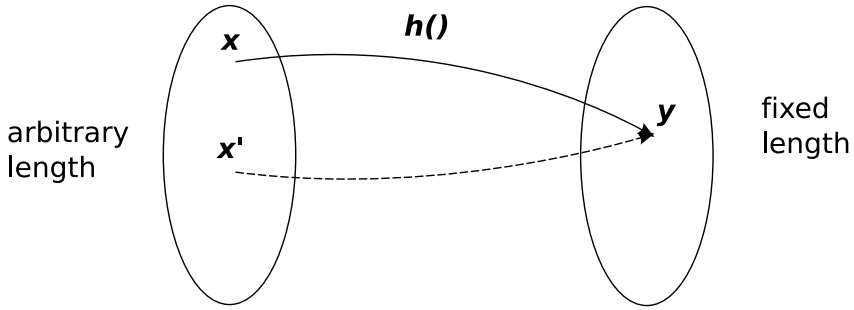
One Way Hash Functions

Simon Foley,
Department of Computer Science,
University College Cork

September 28, 2015

One Way Hash Functions



A function h maps arbitrary length value x to fixed length value y such that:

- Hard to reverse. Given value y not feasible to find x with $y = h(x)$.
- Collision freeness. Hard to find values x, x' such that $h(x) = h(x')$.
- Unpredictability. The hash value $h(x)$ does not give any information about any part of its operand x.

Examples

- ~~MD5~~ (RFC1321; 128 bit), ~~SHA1~~ (RFC3174; 160 bit), SHA256 (FIPS PUB 180-2 Secure Hash Standard, 256 bit), ...
- Last block of CBC-encryption (more computationally expensive).

Reversing a one-way hash function

It is feasible to reverse a one-way hash function if it maps to a relatively small hash-value.

Given a hash value v of size n bits then there is a high degree of probability that v will match the hash of one of 2^n random messages generated by an attacker.

- Given a 1-bit hash value v, attacker computes the hash of two randomly generated messages; high probability of a match with v.
- Given a 2-bit hash value v attacker computes the hash of four randomly generated messages; high probability of a match with v,
- etc.

Thus, in principle it requires 2^{128} hash calculations to reverse MD5 hash value; 2^{160} to reverse SHA1 hash value, etc., which is not feasible.

Recent (2009) design flaws have been found in MD5 which means that it only takes 2^{123} tests to reverse; this is still a 'theoretical' attack.

Hash functions MD5 and SHA1 are relatively effective at ensuring difficulty to reverse. However, they are not effective at ensuring collision freeness.

Protecting *nix passwords using one way hash

Store userids and *hashed* passwords in `/etc/passwd`; readable by all. Given a copy of file then easy for `logind` to check user's password, but not feasible for attacker to discover user's password.

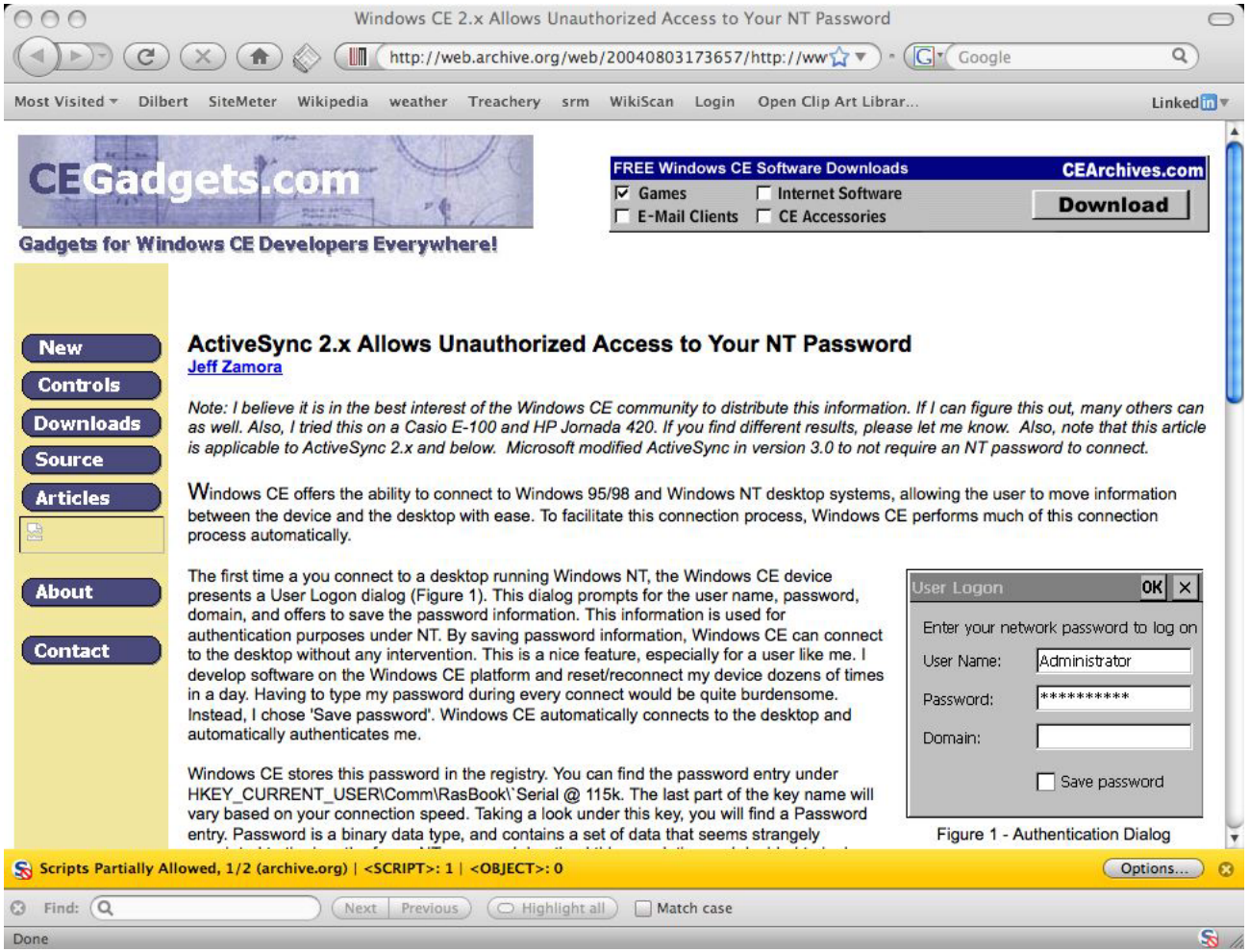
```
homer:6V7P6WEXFEi9Q:100:10:Homer Simpson:/homer:/bin/csh
monty:43fhehGGwGiGh:10:10:Montgomery Burns:/monty:/bin/csh
```

Early Unix used the crypt(3) one-way hash function implemented as DES encrypt of a blocks of nulls, with the password providing a key.

Later Unix (eg Linux) implemented crypt(3) as MD5 hash function, for example, the GNU C Library. In practice a range of possible hash functions are offered for password encryption. You can check: the printable version of md5 hash-value has prefix 1, while a 5/6 is used for SHA256/512.

In practice, the hashed passwords are not not stored in `/etc/passwd` but in the shadow password file `/etc/shadow`, which can only be accessed by root and provides an additional layer of security (belt and braces).

This is not how you Hash Passwords



Implemented the hash of $password$ as susagep $\oplus$ $password$.

Brute force attack on a hashed password

Consider unknown password p hashed using MD5 giving a 128-bit value $V = md5(p)$. An attacker has a copy of V.

The attacker generates random password values p' and tests $md5(p') = V$. The attacker has a good chance of a match after generating/testing 2^{128} passwords. This is not a feasible attack.

However, in practice, the attacker will limit his search to poorly chosen passwords. For example:

The attacker guesses that an unsophisticated user will pick a short password (7 characters) composed only of lowercase characters and numbers. He limits his (feasible) search to just these kinds of passwords

$$36^7 \text{permutations} \quad \approx \quad 7.8 \times 10^{10} \text{passwords}$$

Alternatively, the attacker guesses that an unsophisticated user will pick a word from a dictionary as their password. In this case he limits his (feasible) search to just dictionary words.

Precomputation dictionary Attack on *nix Password Files

Attacker wants to avoid having to brute force each password in password file; builds table of dictionary words and corresponding hash values:

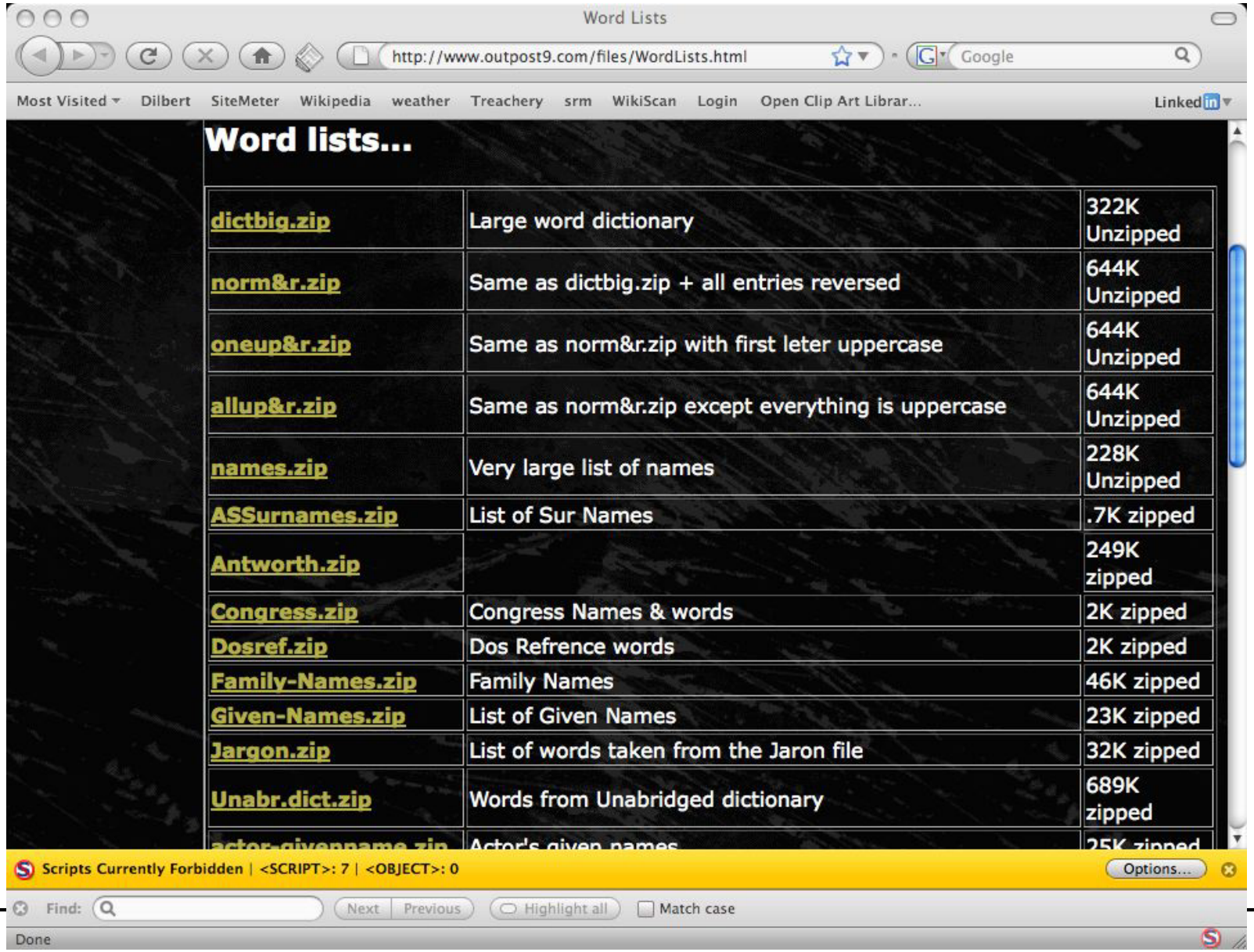
password	h(password)
aardvark	1ac23b37db0039dda62896bb21f312755
boy	$1$653805544e622bacc4cc028613a1358a
...	...

Attacker merges this table against `/etc/passwd` in hope of matching the hashes of poorly chosen passwords. The cost of storing and building a dictionary table is small and the attacker can use multiple dictionaries. Use dictionaries for different languages, Klingon, lines from songs, etc.

A user can also spot another user with the same password (as themselves).

This dictionary attack is an example of a *pre-computation attack*: most of the effort goes into building the dictionary, while the seach/merge is relatively cheap and the dictionary can be re-used. Technique can also be applied in a known-plaintext attack.

Some Dictionary Tables



Using Salt to defend against a Dictionary Attack

Strategy: make it impractical to build a pre-computation dictionary table.

When a password is chosen by the user, a random *salt* value s is generated and hashed with the password. The password file stores

$$s : h(s\hat{}password)$$

where $\hat{}$ denotes concatenation.
If the salt is large then building the dictionary table beomes costly.

$word$	salt	$h(salt\hat{}word)$
aardvark	0	$1$29b43ef4c7e4b84ff9f25ea158f46818
aardvark	1	$1$263818db1dc48169633a51e04fa0bf98
...	...	...
boy	0	1fc0f90e9b32b460b569c6d27291bc3ba
boy	1	1ef90e3e32b460b569c6d2723234aeba
...	...	...

Unix uses a 12 bit salt: 4096 possible hash values for each password. This means the attacker must put much more effort into building the table.

Some pre-computation dictionary attack tools: Crack, John the Ripper, L0phtCrack, Cain and Able.

Question

Suppose my password file uses a 128 bit salt along with the SHA1 (160 bits) hash function.

- ☐ Does this defend against a pre-computation dictionary attack?
- ☐ Does this defend against a brute-force attack?
- ☐ Does this defend against a password guessing attack that brute-forces all words from a dictionary?

Windows LAN Manager (LM) Hash Function (prior to WinNT)

Microsoft's 'home-made' one-way hash function.

- Turn password into 14-character string, either by truncating longer passwords or padding shorter passwords with nulls.
- Convert all lowercase characters to uppercase.
- Using each 7-char string as a DES key, encrypt a fixed constant with each key, yielding two 8-byte encrypted strings.
- Concatenate the two strings together to create 16-byte hash value.

Dictionary attacks are easy against this scheme.

- Most people pick easily guessable passwords.
- All characters converted to uppercase (reduces effective key space)
- No salt values used. Easy to build a dictionary of hash values.
- The two 7-char 'halves' of password are hashed independently. Brute force halves independently; Complexity of two halves same as complexity of one half. Easy to recognize passwords less than 7 characters (second half of key is all nulls).

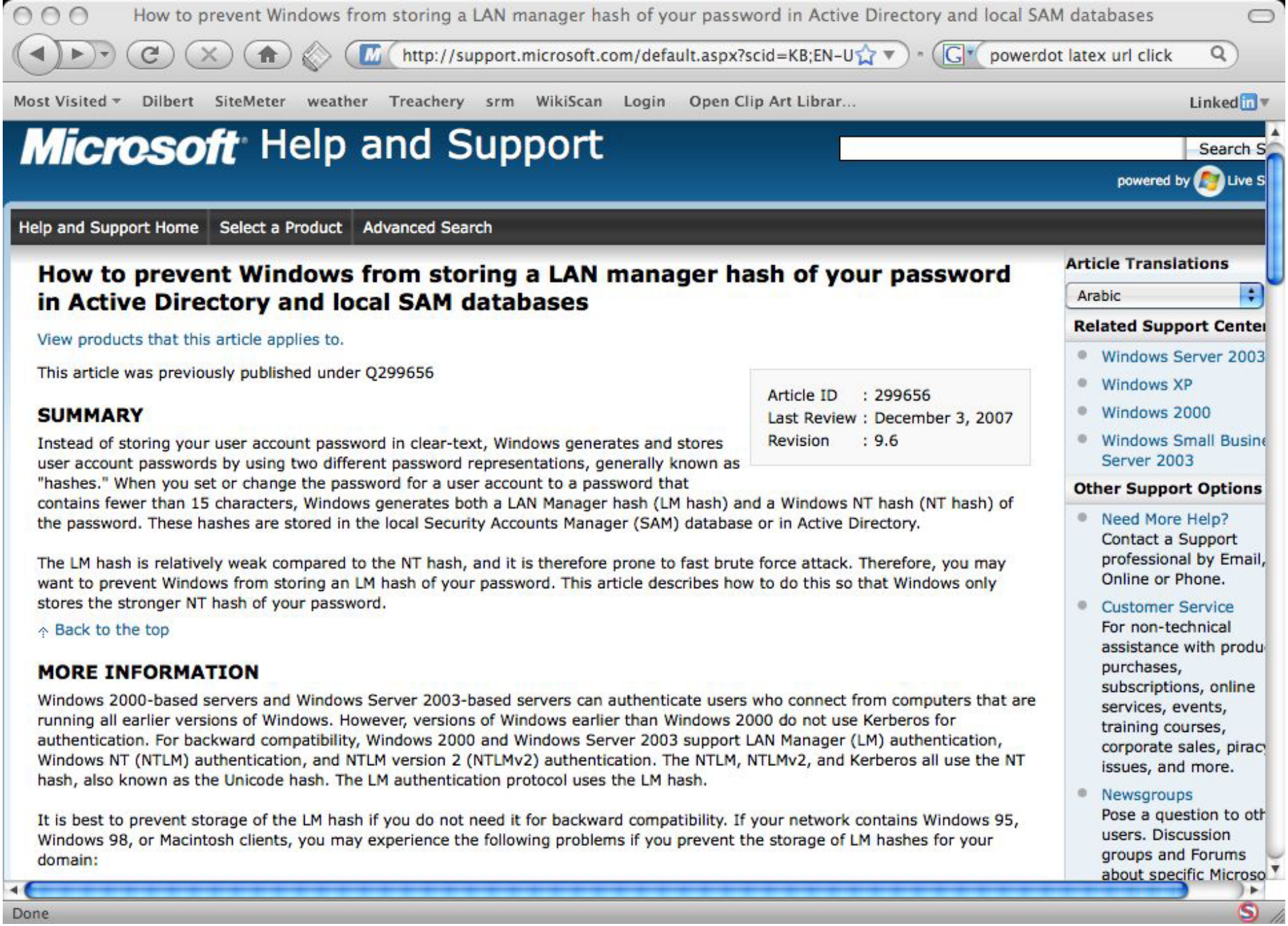
Windows NTLM Hash function

Used to authenticate users on NT systems.

- Password (upto 14 characters long) and case-sensitive converted to Unicode.
- Password hashed using MD4/MD5.

Attacks/Weaknesses:

- No Salt: can spot when two people have same password. Dictionary attack still possible.
- For backwards compatibility, both hashes (LM and NT) hashes may be used, even though the NT system uses only the NT hash to authenticate. Can attack the weaker LM hash value, and then test various lower-case alternatives to find the NT hash [Lophtcrack]. LM has can be disabled for XP and NT (enabled by default); With vista LM hash is disabled by default.


How to prevent Windows from storing a LAN manager hash of your password in Active Directory and local SAM databases
http://support.microsoft.com/default.aspx?scid=KB;EN-U
powerdot latex url click
Most Visited
Dilbert
SiteMeter
weather
Treachery
srm
WikiScan
Login
Open Clip Art Librar...
Linked
Microsoft Help and Support
Search S
powered by Live S
Help and Support Home
Select a Product
Advanced Search
How to prevent Windows from storing a LAN manager hash of your password in Active Directory and local SAM databases
View products that this article applies to.
This article was previously published under Q299656
Article ID : 299656
Last Review : December 3, 2007
Revision : 9.6
SUMMARY
Instead of storing your user account password in clear-text, Windows generates and stores user account passwords by using two different password representations, generally known as "hashes." When you set or change the password for a user account to a password that contains fewer than 15 characters, Windows generates both a LAN Manager hash (LM hash) and a Windows NT hash (NT hash) of the password. These hashes are stored in the local Security Accounts Manager (SAM) database or in Active Directory.
The LM hash is relatively weak compared to the NT hash, and it is therefore prone to fast brute force attack. Therefore, you may want to prevent Windows from storing an LM hash of your password. This article describes how to do this so that Windows only stores the stronger NT hash of your password.
Back to the top
MORE INFORMATION
Windows 2000-based servers and Windows Server 2003-based servers can authenticate users who connect from computers that are running all earlier versions of Windows. However, versions of Windows earlier than Windows 2000 do not use Kerberos for authentication. For backward compatibility, Windows 2000 and Windows Server 2003 support LAN Manager (LM) authentication, Windows NT (NTLM) authentication, and NTLM version 2 (NTLMv2) authentication. The NTLM, NTLMv2, and Kerberos all use the NT hash, also known as the Unicode hash. The LM authentication protocol uses the LM hash.
It is best to prevent storage of the LM hash if you do not need it for backward compatibility. If your network contains Windows 95, Windows 98, or Macintosh clients, you may experience the following problems if you prevent the storage of LM hashes for your domain:
Article Translations
Arabic
Related Support Center
Windows Server 2003
Windows XP
Windows 2000
Windows Small Busine Server 2003
Other Support Options
Need More Help? Contact a Support professional by Email, Online or Phone.
Customer Service For non-technical assistance with produ purchases, subscriptions, online services, events, training courses, corporate sales, piracy issues, and more.
Newsgroups Pose a question to oth users. Discussion groups and Forums about specific Microso
Done

Lightbeam shines a light on which web…

NSA whistleblower Edward Snowden s…

Anatomy of a password disaster - Adobe's giant-sized cryptographic blunder

by Paul Ducklin on November 4, 2013 | 61 Comments

FILED UNDER: Adobe, Cryptography, Data loss, Featured, Privacy

One month ago today, we wrote about Adobe's giant data breach.

As far as anyone knew, including Adobe, it affected about 3,000,000 customer records, which made it sound pretty bad right from the start.

But worse was to come, as recent updates to the story bumped the number of affected customers to a whopping 38,000,000.

We took Adobe to task for a lack of clarity in its

Related Articles

Adobe customer data breached - login and credit card data probably stolen, all passwords reset

Facebook locks users in a closet for using same passwords/emails on Adobe

Question

Suppose that each password p is stored in encrypted form in a password file as $E_{des}^{ecb}(K_A, p)$ using triple DES (ECB mode) where K_A is a secret master key. What are the vulnerabilities?

Question

Suppose that each password p is stored in encrypted form in a password file as $E_{des}^{ecb}(K_A, p)$ using triple DES (ECB mode) where K_A is a secret master key. What are the vulnerabilities?

The entire scheme relies on the security of a stored stored secret (master key): if an attacker discovers the key then every password is compromised.

Each (8-byte) plaintext block from a password is separately encrypted (result of DES-ECB). Therefore, an attacker will be able to distinguish short from long passwords by looking at the ciphertext: an 7-character password will be stored as 8-bytes of ciphertext, while a 12 character password will be stored as 16 bytes of cyphertext. The attacker might target short passwords for password-guessing attacks.

There is no randomness in the password encryption: two users with the same password will have the same encrypted password entry. Thus an attacker with an account could look for (encrypted) passwords in the file that match his own (encrypted) password. Its likely that same password that occurs multiple times in the file will be a weak password and subject to a password guessing attack.

http://arstechnica.com/tech-policy/2014/06/poorly-anonymized-logs-reveal-nyc-cab-drivers-detailed-whereabouts/

Poorly anonymized logs reveal NYC cab drivers' detailed whereabouts

Botched attempt to scrub data reveals driver details for 173 million taxi trips.

by Dan Goodin - Jun 23, 2014 6:25 pm UTC

David R. Tribble

In the latest gaffe to demonstrate the privacy perils of anonymized data, New York City officials have inadvertently revealed the detailed comings and goings of individual taxi drivers over more than 173 million trips.

City officials released the data in response to a public records request and specifically obscured the drivers' hack license numbers and medallion numbers. Rather than including those numbers in plaintext, the 20 gigabyte file contained one-way cryptographic hashes using the MD5 algorithm. Instead of a record showing medallion number 9Y99 or hack number 5296319, for example, those numbers were converted to 71b9c3f3ee5efb81ca05e9b90c91c88f and 98c2b1aeb8d40ff826c6f1580a600853, respectively. Because they're one-way hashes, they can't be mathematically converted back into their original values. Presumably, officials used the hashes to preserve the privacy of individual drivers since the records provide a detailed view of their locations and work performance over an extended period of time.

LATEST FEATURE STORY

FEATURE STORY (2 PAGES)

50 years of Moog, the analog synth that still beats 1s and 0s

A look back at Bob Moog's industry-altering invention.

WATCH ARS VIDEO

Space Shuttle Enterprise Tour

A mini-documentary on one of NASA's experimental toys, the Enterprise.

STAY IN THE KNOW WITH

LATEST NEWS

MARS NEEDS MAVEN

UPDATED: Maven successfully enters Mars orbit

MOST IMPORTANTLY, DON'T LIE

Kickstarter lays down new rules for when

Pre-computation Attacks and Space-Time Tradeoff

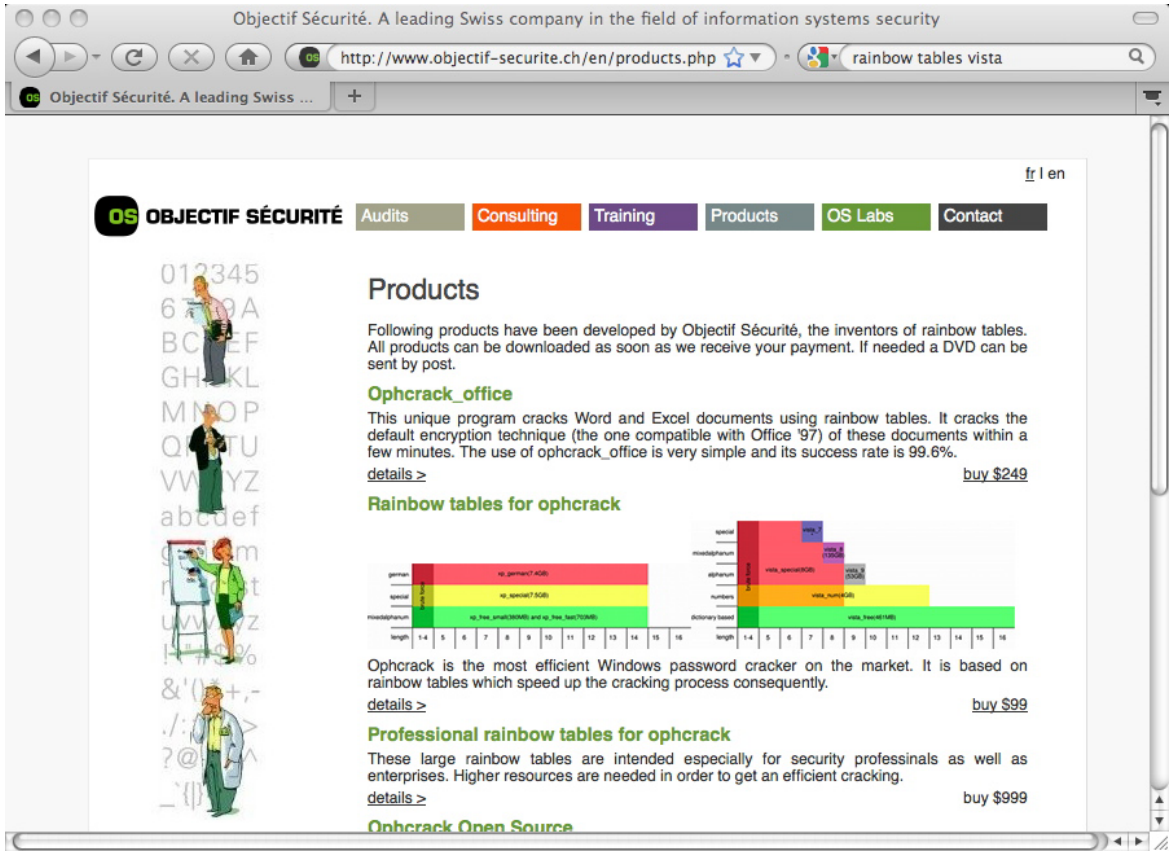
Memory is now relatively cheap and it has become feasible to build pre-computation tables that include not just dictionary words, but also classes of password permutations. Building these tables can be (time) costly, but once built, searching/merging for a hash value becomes cheap.

Storing a table that contained every possible password/hash pair would not be feasible in practice (though lookup is instant). A rainbow table is a special data-structure that provides a more compact representation of passwords/hashes but with a computational cost for lookup (dependent on space allocated for table).

For example, the tool Ophcrack uses rainbow tables to find Windows passwords.

- Charset [0..9][a..z][A..Z], size 380MB, for LMHash on XP, free.
- Charset [0..9][a..z][A..Z], size 380MB, for LMHash on XP, free.
- Dictionaries, limited permutations, size 461MB for NTHASH, free.
- Nearly everything on keyboard, size 8.0GB for NTHASH, $99

Rainbow Tables for Vista



Vista (52.0GB): Success rate: 99%
Passwords of length 8: Charset: 0123456789abcdefghijklmnopqrstuvwxyz with the first letter capitalized
Passwords of length 9 Charset: 0123456789abcdefghijklmnopqrstuvwxyz

Cracking Stealth MXP Memory Stick Protection

In order to support a no password re-use policy, USB Stick stored history of past passwords, each hashed (unsalted) with SHA1.

OWASP Password Storage Cheat Sheet

Some useful guidance on storing passwords:
`https://www.owasp.org/index.php/Password_Storage_Cheat_Sheet`

Collisions and One Way hash functions

Recall the requirement

- Hard to reverse. Given y not feasible to find x such that $y = h(x)$.

Intuitively, given a specific n-bit hash value y (eg a hashed password) then I would expect to have to generate and test up to 2^n random messages x before having a *good chance* of finding $y = h(x)$.

Recall the requirement

- Collision freeness. Hard to find values x, x' such that $h(x) = h(x')$.

(We'll see later that this is very important for digital signatures.)

It turns out that we should not consider it taking up to 2^n messages/tests to find an x, y with the same hash value. In practice, we have good chance of finding a pair within $2^{n/2}$ due to the Birthday Paradox.

For example, we have good chance of finding a pair of messages m and m' such that $\texttt{md5}(m) = \texttt{md5}(m')$ with just 2^{64} tests, making it vulnerable to brute-force attack. Note however, MD5 suffers more serious weakness and collisions can be found within seconds running on a fast desktop.

Birthday Pardox

With 23 people in room there's more than 50% chance 2 share same birthday.

- Probability first two people picked have different birthday: $(1 - 1/365)$.
- Probability that the third person selected has different birthday from first two (given first two have different birthdays) is $(1 - 2/365)$.
- ...

Thus, the probability that first k people have different birthday is

$$(1 - 1/365) \times (1 - 2/365) \times \cdots (1 - (k-1)/365) = \frac{365!}{k! \times 365^k}$$

This is less than 0.5 if $k > \sqrt{365} \approx 23$.

Birthday Attack on a One-Way Hash Function

We want to find any two messages m, m' such that $h(m) = h(m')$.
Intuition: looking for two different messages with hash collision in 2^n possible hash values *versus* looking for two different people with birthday collision in 365 possible days.

Probability of no match after k tests is

$$(1 - \frac{1}{2^n}) \times (1 - \frac{2}{2^n}) \times \cdots \times (1 - \frac{k-1}{2^n}) = \frac{(2^n)!}{k! \times 2^{nk}}$$

which is less than 0.5 when $k \approx \sqrt{2^n}$ $(= 2^{n/2})$.

Thus, if 2^n search is considered computationally sufficiently infeasible, then the output of a of a collision-resistant hash function needs to be at least $2n$ bits large if collision search (requiring $2^{2n/2}$ operations) is to be infeasible.
In principle:

- 2^{64} MD5 tests: more than 50% chance of finding collision.
- 2^{80} SHA1 tests: more than 50% chance of finding collision.
- ...

Recent Results on finding Collisions

Collisions have been announced in many existing hash functions, including

- MD4 [1996-complexity 2^{22}; 2005-complexity 2^{8}].
- MD5 [1993-complexity 2^{16}]
- SHA0 [2005-complexity 2^{39}]
- SHA1[2005-complexity 2^{63}]

Currently, SHA-2 (SHA256 and SHA512) are considered safe. However SHA1 is still used and even MD5, but to a lesser extent. We will return to this problem again when we study digital signatures (which rely on one-way hash functions).

NIST(US) launched [2008] a competition for new hash algorithms.

The winner was announced in 2012 and the standard for SHA3 is forthcoming; see `http://csrc.nist.gov/groups/ST/hash/index.html`

However, there is some debate as to whether NIST should be relied upon: L. Hay Newman, *Can You Trust NIST?*, IEEE Spectrum, October 2013. See `https://konklone.com/post/why-google-is-hurrying-the-web-to-kill-sha-1` for SHA-1 assessment.

Postscript files with MD5 hash value a25f7f0b 29ee0b39 68c86073 8533a4b9

Julius. Caesar
Via Appia 1
Rome, The Roman Empire

May, 22, 2005

To Whom it May Concern:

Alice Falbala fulfilled all the requirements of the Roman Empire intern position. She was excellent at translating roman into her gaul native language, learned very rapidly, and worked with considerable independence and confidence.

Her basic work habits such as punctuality, interpersonal deportment, communication skills, and completing assigned and self-determined goals were all excellent.

I recommend Alice for challenging positions in which creativity, reliability, and language skills are required.

I highly recommend hiring her. If you'd like to discuss her attributes in more detail, please don't hesitate to contact me.

Sincerely,

Julius Caesar

Julius. Caesar
Via Appia 1
Rome, The Roman Empire

May, 22, 2005

Order:

Alice Falbala is given full access to all confidential and secret information about GAUL.

Sincerely,

Julius Caesar

[from: Lucks and Daum, *The Story of Alice and her Boss: Hash Functions and the Blind Passenger Attack*, Eurocrypt 2005 rump session.]

Such recommendations change over time. NIST is a good place to check for the most recent best practices. *Recommendation for Key Management Part 1: General* is a good place to start.

Providing Secrecy and Integrity

Recall that it is not easy to provide secrecy and integrity in a single cryptographic pass of the message.

Use a one-way hash function to provide a cryptographic checksum of the message.

Alice and Bob share a secret key KAB. Alice sends message M to Bob:

$$Alice \rightarrow Bob : E(K_{AB}, M \hat{} h(M))$$

Bob decrypts the message, recalculates $h(M)$ and checks it against hash provided. If the hash value is different then Bob knows the message has been corrupted.

Note that we often use $\{M\}_{K_{AB}}$ as an abbreviation of $E(K_{AB}, M \hat{} h(M))$

We often refer to a hash function as a message digest.

Keyed One Way Hash Functions

A keyed hash function $h_K(M)$ provides a hash-based implementation of a message authentication code.

For example, Alice and Bob share a secret key KAB. Alice wants to ensure integrity and sends message M to Bob

$$Alice \rightarrow Bob : M, h_{K_{AB}}(M)$$

Bob recalculates $h_{K_{AB}}(M)$ and checks it against hash provided.

The standard HMAC provides keyed hash calculations for MD5, SHA, etc. Given hash function $h(M)$, it is approximately implemented as

$$h_K(M) = h(K \hat{} h(K \hat{} M))$$

Which is not unreasonable given our requirement that

- Unpredictability. The hash value $h(x)$ does not give any information about any part of its operand x.

See RFC 2104 for full details.