Algorithm: **MCS_PHS**

Principal submitter: **Mikhail Maslennikov**

Revision: February 24, 2014

INITIAL SECURITY ANALYSIS MCS_PHS

In this document password hashing scheme MCS_PHS [1] will be analyzed for resistance to the following attacks:

1) attacks on hashing algorithm MCSSHA-8;
2) dictionary attack and brute force attack;
3) other security measures.

Attacks on hashing algorithm MCSSHA-8.

MCSSHA-8 hash algorithm was developed based on the requirements [4], but it use nonlinear feedback shift register (NFSR) with byte elements, so it's possible to calculate hash with any length in bytes, not only 28, 32, 48 and 64 bytes as in NIST requirements. In MCSSHA-8 hash length in byte can be any value from 4 to 64 and any hash values for same messages should be different as random values.

This algorithm is a continuation of a series of hash algorithms MCSSHA family (MCSSHA 3 – 6). During the SHA-3 competition independent experts Jean-Philippe Aumasson and María Naya-Plasencia have found some inconsistencies algorithms MCSSHA 3 - 6 requirements NIST [5].

MCSSHA-8, like previous versions of MCSSHA has algorithms, consist from three stages: preprocessing, pre-hash and final hash computation. Preprocessing and pre-hash computation for MCSSHA-8 are same, like MCSSHA-5 and 6. But final hash computation is different from all previous versions [2].

In final hash computation for MCSSHA-8 we build two hash values length N from state of NFSR length 2N. Final hash is bitwise XOR of this two hashes. In this case described in [5] methods become ineffective.

Dictionary attack and brute force attack

As noted in [6], "the simplest way to crack a hash is to try to guess the password, hashing each guess, and checking if the guess's hash equals the hash being cracked. If the hashes are equal, the guess is the password. The two most common ways of guessing passwords are **dictionary attacks** and **brute-force attacks**. A dictionary attack uses a file containing words, phrases, common passwords, and other strings that are likely to be used as a password. Each word in the file is hashed, and its hash is compared to the password hash. If they match, that word is the password. These dictionary files are constructed by extracting words from large bodies of text, and even from real databases of passwords. Further processing is often applied to dictionary files, such as replacing words with their "leet speak" equivalents ("hello" becomes "h3110"), to make them more effective.

A brute-force attack tries every possible combination of characters up to a given length. These attacks are very computationally expensive, and are usually the least efficient in terms of hashes cracked per processor time, but they will always eventually find the password. Passwords should be long enough that searching through all possible character strings to find it will take too long to be worthwhile.
There is no way to prevent dictionary attacks or brute force attacks. They can be made less effective, but there isn't a way to prevent them altogether. If your password hashing system is secure, the only way to crack the hashes will be to run a dictionary or brute-force attack on each hash."

So it’s impossible to prevent dictionary or brute force attack. In MCS_PHS we can find some properties that allow to increase complication of using these methods.
Using many different hash functions. In MCS_PHS scheme: first step – calculating from password and salt hash length 64 bytes and then step by step decrease hash length using MCSSHA-8 length N – 1 for hash length N. For each N hash algorithm MCSSHA-8 with hash length N will be marked as MCSSHA8_N and hash for message M – MCSSHA8_N(M). For any message M and different N1 and N2 the values MCSSHA8_N1(M) and MCSSHA8_N2(M) are random and practically independent. So, if we want to use table of hash values, we need tables for MCSSHA8_N for each N or one table for all MCSSHA-8 iteration from 64 to 32, whose construction is more complex.

Other security measures

1. Security properties expected from MCS_PHS.

- random looking output;
- one-way;
- collision resistant;
- immune to length extension.

Random looking output. MCS_PHS scheme used only MCSSHA-8 hash algorithm that use NFSR transformation for byte sequence. In feedback function of NFSR present substitution π, which properties allow to obtain the output values do not differ from random and equiprobable – see 10.1.1 of [7].

One-way and collision resistance. Because NSFR worked with delays (see [2]), so there are very complicated non-linear equations for hash values. It’s practically impossible to find any another method to decide this eqiations exept brute force.

Immune to length extension. All algorithms from MCSSHA family doesn’t use padding. Password length and salt length present in block for first hash MCSSHA8_64. If we change in MCS_PHS some length or some bytes in password or salt then final result of MCSSHA8_64 hash will be different as random and equiprobable.

Possible attack on MCS_PHS

This kind of attack was described in 5.1 [8].

The attack on PBKDF1 is based on an obvious relation between keys derived using the *same* salt. For any salt *s* and two iteration counts $c_0 < c_1$, let $y_i = F(p, s, c_i) = H^{(c_i)}(p\_s)$. Then, it is easy to see that $y1 = H(c1-c0)(y0)$. This relation allows an attacker to distinguish *y*0 from a random function with one *F* query (*s, c*1) and (*c*1 − *c*0) *H* queries. Note that if the key $y0 = H(c0)(p\_s)$ were ever compromised for some reason, then any key derived using the same salt *s* and an iteration count *larger than* *c*0 would all be compromised. This might happen in practice if the user (or the security administrator of the system) decides to increment the iteration count. Therefore, it is a good practice in general to use different salt values in deriving different keys.

To protect against this attack in MCS_PHS in final stage we use *pre-derived key*, and then calculate two hashes from this key for preparing final derived key. So if final derived key will be compromised, attacker couldn’t restore pre-derived key using only final derived key and couldn’t continue hashing cycle.

EFFICIENCY ANALYSIS MCS_PHS

In the table 1 compare speeds for PBKDF2 with SHA-1 and MCS_PHS. Test conditions: Intel Core i7 3537U CPU, @ 2,00 GHz, 2,50 GHz.
PBKDF2 was build using OpenSSL public codes.

Table 1. Comparing speed for PBKDF2 with SHA-1 and MCS_PHS

Algorithm	Derived key length	Cycles	Time (sec.)
PBKDF2	20	4096	0,047
MCS_PHS	32	4096	0,055
PBKDF2	20	10000	0,114
MCS_PHS	32	10000	0,132
PBKDF2	20	1000000	10,549
MCS_PHS	32	1000000	12,095
PBKDF2	20	16777216 (extremely long from RFC 6070)	184,252
MCS_PHS	32	16777216	213,078
MCS_PHS	64	16777216	400,998

In the table 2 compare speeds of MCS_PHS with different parameters.

Table 2. Speed MCS_PHS with different parameters.

Initial memory (byte) (m_cost)	Derived key length	Cycles (t_cost)	Time (sec.)
256	32	0	0,001
256	32	1000	0,014
256	64	0	0,000
256	64	1000	0,026
32	32	0	0,001
32	32	1000	0,015
32	64	0	0,000
32	64	1000	0,029

Required memory

Required memory consist from memory for MCSSHA-8 hash algorithm and memory for MCS_PHS scheme. As noted in [2], required memory for MCSSHA-8 hash algorithm not above 0,5 KB. Memory for MCS_PHS scheme is not above 0,5 KB too. So, total memory is not above 1 KB.

Conclusion

For MCS_PHS scheme applicated only common methods like brute force and dictionary attack. Costs of testing can be adjusted by the parameters MCS_PHS. Efficiently of MCS_PHS comparable with PBKDF2 with SHA-1.

References

1. Mikhail Maslennikov. Password hashing scheme MCS_PHS. http://crypto.systema.ru/mcssha/MCS_PHS (eng).pdf

2. Mikhail Maslennikov. Secure hash algorithm MCSSHA-8. http://crypto.systema.ru/mcssha/MCSSHA-8 (eng).pdf

3. PKCS #5: Password-Based Cryptography Specification. Version 2.0. RFC 2898.

4. Announcing Request for Candidate Algorithm Nominations for a New Cryptographic Hash Algorithm (SHA–3) Family. NIST, Docket No.: 070911510–7512–01.
http://csrc.nist.gov/groups/ST/hash/documents/FR_Notice_Nov07.pdf

5. Jean-Philippe Aumasson and Mar´ıa Naya-Plasencia. Cryptanalysis of the MCSSHA Hash Functions.
https://131002.net/data/papers/AN09.pdf

6. https://crackstation.net/hashing-security.htm

7. Mikhail Maslennikov. Secure hash algorithm MCSSHA-3.
http://crypto.systema.ru/mcssha/MCSSHA-3.pdf

8. Frances F. Yao and Yiqun Lisa Yin. Design and Analysis of Password-Based Key Derivation Functions.
http://palms.ee.princeton.edu/PALMSopen/yao05design.pdf