and renamed to data.pak. Running self-extract program afterwards will create file named *outfile* once containing data of *infile1* or *infile2*. E.g.:

```
create-package contract.pdf contract1.pdf contract2.pdf
```

will take contract1.pdf and contract2.pdf, put them into data1.pak and data2.pak. Those data.pak's when used with self-extract, will create a file named contract.pdf, once containing data of contract1.pdf, second time data of contract2.pdf. You can check for yourself that both files will have the same MD5 sum.

Table 1. Layout of data.pak file

Size in bytes	Data stored
128	colliding block
1	filename length - *fnamelen*
fnamelen	filename to be extracted
4	32-bit integer size of first stored file - *filesize1*
4	32-bit integer size of second stored file - *filesize2*
filesize1	data of file1
filesize2	data of file2

It works by using the layout of data.pak file as described in table 1. The colliding block is different for each of the data.pak files. It is exactly one of the pair of binary strings supplied by Chinese cryptographers [2]. The rest of data in both data.pak files is always identical.

When computing MD5 sum of data.pak files, the first colliding block (first 1024 bits) causes the hash context to become identical. Since the remaining data really are equal, it results in an equal MD5 hash.

The self-extract program decides which file to extract based on one bit from the colliding block (which happens to be different in the data.pak files), the bit used is defined as:

```
#define MD5_COLLISION_OFFSET 19
#define MD5_COLLISION_BITMASK 0x80
```

These two values denote exactly the position and mask of the differing bit. *MD5_COLLISION_OFFSET* is the index of the differing byte in the data.pak file. As you can see, it lies in the first 128 bytes of colliding block. The self-extract.exe executable file itself is the same in both packages, therefore has always the same MD5 hash.

The relevant code is:

```
uint8_t colliding_byte;

//seek to and read the byte where MD5 collision occurs
packedfile.seekg(MD5_COLLISION_OFFSET, ios::beg);
packedfile.read((char *)&colliding_byte, 1);
```

Then, decision which file to extract looks like:

```
//use boolean value of colliding byte vs. bitmask
//comparison as index to filetable
unsigned int fileindex = (colliding_byte &
    MD5_COLLISION_BITMASK) ? 1 : 0;
```

Now the program knows which file to extract, seeks to the right position in packed file and extracts the right amount of bytes into output file.

4 Improving the Attack

In the previous section we have seen an attack requiring two files. We had to use the strings that are known to cause collision in MD5 hash. It restricted us to put the colliding block at the beginning of a file. Once the algorithm for finding MD5 collisions for any initialization vector is known, colliding blocks can be put at any 512-bit block position in file. Such knowledge allows for a pair of single-file executables to be created, both having identical MD5 sum, though each of them doing arbitrary different actions of attacker's choice. Two users now receive only two single self-extract files with equal MD5 sum. Each of them obviously will extract different files (see figure 2). Commonly used self-extract executables are in a single file, so it would be less suspicious.

The only requirement is that such block must be aligned at 512-bit boundary in the file (this requirement is a consequence of the MD5 algorithm design).

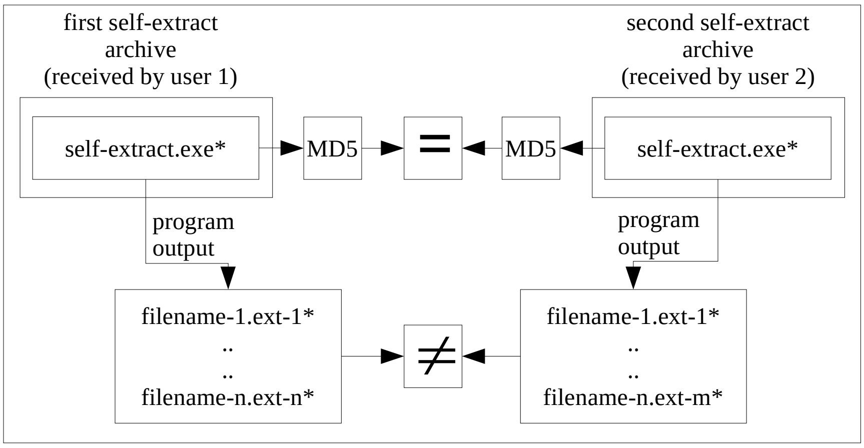

Fig. 2. Now only one self-extract executable is required. Both self-extract executables have the the same MD5 hash, though running those programs result in different contracts extracted. *) Files (may) have identical names, but different contents

Note that this does not limit an attacker to create solely self-extract executables, any kind of executables could be made this way.

4.1 Executable File Structure

For deeper understanding we explain executable file structure. When some source code is compiled and linked, the resulting executable has layout as described in table 2 (simplified).

Table 2. Layout of an executable file

Header	Static data	Machine code

Header usually contains platform-specific data used by operating system's executable loader (e.g. relocations, code start position). Static data contain variables and their values that were known at linking time. Machine code is generated by compiler and linker from source code.

The key idea is to put the colliding block into the static data section where it is accessible by the program code. Let's suppose that the colliding block is 1024 bits long (assumption is based on the Chinese attack [2]). A code snippet accomplishing the conditions might be (see md5-poc-in-middle.cpp for complete code):

```
#define MD5_COLLISION_BLOCK_BITS 1024
#define MD5_COLLISION_BLOCK_ALIGN 512
#define MAGIC_OFFSET 0x41424344
#define MAGIC_BITMASK 0x45464748
#define MAGIC_BLOCK 0x494a4b4c
```

```
unsigned char MD5_CollisionBlock[
(MD5_COLLISION_BLOCK_BITS+MD5_COLLISION_BLOCK_ALIGN)/8
] = {MAGIC_BLOCK, 0, 0, 0};

unsigned int MD5_CollisionOffset = MAGIC_OFFSET;
unsigned int MD5_CollisionBitMask = MAGIC_BITMASK;
```

First, note the "magic" constants. Their sole purpose is to be able to find quickly their position in the resulting executable file (some linkers may be able to tell the location of linked structures directly, this is in case they do not). These variables are static and linker will put them into static data section.

MD5_CollisionBlock variable is an array 1024+512 bits long. We need to use only 1024 bits of them, the rest is padding so that the colliding block will fit at 512-bit boundary.

MD5_CollisionOffset and *MD5_CollisionBitMask* determine uniquely the position of the differing bit in the *MD5_CollisionBlock* array.

Then, we can access the bit in question and make decisions based on it like this:

```
bool decision()
{
return (MD5_CollisionBlock[MD5_CollisionOffset] &
MD5_CollisionBitMask);
}

void some_function()
{
if (decision()) do_good_thing(); else do_bad_thing();
}
```

After compiling and linking this program, we find the positions of all three described variables in resulting executable file (using hex editor, for example) and update them as necessary. At first we compute the colliding blocks to be put into the *MD5_CollisionBlock* array:

1. Let I denote the index of first byte in executable file (counting from 0) where colliding blocks should be placed
2. We take bytes indexed 0..I-1 of the executable file and start computing MD5. The result is hash context H
3. Using H as initialization vector value we compute the collision blocks
4. The colliding blocks are put at the index I of executable file

Then we point *MD5_CollisionOffset* and *MD5_CollisionMask* to the right location. We assume the linker will put the variables into the executable file in the the order we declared them. That will allow changing the offset and bitmask without recalculating the collision blocks.

This exploit is very simple. Such code may cause suspicion. However, avoiding suspicion is not that hard and can be worked around in various ways: many companies release only proprietary software without source code, code can be obfuscated in many ways, e.g. using buffer overflows to read different variables than the code seems to do, incurring very rare deliberate race conditions (that the attacker knows how to trigger), using the differing bits as a part of computation somewhere (e.g. as an index).

5 Example of Attacking Digital Signature Using GPG

Let's take our example from section 2 and try signing it. Each file (self-extract, data.pak) is signed separately using MD5 algorithm for message digest. GPG 1.2.4 for Linux [5] was used for the demonstration. See appendix and archive at homepage [3], where you can find the signatures and keys used in this demonstration. The passphrase for the secret key is "verysimplepassphrase" (without quotes).

At first we create a secret key:

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

We enter these data for the key (RSA as signature algorithm, 1024-bit modulus size):

```
Real name: Collision test key
Email address: collision@md5.abc
Comment: Used to demonstrate signature collision when
using MD5 algorithm
```

We will be signing both data.pak and self-extract files used in section 2.

In both directories contr1, contr2 we sign the data.pak and self-extract file and create a detached signature (it will be stored in data.pak.asc or self-extract.asc, respectively):

Command:

```
gpg -u collision --digest-algo md5 -ab -s data.pak
gpg -u collision --digest-algo md5 -ab -s self-extract
```

For your convenience, here is the parameters' meaning:

-u selects key to sign with

--digest-algo selects message digest algorithm

-a means ASCII output (instead of binary)

-b means to create "detached signature" (i.e. the resulting file will only contain signature without the original file)

-s stands for "sign" command

The last parameter is file to sign.

Now, if we test it against the file we signed, it obviously works:

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

Parameter meaning:

-v tells gpg to be verbose

--verify is the verify command

first filename is the signature file

second filename is the file to verify

Output:

```
gpg: armor header: Version: GnuPG v1.2.4 (GNU/Linux)
gpg: Signature made Tue Nov 23 21:54:28 2004 CET using
RSA key ID 3E200834
gpg: Good signature from "Collision test key (Used to
demonstrate signature collision when using MD5
algorithm) <collision@md5.abc>"
gpg: binary signature, digest algorithm MD5
```

However, if we test it against the other data.pak in contr2 directory, it will claim that the signature is good, though the file is different!

```
gpg -v --verify data.pak.md5.asc ../contr2/data.pak
```

Output:

```
gpg: armor header: Version: GnuPG v1.2.4 (GNU/Linux)
gpg: Signature made Tue Nov 23 21:54:29 2004 CET using
RSA key ID 3E200834
gpg: Good signature from "Collision test key (Used to
demonstrate signature collision when using MD5
algorithm) <collision@md5.abc>"
gpg: binary signature, digest algorithm MD5
```

Testing both self-extract files in both directory will work, since they are identical:

```
gpg -v --verify self-extract.asc self-extract
```

Output:

```
gpg: armor header: Version: GnuPG v1.2.4 (GNU/Linux)
gpg: Signature made Tue Nov 23 21:55:02 2004 CET using
RSA key ID 3E200834
gpg: Good signature from "Collision test key (Used to
demonstrate signature collision when using MD5
algorithm) <collision@md5.abc>"
gpg: binary signature, digest algorithm MD5
```

Testing the other self-extract in contr2 directory using signature from contr1 directory:

```
gpg -v --verify self-extract.asc ../contr2/self-extract
```

Output:

```
gpg: armor header: Version: GnuPG v1.2.4 (GNU/Linux)
gpg: Signature made Tue Nov 23 21:55:04 2004 CET using
RSA key ID 3E200834
gpg: Good signature from "Collision test key (Used to
demonstrate signature collision when using MD5
algorithm) <collision@md5.abc>"
gpg: binary signature, digest algorithm MD5
```

6 Real-world Attack Scenario

Now it is clear that there is possibility to create two custom self-extract executable binaries containing any arbitrary files having equal MD5 sums. Take a web-browser software as an example.

Suppose a packager in a company creates self-extract packages intended for distribution. Once the web-browser's development is finished, all of the web-browser's files are given to packager to create installation scripts and create a package ready for distribution. Packager creates the scripts and the package itself. The package is then sent to testing department to check whether it passes the tests and whether the web-browser can be installed. If all tests are passed, the package is good and will be digitally signed. Testing department will sign the package as to prove that the software was tested, passed the tests and is ready for distribution. Both web-browser package, its MD5 sum and signature is then put on company's ftp or web page for download.

Now suppose the packager is a dishonest person (an attacker). So he/she creates a pair of packages with equal MD5 sums, one containing the original files and the other package deliberately flawed. The good package is sent for testing, passes tests and gets signed using MD5 hash as message digest. Later it's put on ftp/web page for download. The attacker as an insider has access to company's servers (either legitimate or not), so he/she replaces the good package with flawed package.

The MD5 sum and digital signature will hold even for the flawed package. If the attacker is clever, he/she will modify the original software only slightly and in a discreet way. A very rare and obfuscated race condition that only he/she knows how to trigger is a good example. The flawed web browser will be downloaded and installed. Since MD5 sum and signature holds and the software does not act suspiciously, it can take a long time until the flaw is detected (if ever). All mischief after discovering the flaw would fall upon testing department's head, since their signature guarantees that the software was well-tested.

Now the attacker can for example sell the knowledge of the flaw to spammers, virus-writers, create own web-pages which will infect computers using the flawed browser or hack into vulnerable web-servers and slightly modify the pages adding own code that triggers the flaw in browser and causes infection, later using infected computers to send spam, launch distributed denial of service attacks, etc.

7 Conclusion

At present, the Chinese attack [2] is sometimes not taken very seriously, since it cannot generate second preimage collisions. In this paper we have clearly demonstrated, how dangerous can be just the use of the pair of published MD5 collisions. Further, we have proved that the possibilities will increase after publishing details of the Chinese attack [2]. In other words, it will become more difficult to distinguish attack from legitimate situation.

In particular, we have shown example and method that only by using the known colliding strings it is possible to create two different arbitrary documents (chosen by attacker) having identical digital signature after all. Presented examples together with versatile method should lead to replacement of MD5 hash function.

Acknowledgement

This work was inspired by Vlastimil Klima's lecture [4], who has also assisted reviewing this paper.

References

1. Rivest, R.: "The MD5 Message-Digest Algorithm", RFC 1321, April 1992, ftp://ftp.rfc-editor.org/in-notes/rfc1321.txt
2. X. Wang, D. Feng, X. Lai, H. Yu, "Collisions for Hash Functions MD4, MD5, HAVAL-128 and RIPEMD", rump session, CRYPTO 2004, *Cryptology ePrint Archive*, Report 2004/199, http://eprint.iacr.org/2004/199
3. Homepage of this project: http://cryptography.hyperlink.cz/2004/collisions.htm
4. Klima, V.: "Hash functions, MD5 and Chinese attack", Seminar "Security of Information Systems", Faculty of Mathematics and Physics, Charles University in Prague, November 22, 2004, http://cryptography.hyperlink.cz/2004/Hasovaci_funkce_a_cinsky_utok_MFFUK_2004.pdf
5. The GNU Privacy Guard - GnuPG, http://www.gnupg.org
6. Dobbertin, H.: "Cryptanalysis of MD4", Journal of Cryptology, 1998, p. 253-271, http://www.ruhr-uni-bochum.de/itsc/download/KryptIISS04/Cryptanalysis_MD4.pdf

Appendix A: Structure of Archive from Homepage

The archive md5-poc.zip contains one directory named 'MD5-proof-of-concept' with the following files (can be downloaded at [3]):

- *Makefile* - makefile to compile the sources on the *nix platforms
- *self-extract.cpp* - source of the self-extract program from section 2
- *md5-poc-in-middle.cpp* - source code for the executable described in section 4
- *create-package.cpp* - source code for the custom package creation program used in section 3
- *collision.h* - two blocks causing MD5 collision written as a C array
- *collision.key.asc* - secret key used for demonstration in section 5. Passphrase for the key is "verysimplepassphrase" (without quotes)
- *create-package.exe* - Windows executable of the package creation program
- *Readme.1st* - some notes on using the archive

- *contr1* - directory containing one version of self-extract archive
- *contr1/data.pak* - data itself, the same MD5 hash as data.pak in contr2 directory
- *contr1/self-extract.exe* - Windows executable that extracts data.pak, it is identical to self-extract.exe from the contr2 directory
- *cont1/data.pak.asc* - GPG signature of data.pak using MD5 message digest

- *contr2* - directory containing other version of self-extract archive
- *contr1/data.pak* - data itself, the same MD5 hash as data.pak in contr1 directory
- *contr1/self-extract.exe* - Windows executable that extracts data.pak, it is identical to self-extract.exe from the contr1 directory
- *cont1/data.pak.asc* - GPG signature of data.pak using MD5 message digest

Appendix B: Selecting the Appropriate File Format

There are many existing commonly used file formats to choose from. The decision which one to use is based upon following criteria:

- two colliding messages provided differ only in 6 bits, that must be enough to create a significant difference in meaning (requirement is a consequence of the Chinese attack published in [2])
- other 1018 bits of message must not affect interpretation of crafted message in an undesirable way
- we have to expect that colliding strings will consist of arbitrary data from a software application's point of view

We chose the executable format due to a number of reasons:

- all platforms have some sort of executable binary format
- proposed way of construction is platform-independent (at least for all platforms we are aware of)
- most applications are in binary format and are distributed in some sort of binary archive (self-extracting executables are very common)
- any file type (e.g. contract in PDF format) can be put into a self-extracting executable and signed
- last and most important, executable format is very generous in terms of being able to craft a special executable with desired properties