Steve Friedl's Unixwiz.net Tech Tips
An Illustrated Guide to Cryptographic Hashes

With the recent news of weaknesses in some common security algorithms (MD4, MD5, SHA-0), many are wondering exactly what these things are: They form the underpinning of much of our electronic infrastructure, and in this Guide we'll try to give an overview of what they are and how to understand them in the context of the recent developments.

But note: though we're fairly strong on security issues, we are **not** crypto experts. We've done our best to assemble (digest?) the best available information into this Guide, but we welcome being pointed to the errors of our ways.

What is a cryptographic hash?

A "hash" (also called a "digest", and informally a "checksum") is a kind of "signature" for a stream of data that represents the contents. The closest real-life analog we can think is "a temper-evident seal on a software package": if you open the box (change the file), it's detected.

Let's first see some examples of hashes at work.

Many Unix and Linux systems provide the **md5sum** program, which reads a stream of data and produces a fixed, 128-bit number that summarizes that stream using the popular "MD5" method. Here, the "streams of data" are "files" (two of which we see directly, plus one that's too large to display).

```
$ cat smallfile
This is a very small file with a few characters

$ cat bigfile
This is a larger file that contains more characters.
This demonstrates that no matter how big the input
stream is, the generated hash is the same size (but
of course, not the same value). If two files have
a different hash, they surely contain different data.

$ ls -l empty-file smallfile bigfile linux-kernel
-rw-rw-r--    1 steve    steve           0 2004-08-20 08:58 empty-file
-rw-rw-r--    1 steve    steve          48 2004-08-20 08:48 smallfile
-rw-rw-r--    1 steve    steve         260 2004-08-20 08:48 bigfile
-rw-r--r--    1 root     root      1122363 2003-02-27 07:12 linux-kernel

$ md5sum empty-file smallfile bigfile linux-kernel
d41d8cd98f00b204e9800998ecf8427e  empty-file
75cdbfeb70a06d42210938da88c42991  smallfile
6e0b7a1676ec0279139b3f39bd65e41a  bigfile
c74c812e4d2839fa9acf0aa0c915e022  linux-kernel
```

This shows that *all* input streams yield hashes of the same length, and to experiment, try changing just one character of a small test file: you'll find that even very small changes to the input yields very large changes in the value of the hash (though the *size* of the generated hash remains constant).

Hashes are "digests", not "encryption"

This is a common confusion, especially because all these words are in the category of "cryptography", but it's important to understand the difference.

Encryption transforms data from a cleartext to ciphertext **and back** (given the right keys), and the two texts should roughly correspond to each other in size: big cleartext yields big ciphertext, and so on. "Encryption" is a **two-way** operation.

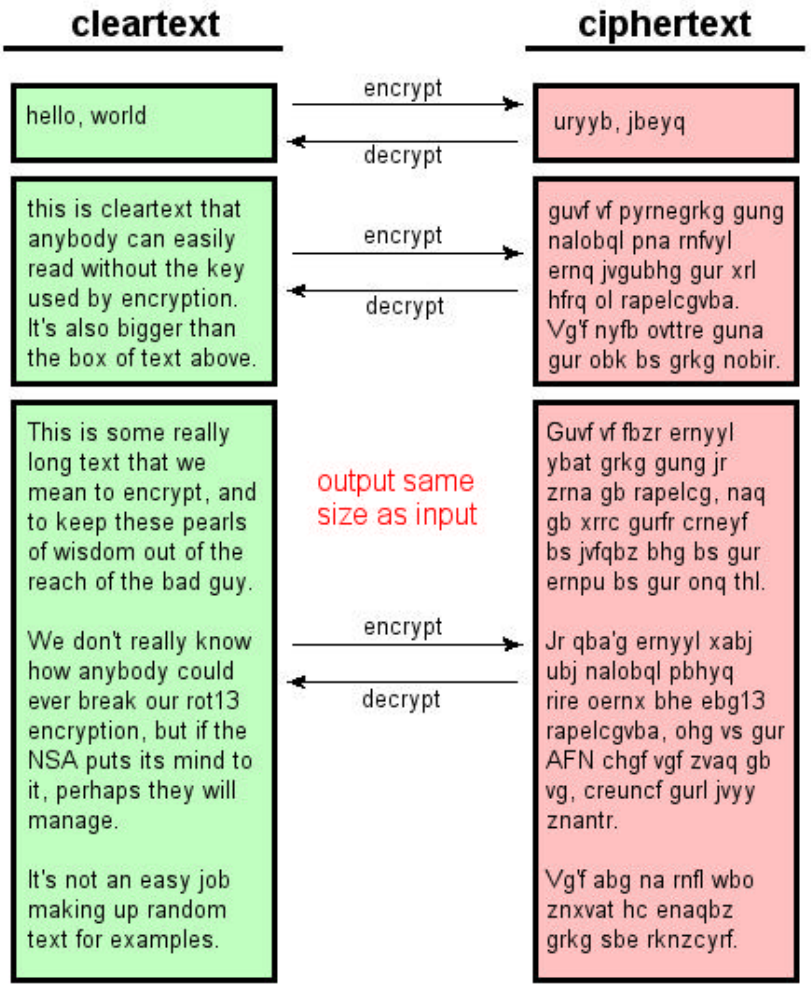
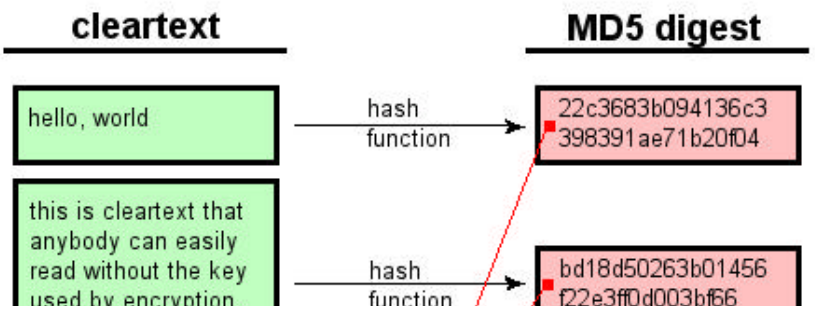


Fig. 1: Encryption - a two-way operation

Hashes, on the other hand, compile a stream of data into a small **digest** (a summarized form: think "Reader's **Digest**"), and it's strictly a **one way operation**. All hashes of the same type - this example shows the "MD5" variety - have the same size no matter how big the inputs are:



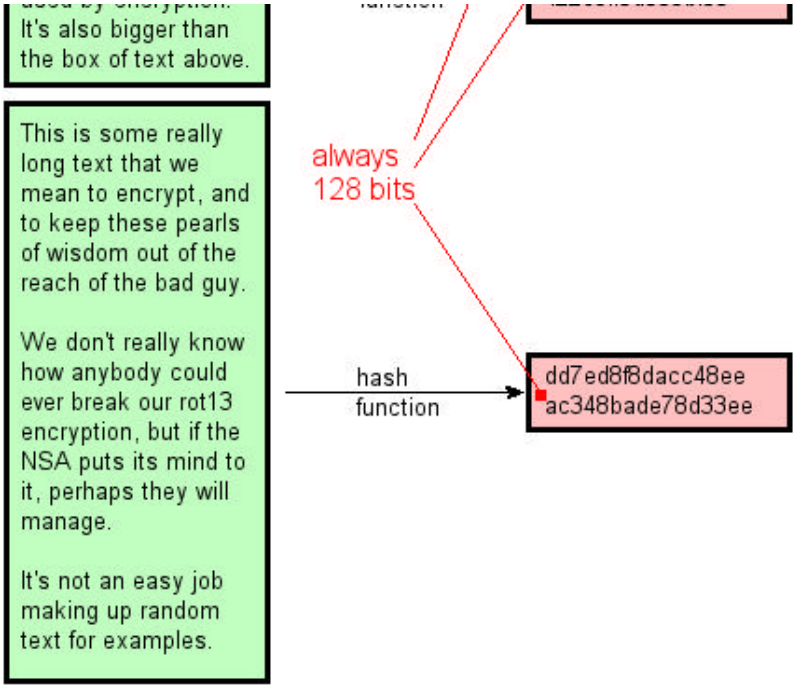


Fig. 2: Hashing - a one-way operation

"Encryption" is an obvious target for attack (e.g., "try to read the encrypted text without the key"), but even the one-way nature of hashes admits of more subtle attacks. We'll cover them shortly, but first we must see how hashes fit in the overall framework.

We'll note here that though hashes and digests are often informally called "checksums", they really aren't. Traditional checksums, such as a Cyclic Redundancy Check are designed to catch data-transmission errors and not deliberate attempts at tampering with data. Aside of the small output space (usually 32 bits), they are not designed with the same properties in mind. We won't mention "regular" checksums again.

How are hashes used?

There are quite a few uses for hashes, and we'll mention them here so we have something to build on when we try to subvert them for evil purposes, and to see the ramifications of the recent developments.

Verifying file integrity

The most obvious use is "verifying file integrity". If you have just downloaded a large piece of software from a website, how do you know that you've received it correctly and that it has not been tampered with?

One way is to download the file again and compare the bits: if the bits are the same, you're probably ok, but if they're different, which ones are the *right* bits? Finding out means yet another download with compare, and this gets very tedious very quickly.

Instead, if the website publishes the hash values of its download bundles, you can check it yourself. For instance, the ProFTPD project (an excellent open source FTP server) publishes their hashes:

Highly configurable GPL-licensed FTP server software

MD5 sums and PGP signatures of release files

MD5 Digest Hashes

```
417e41092610816bd203c3766e96f23b proftpd-1.2.8p.tar.bz2
abf8409bbd9150494bc1847ace06857a proftpd-1.2.8p.tar.gz
7c85503b160a36a96594ef75f3180a07 proftpd-1.2.9.tar.bz2
445fbf24e2ec300800a184eb81296bda proftpd-1.2.9.tar.gz
d834bb822816a2ce483cc2ef1a9533e7 proftpd-1.2.10rc3.tar.bz2
1e306d2f54ea92895ecff6659498b911 proftpd-1.2.10rc3.tar.gz
```

Fig. 3: MD5 checksums for downloadable software

Now it's just a matter of running the **md5sum** command on the file you downloaded and comparing it with the published values.

Important!

When considering hash values, ***close doesn't count!*** (just like horseshoes)

If the hashes being compared differ in *any* way, **even by just a single bit**, the data being digested is not the same! There is no equivalent of "roundoff error" or "almost" in cryptographic hashes.

Hashing passwords

A second example that's very common - but less obvious - is the hashing of access passwords, and we'll reiterate that "hashing is not encryption".

It's a bad idea for computer systems to store passwords in cleartext (in their original form), because if the bad guy can somehow get to where they're stored, he gets all the passwords. Knowing how many people foolishly use one password at multiple sites, getting a stash from one system may give access to others.

A more secure way is to store a **hash of the password**, rather than the password itself. Since these hashes are **not reversible**, there is no way to find out "what password produced this hash?" - and the consequence of a compromise is much lower.

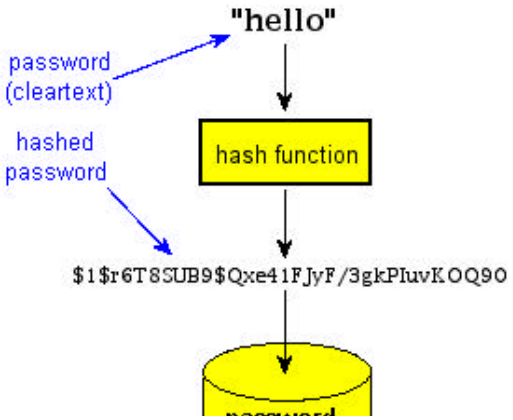


Fig. 4: Storing a hash instead of a password

Now we have squirreled away the password into a safe place, but since this is a one-way function, how will we know that some future user at a login prompt gives us the same password?

The answer is simple: we take the proposed password -- in cleartext -- run it through the same hash function, and see if this result matches the hash we've saved in the password store. If they match, the user *must* have known the proper password so access is granted, but if the hashes are not identical, access is denied.

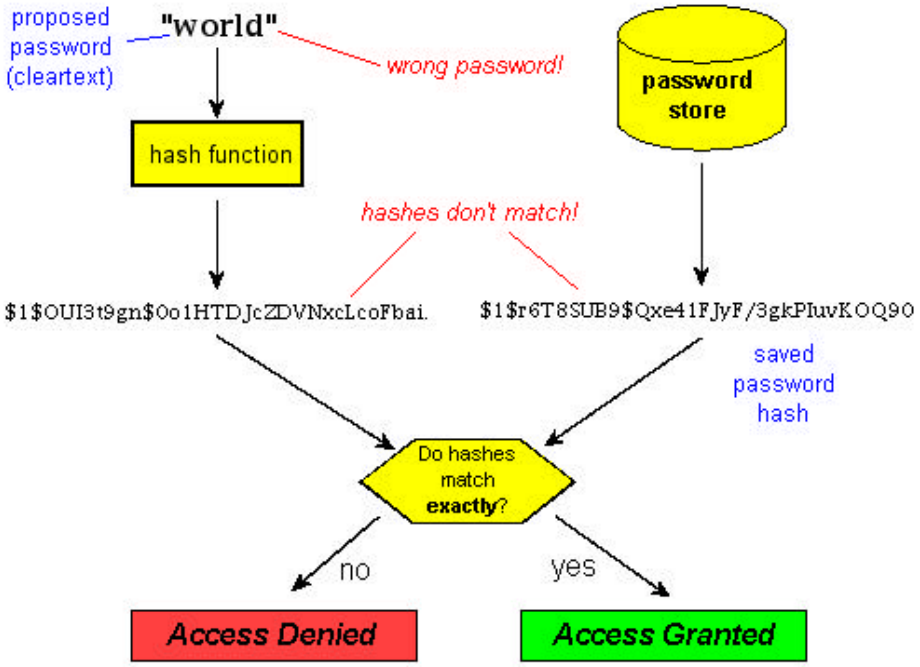


Fig. 5: Testing a proposed password against the stored hash

Digitally Signed Documents

This is quite a bit more complicated, and we're going to gloss over the details of this subject, covering only enough to show the role of cryptographic hashes. David Youd has written an excellent Introduction to Digital Signatures that covers this in much more detail than matters here.

Loosely speaking, "signing" a document electronically is the digital equivalent of placing an autograph on paper, and our discussion revolves around how the signature is represented. How does one know that *this* digital signature applies to *this* document?

The answer: one signs the **hash** of the document, and the result of this step (which is done by public-key encryption) is a **digital signature**. The process is shown here:

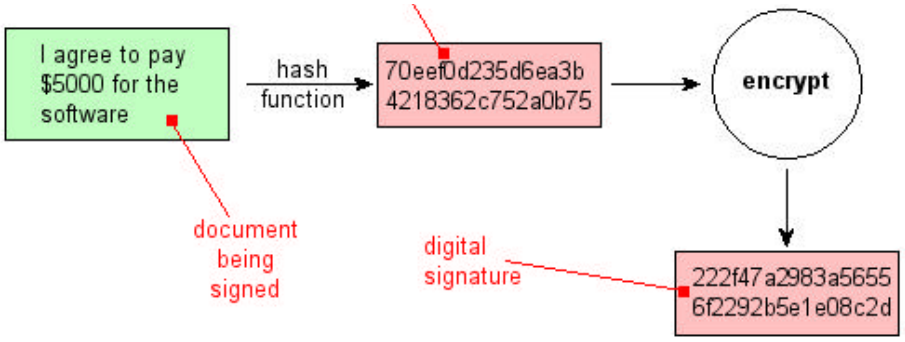


Fig 6: Digitally signing a document

At some later date, we can prove that you signed the document by decrypting the signature with your public key, which yields the hash, and showing that our document's hash matches the signed one.

We'll see later how a weakness in the hash function could lead to shenanigans.

But what about collisions?

The thoughtful reader may wonder how this could work: if it's possible to uniquely represent every possible stream of data in 128 bits - that's 16 bytes -- then why would one ever need a file larger than that? If this isn't the case, then it seems obvious that many input streams are available that can produce any given hash.

When different chunks of data produce the same hash value, this is known as a **collision**, and it follows from the last paragraph that they inherently have to exist:

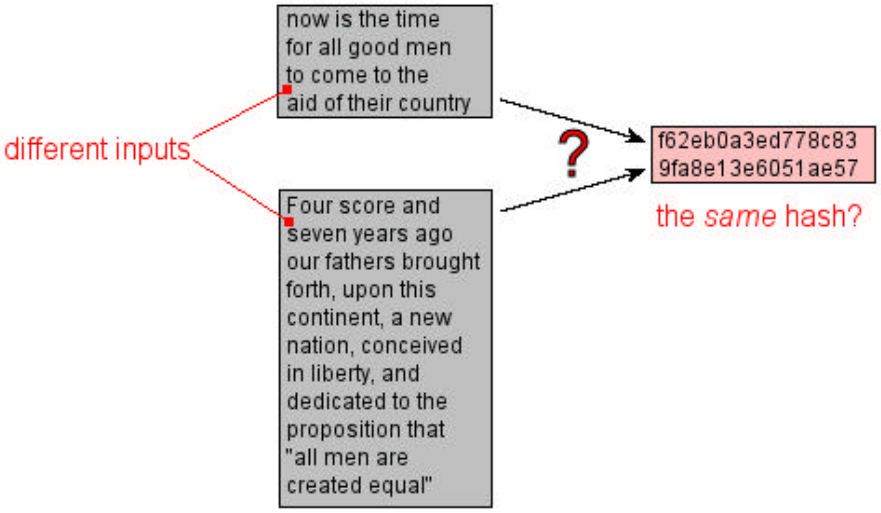


Fig 7: What a hash collision might look like (just a hypothetical example)

If so, this seems to undermine the whole premise of cryptographic hashes until one learns that ***nobody has found a collision yet*** (well, *almost* nobody, but we're getting to that) .

This astonishing fact is due to the astonishingly large number of possible hashes available: a 128-bit hash can have 3.4 x 10^{38} possible values, which is:

340,282,366,920,938,463,463,374,607,431,768,211,456 possible

hashes

If the hash algorithm is properly designed and distributes the hashes uniformly over the output space, "finding a hash collision" is much less likely than a million people correctly guessing the California Lottery numbers every day for a billion trillion years (weekends included).

Other hashes have even more bits: the SHA-1 algorithm generates 160 bits, whose output space is four billions times larger than that produced by MD5's 128 bits.

What's inside a cryptographic hash?

The first answer is "it depends on the kind of hash", but the second answer usually starts with "a lot of math". A colloquial explanation is that all the bits are poured into a pot and stirred briskly, and this is about as technical we care to delve into here.

There are plenty of resources that show the internal workings of a hash algorithm, almost all of which involve lots of shifting and rotating through multiple "rounds".

A good explanation for the SHA-1 algorithm (a popular one) can be found at Wikipedia by clicking on this image:

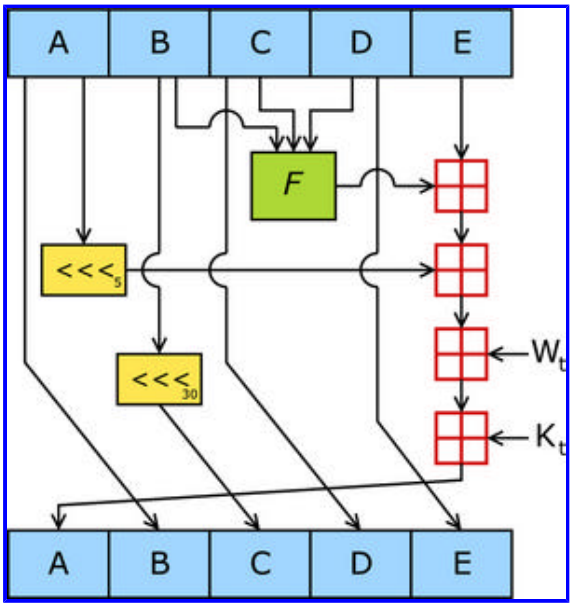


Fig. 8 - *One iteration within the SHA-1 compression function. A, B, C, D and E are 32-bit words of the state;*
F is a nonlinear function that varies;
<<< denotes left circular shift.
Kt is a constant.

Though we're not going to elaborate on the internals, it does seem appropriate to mention some of the popular hash algorithms:

- MD4 (128 bits, obsolete)
- MD5 (128 bits)
- RIPEMD-160 (160 bits)
- SHA-1 (160 bits)

- SHA-256, SHA-384, and SHA-512 (longer versions of SHA-1)

Each has its own advantages in terms of performance, several variations of collision resistance, how well its security has been studied professionally, and so on. "Picking a good hash algorithm" is beyond the scope of this Tech Tip.

"Collision resistance" in more detail

As we've mentioned several times, "collisions" play a central role in the usefulness of a cryptographic hash, mainly in the sense that their presense undermines the conclusions one can make about any given hash produced. Some algorithms are better than others at avoiding collisions, but even this is a multi-faceted attribute.

There are three related characteristics that describe the quality of a hash function's anti-collision security, and we use the term "hard" to mean "no easier than via brute-force means".

Collision Resistance

means that it's hard to find two inputs that produce the same hash value, but **you get to pick both inputs**.

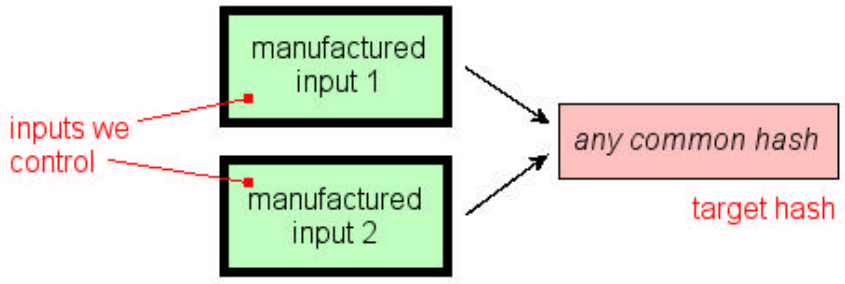


Fig. 9: Collision Resistance depicted

We manufacture both of the inputs in an attempt to coax the same hash value from each, and we don't care what the *particular* hash value generated is (just that they both match).

Exploting weak collision resistance: If we are able to create two inputs that generate the same hash, digital signature become suspect. In our example above, the document signed was "a promise to pay", and being able to substitute one signed document for another would certainly lead to havoc:

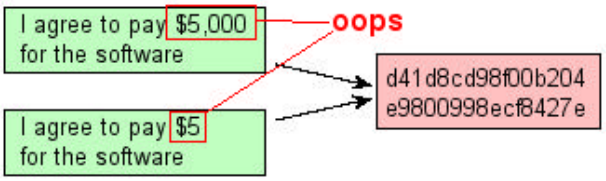


Fig. 10: Breaking digital signatures

As an aside, it's been recommended to always make a cosmetic change to any document you sign: if the person giving you the document secretly has a hash-equivalent pair, your small change - even a byte or two - will render the "other" file useless.

Preimage resistance

means that it's hard to find an input that generates a given hash value when you **don't** know the original input that made the target hash.

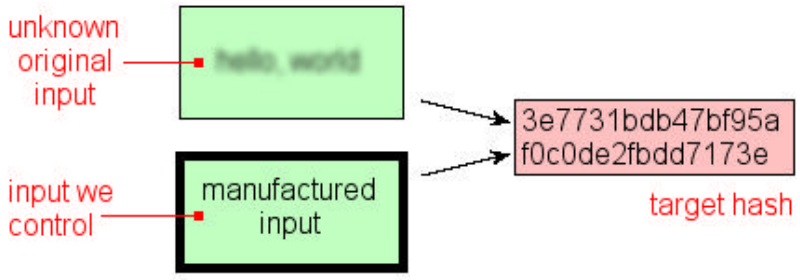


Fig. 11: Preimage Resistance depicted

Exploiting weak preimage resistance: If we are able to "work backwards" from a hash and create some text that produces the **same** hash, we can use this to beat hashed passwords. We won't ever know the **actual** input data that was used, but that doesn't matter. Looking at the flow for validating against hashed passwords, all that matters is that the **hashes** match, not the **passwords**, so if we can find *any* other text that produces the stored hash, we'll be granted access. "Collisions" mean "more than one password will be accepted".

- **Second preimage resistance**

means that it's hard to find an input that produces a given hash value when you **do** know the original input that generated the target hash.

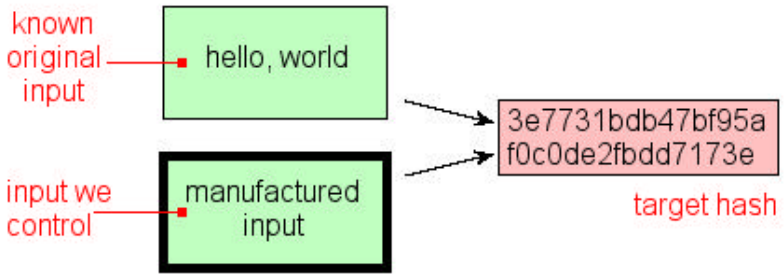


Fig. 12: Second Preimage Resistance depicted

This seems like a somewhat easier case of the previous item: the goal is to produce a new input that generates the given hash, but this time we know the original text that created it. We're not entirely clear just how much help this extra knowledge is. **Exploiting second weak preimage resistance**: As with preimage resistance, we want to fool somebody into authenticating *our* data as genuine, and we'd most likely use this when trying to introduce a corrupted software distribution.

Earlier we saw that ProFTPD (and many other organizations) publishes software and matching md5 checksums, and if we are able to maliciously modify the source code but nevertheless keep the same checksum, downloaders around the globe will accept our badware as genuine.

So what's the big news?

Some very bright researchers in China presented a paper, Collisions for Hash Functions MD4, MD5, HAVAL-128 and RIPEMD, at the Crypto 2004 conference in August 2004, and it's shaken up the security world considerably. This was some **outstanding** cryptography research.

They have found ways to reliably generate collisions in four hash functions much faster than brute-force time, and in one case (MD4, which is admittedly obsolete), *with a hand calculation*. This has been a stunning development.

These are all of the "we control both inputs" type - the first of our three kinds of collisions - and it holds the most promise in the compromise of digital signatures where the bad guy can create *two* contradictory documents and pull a switcheroo later.

We've heard there are already tools emerging that can reliably generate collision-pairs in "reasonable" amounts of CPU time, and with just a bit of searching found one example for MD5:

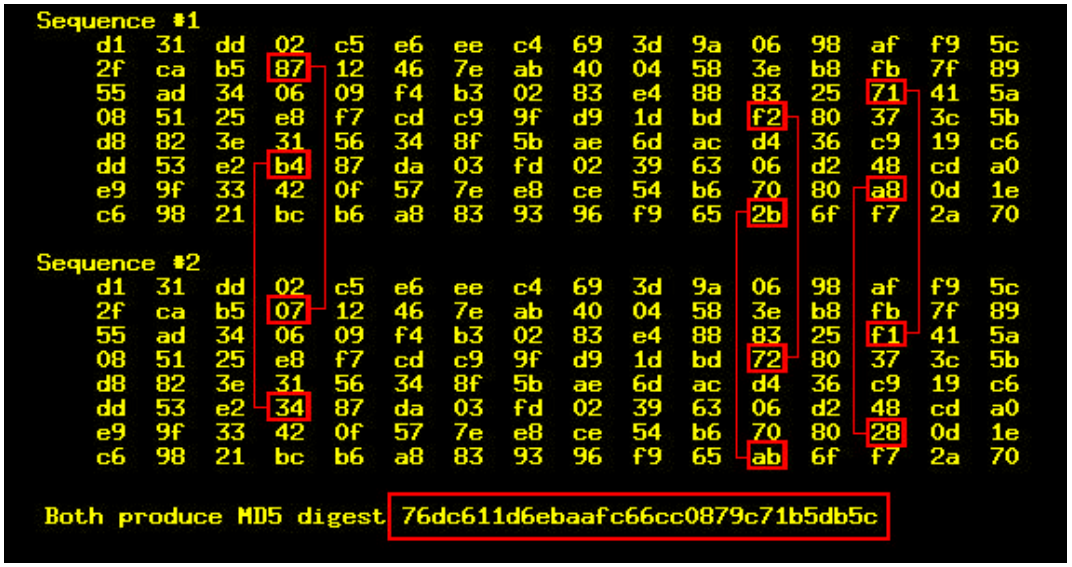


Fig. 13: a real MD5 collision

There is also a buzz about weaknesses in the SHA-0 and SHA-1 algorithms, but this is much more preliminary.

A researcher has managed to find a collision in SHA-0, which is not in such widespread use, and another has found a collision in "reduced SHA-1". A "reduced" algorithm typically eliminates some steps in the hashing process (40 rounds rather than 80), and even though finding a collision in what amounts to 1/2 of SHA-1 doesn't have any immediate impact, it's like a neon sign to the researchers saying "**look here**".

There is no evidence that preimage resistance or second preimage resistance are at risk in any of the algorithms (e.g., if you have an existing hash, find an input that produces that hash), but many of the cryptorati believe that blood is in the water and the sharks are on their way.

Once a weakness in an algorithm has been identified - even if it's theoretical or only on a reduced version - it's often the start of the whole thing unravelling.

What does this mean?

What does this mean?

In the short term, this will have only a limited impact on computer security. The bad guys can't suddenly start tampering with software that can fool published checksums, and they can't suddenly start cracking hashed passwords. Previously-signed digital signatures are just as secure as they were before, because one can't retroactively generate new documents to sign with your matched pair of inputs.

What it does mean, though, is that we've got to start migrating to better hash functions. Even though SHA-1 has long been thought to be secure, NIST (the National Institute of Standards and Technology) has standard for even longer hash functions which are named for the number of bits in their output: SHA-224, SHA-256, SHA-384, and SHA-512.

Five hundred twelve bits of hash holds 1.34×10^{154} possible values, which is far, far more than the number of hydrogen atoms in the universe [ref]. This is likely to be safe from brute-force attacks for quite a while.

Other Voices

As we conclude this Tech Tip, we remind the reader that **we are not crypto experts**, and have merely collected information from far and wide in a manner that we believe covers the topic with clarity and fidelity. But we could be wrong, and in the case of predicting the future, probably **are**.

Those wishing the authoritative word on this are encouraged to legendary security expert Bruce Schneier, who has written an analysis that appeared in Computerworld:

> **Opinion: Cryptanalysis of MD5 and SHA: Time for a new standard**
> Crypto researchers report weaknesses in common hash functions

As his was an opinion piece and not a Tech Tip, he hasn't covered the background, but his big-picture analysis are certainly much more likely to be correct than ours.

We also recommend Bruce's monumental work Applied Cryptography, which has been the gold standard of crypto texts for some time. We refer to this *constantly* when researching security issues and cannot recommend this book highly enough.

Professor Edward Felten wrote about this in his Freedom to Tinker weblog, and his is also an authoritative voice on the subject.

Feedback and/or corrections to this paper are gladly accepted.