Algorithm: **MCS_PHS**

Principal submitter: **Mikhail Maslennikov**

Revision: February 16, 2014

PASSWORD HASHING SCHEME MCS_PHS

Table of Contents

1. INTRODUCTION

In many applications of public-key cryptography, as described in [2], user security is ultimately dependent on one or more secret text values or passwords. Since a password is not directly applicable as a key to any conventional cryptosystem, however, some processing of the password is required to perform cryptographic operations with it. Moreover, as passwords are often chosen from a relatively small space, special care is required in that processing to defend against search attacks.

A general approach to password-based cryptography, is to combine a password with a salt to produce a key. The salt can be viewed as an index into a large set of keys derived from the password, and need not be kept secret. Although it may be possible for an opponent to construct a table of possible passwords (a so-called "dictionary attack"), constructing a table of possible keys will be difficult, since there will be many possible keys for each password. An opponent will thus be limited to searching through passwords separately for each salt.

This document specifies password hashing scheme MCS_PHS. This scheme based on secure hash algorithm MCSSHA-8. This algorithms is iterative, one-way hash functions that can process a message to produce a condensed representation called a *message digest*. Description of MCSSHA-8 present in [1]. This algorithm enable the determination of a message’s integrity: any change to the message will, with a very high probability, result in a different message digest. This property is useful for MCS_PHS password hashing scheme.

MCS_PHS can be used by several protocols to derive encryption keys from a password.

2. NOTATION

c	iteration count, a positive integer
DK	derived key, a byte sequence
dkLen	length in bytes of derived key, a positive integer
Hash_k	underlying hash function with output length k (in bytes).
KDF	key derivation function
P	password, a byte sequence
dpLen	password length in bytes
S	salt, a byte sequence
dsLen	salt length in bytes
dCostLen	additional memory length, a positive integer

3. SALT

As described in [2], salt in password-based cryptography has traditionally served the purpose of producing a large set of keys corresponding to a given password, among which one is selected at random according to the salt. An individual key in the set is selected by applying a key derivation function KDF, as

DK = KDF (P, S)

where DK is the derived key, P is the password, and S is the salt.

This has two benefits:

1. It is difficult for an opponent to precompute all the keys corresponding to a dictionary of passwords, or even the most likely keys. If the salt is 128 bits long, for instance, there will be as many as 2^128 keys for each password. An opponent is thus limited to searching for passwords after a password-based operation has been performed and the salt is known.

2. It is unlikely that the same key will be selected twice. Again, if the salt is 128 bits long, the chance of "collision" between keys does not become significant until about 2^64 keys have been produced, according to the Birthday Paradox. This addresses some of the concerns about interactions between multiple uses of the same key, which may apply for some encryption and authentication techniques.

4. Iteration count

An iteration count has traditionally served the purpose of increasing the cost of producing keys from a password, thereby also increasing the difficulty of attack.

5. Key derivation function (KDF)

A key derivation function produces a derived key from a base key and other parameters. In a password-based key derivation function (PBKDF), the base key is a password and the other parameters are a salt value and an iteration count.

MCS_PHS use PBKDF_MCS key derivation function.

PBKDF_MCS uses to derive keys a hash function that can calculate different hash value for any hash length before 32 and 64 bytes.

PBKDF_MCS (P, dpLen, S, dsLen, c, dCostLen, dkLen)

Options: Hash_k underlying hash function with hLen = k

Input:	P	password, a byte sequence
	dpLen	password length in bytes
	S	salt, a byte sequence
	dsLen	salt length in bytes
	c	iteration count, not negative integer
	dCostLen	additional memory, a positive integer
	dkLen	intended length in bytes of derived key, a positive integer

Output: DK derived key, a dkLen-byte sequence

Steps:

1. if(dkLen > 64) output "derived key too long" and stop.
2. if(dpLen + dsLen + 2 > dCostLen) output "password or salt too long" and stop.
3. Prepare initial sequence T_0 = (dpLen || P || dsLen || S || < i, i+1,..., dCostLen – 1 >), where i = dpLen + dsLen + 2.
4. Perform 64 – dkLen + 1 iteration:

T_1 = Hash_64(T_0),
T_2 = Hash_63(T_1),

...

T_{64 – dkLen + 1} = Hash_{dkLen}(T_{64 – dkLen})

5. Apply the underlying hash function Hash_{dkLen} for c iterations to the T_{64 –(dkLen + 1)} to produce a pre-derived key pDK. If c = 0 this stage absent and pDK = T_{64 – dkLen + 1}.

   U_1 = Hash_{dkLen}(T_{64 –(dkLen + 1)}),
   U_2 = Hash_{dkLen}(U_1),
   U_3 = Hash_{dkLen}(U_2),

   ...
   U_c = Hash_{dkLen}(U_{c-1}),
   pDK = U_c

6. Produce derived key DK from pre-derived key pDK.

   If dkLen != 64, then

   Tmp = Hash_64(pDK),
   DK = Hash_{dkLen}(Tmp)

   If dkLen = 64, then

   Tmp = pDK + <0,1,...,63>
   DK = Hash_{dkLen}(Tmp)

   where '+' is add operation in Z/256.

6. Control examples. Hash – MCSSHA-8.

Password: "qwerty12345"
Salt: "AE 0F 55 70 C2 BD 3E 90 24 CA 76 7F B8 6D 10 87"
c: 0
dCostLen: 256
dkLen: 32

Step by step:

1. Block for Hash_64:

```
0b 71 77 65 72 74 79 31 32 33 34 35 10 ae 0f 55 70 c2 bd 3e 90 24 ca 76 7f b8 6d 10 87 1d
1e 1f 20 21 22 23 24 25 26 27 28 29 2a 2b 2c 2d 2e 2f 30 31 32 33 34 35 36 37 38 39 3a 3b
3c 3d 3e 3f 40 41 42 43 44 45 46 47 48 49 4a 4b 4c 4d 4e 4f 50 51 52 53 54 55 56 57 58 59
5a 5b 5c 5d 5e 5f 60 61 62 63 64 65 66 67 68 69 6a 6b 6c 6d 6e 6f 70 71 72 73 74 75 76 77
78 79 7a 7b 7c 7d 7e 7f 80 81 82 83 84 85 86 87 88 89 8a 8b 8c 8d 8e 8f 90 91 92 93 94 95
96 97 98 99 9a 9b 9c 9d 9e 9f a0 a1 a2 a3 a4 a5 a6 a7 a8 a9 aa ab ac ad ae af b0 b1 b2 b3
b4 b5 b6 b7 b8 b9 ba bb bc bd be bf c0 c1 c2 c3 c4 c5 c6 c7 c8 c9 ca cb cc cd ce cf d0 d1
d2 d3 d4 d5 d6 d7 d8 d9 da db dc dd de df e0 e1 e2 e3 e4 e5 e6 e7 e8 e9 ea eb ec ed ee ef
f0 f1 f2 f3 f4 f5 f6 f7 f8 f9 fa fb fc fd fe ff
```

2. Hash_64 result:

```
77 5e be 55 57 97 02 b8 0d fd cd 23 08 33 24 f1 04 aa 61 2c 2f aa 8a aa 28 0f fb 5e fe 66
32 69 f6 71 f7 0d c5 e4 da 4b 88 c6 db 5a 4d c4 61 b1 f8 d1 ec c4 d0 e7 ef 11 65 8c e4 47
04 c2 f4 26
```

3. Hash_63 result:

3b 83 f1 e2 8e 17 3b b8 2d 3b 40 b8 cb 7c 23 3f 6d 96 34 67 93 ab b2 86 ef 57 59 23 3a 94 e5 fe c6 c6 7d e8 62 a3 74 38 42 e2 f4 3f 4a dd 07 83 28 22 9a cd 30 a8 0a 55 86 fc 23 f1 07 db 43

4. Hash_62 result:

9d b8 27 63 33 68 c5 60 cd cb 6a 41 14 3f 6e c7 cb ae 1d f2 be f6 07 bf fe 62 30 c3 b5 10 65 ee 5d 68 23 8b 5c 08 0b 15 1a 51 03 bb fe e3 21 ff 76 5f af 40 c9 8f b2 f7 f6 7e 63 a3 5d e5

5. Hash_61 result:

3c b9 33 8f f1 89 14 40 89 5e 80 46 af 69 cf f7 53 35 80 d4 85 f4 e5 f5 fe 28 10 75 db 65 92 d0 97 63 b7 9c a8 90 17 c8 aa 7a d4 19 53 e8 c3 a2 e7 21 7f 0c 0d ce 83 4c a5 ab e2 60 a8

6. Hash_60 result:

fa b6 4e 37 ae a7 9c c6 7e f8 54 60 c5 af a6 c8 be e9 d1 9c d4 37 cf d4 dd d3 99 0e e8 fa cb 36 d4 89 bb dc 50 51 55 da 0e 3d b4 51 90 7d 26 c4 65 5d be 20 09 7e d6 c1 96 05 f3 b3

7. Hash_59 result:

8c eb 8c 5b d3 51 f1 4a e0 23 32 4d 34 82 fd f9 0c 4f 3b 91 db 87 a3 c4 7b 81 37 86 be 65 2d 02 16 00 53 ca 3c b1 62 51 b5 b5 77 98 43 3b 5f d0 bc b4 3e a1 5f 24 3e ca 62 a2 b1

8. Hash_58 result:

3d ef ef 1b e9 f8 7e 38 42 0f 09 4b 89 e3 fe af ca a9 eb 53 53 78 d6 9d 30 85 2c 35 35 df e2 24 2d 63 b7 9d bb fc bb 30 81 23 48 b0 c1 fd 87 15 fa 49 b4 93 e4 0d c0 d5 40 ed

9. Hash_57 result:

82 0c e1 25 a9 c7 f0 d4 04 31 b3 5c e1 94 4e 31 b4 5f 81 cd 00 b4 fc c1 5b d2 56 ed 01 49 24 d4 0c aa de 7c 76 01 68 c1 9b 81 45 3d 5e a2 3d 32 a2 8b 25 84 c3 f9 2a b2 62

10. Hash_56 result:

8c 80 db 51 56 23 71 bb 87 51 ae ca 9f ab 6f b7 7f a8 b1 69 77 d2 52 bb 69 89 7e 92 6a ab a5 56 ea 80 ca 45 29 ac a8 cd f2 28 08 90 b4 93 90 10 35 9a a2 51 ae 33 22 6e

11. Hash_55 result:

3f 48 9f 20 86 8a ba 07 1d 0c 5f de 6b d7 7c 87 e2 92 c2 0c 7a 40 27 aa fa 90 c5 99 b6 8c 38 10 80 5b 5e fd 91 54 3a f1 b8 a1 5d 72 06 a1 0b 55 d9 6e da a2 0a 03 73

12. Hash_54 result:

51 26 08 a0 54 a2 a8 04 af eb 52 c1 7b 06 0c f0 5d 3c ba 57 f7 54 41 89 77 06 c2 cc 95 a9 f1 f4 4a 3b 41 2e 5e ba fd 08 5a 36 35 d3 4b 10 b2 93 a4 f1 f9 96 a7 36

13. Hash_53 result:

d0 29 ea 96 f2 48 ab 4b ad aa e1 59 b4 1e d1 ee b9 e8 f5 c3 3c a9 33 3e e4 08 fc 09 7b 10 fc 9b 17 6c 00 d0 2f 3c c4 35 e8 46 f8 54 fa ef 8b 34 bc 5f e6 84 0a

14. Hash_52 result:

10 63 38 c9 bf f2 e9 41 91 f1 86 26 51 ef f1 9d a0 2b 16 d4 a4 4f 5f 80 bf 76 65 58 ae c0 5b e3 fa 80 f9 62 94 00 cd 84 f5 ac f1 d6 4d 3d 28 c6 8d 2f aa 38

15. Hash_51 result:

d6 ee 65 d6 21 6c 16 5f 4a b1 33 65 04 d7 9f 25 41 8c 75 d6 b8 f7 87 fa 9a eb 0f 83 95 8a 1b 67 5f 0a 3c 99 6c 85 4f ed 90 cb 19 fa 10 f1 fd 56 fd 4e c5

16. Hash_50 result:

a2 67 33 d7 ba 43 37 46 36 d5 02 90 17 4d 9e cc e1 18 f2 d7 80 56 d4 88 0c 65 06 a9 a5 4d 6c 6e 6a fb 88 5f a6 c9 40 7d 48 53 b8 d9 81 d2 57 2c b0 0c

17. Hash_49 result:

d8 5f f2 a9 39 4a 0c a1 df e5 91 c4 07 b0 e2 fa 1b 4c 63 25 ea e9 60 2f 82 69 65 71 32 20 f5 a3 fa d0 7c d1 dc ac ac 2f 67 4d 20 93 a4 0a be c0 24

18. Hash_48 result:

35 96 89 d8 e4 78 45 b5 aa b9 0d b7 f3 08 2c 2b 44 50 9d 8f fb f2 4d 79 7a a9 2e 6b 22 f4 f9 0a 76 aa 6a 29 ab e1 c6 66 3a 58 b1 ac 9a e5 e7 70

19. Hash_47 result:

a2 24 03 16 b3 b9 1c 65 ce 70 52 da 92 0d 48 93 50 10 03 f7 4b d0 31 9f 1b 93 12 29 18 a5 a3 4b e8 50 99 84 e7 1d bc f8 3c 85 c4 54 4a 50 c2

20. Hash_46 result:

9f cb 4a 31 0f 2c a7 49 ed dd f5 9e ff 24 28 35 66 2e 1a d2 2c c1 16 63 67 c1 eb 86 f4 9c 63 33 22 30 29 86 12 5e 1d f6 86 7a d3 d4 5b 36

21. Hash_45 result:

83 36 9d c6 5a 1d bb 96 5b f5 58 97 d5 8c 24 74 67 3e 0c 06 78 32 ae 52 71 f0 79 ee 5d e6 03 f6 03 0f af e2 c3 af a0 cd 14 16 ad 62 4b

22. Hash_44 result:

14 60 d9 bd 00 8f b9 6c 7a 5b af d8 0d 16 31 13 44 60 a8 b5 e0 f8 62 43 63 2d 83 40 96 f5 34 68 94 c2 40 d2 f5 c3 52 83 31 20 af c2

23. Hash_43 result:

98 57 19 ab 95 8d 72 87 57 ea d4 9e 1f 19 86 50 4e 8e 52 a8 b3 f1 58 c8 e5 a7 c0 5f 5f 29 76 85 71 d8 26 44 29 d6 de b3 bf f5 1f

24. Hash_42 result:

39 fe d8 03 ad 49 fb f0 3d a9 ca d5 d0 53 fc 8b 7e 7f ed b7 0a 51 b1 19 a2 59 5a c0 e0 d5 16 d0 34 1b 53 4b cd ee 01 ed 0c b6

25. Hash_41 result:

90 5c 5a cb 57 02 ff 79 3f 48 15 32 c2 60 cf 56 ee 03 46 ad 15 f8 0e 24 97 cf 14 ce 7e ca 14 4c 8c 15 b9 d5 a1 f7 d4 c6 c8

26. Hash_40 result:

0d 7f f6 79 4f e6 b1 1b 4f b1 8c 54 67 99 cd 86 ef 62 7b 91 60 a4 96 e6 73 d0 b9 cc f4 38 7d 3f 37 cd c7 80 39 0c 0b 2c

27. Hash_39 result:

c2 25 8d ec 42 af 0e 15 5b 79 d3 08 8e 83 c9 23 40 da 7d 00 6d 79 a4 f5 b8 7d 41 d4 57 4e 65 d3 fc 0f 84 33 bb 30 d7

28. Hash_38 result:

ac 2a b1 34 88 b9 43 2f ed 4d 9d 23 ca 8b 5c 0b df d4 34 e6 dd 4a 4a 25 bf 55 b1 6e 91 42 f3 1a 33 7f ff 20 98 5c

29. Hash_37 result:

9a 27 54 85 8a 1d 14 98 01 35 0b 84 42 c7 f7 57 a9 f6 b8 cb e4 b9 22 f7 48 c8 0a 81 c4 00 ba 3b 8d 3e 01 7d b5

30. Hash_36 result:

72 b1 9c e8 9f 26 29 0b 4d ee 77 82 17 b0 f9 9d 1b 38 c2 95 80 94 31 a7 11 3a c2 c7 ad 1e be 02 62 74 80 e0

31. Hash_35 result:

17 6d 4a 1d e6 ea 5d 1c c1 94 97 bb a5 5a 0f 11 7e 6a 39 af 39 b9 77 66 b1 28 df 2a 6b 4a 0a e5 bb a7 94

32. Hash_34 result:

4c ed 9d bf c3 78 00 de 59 92 12 a6 cc 2b 4c 43 3e de 77 9e 38 b2 d6 96 44 ac cb 2b 20 57 24 1e c3 47

33. Hash_33 result:

1a 62 a3 c3 2b ea 0c 81 06 ef c7 2f f7 01 29 47 fb 6c 04 57 23 9b fb 54 90 b9 0e 69 5e 9a f3 00 26

34. Hash_32 result:

fd 29 aa 2a b9 91 e2 df 3c f3 c6 2d 74 12 22 37 73 01 ad 08 3f 52 fb 33 03 b0 b7 9f 5b fd 32 4d

pDK = fd 29 aa 2a b9 91 e2 df 3c f3 c6 2d 74 12 22 37
73 01 ad 08 3f 52 fb 33 03 b0 b7 9f 5b fd 32 4d

Tmp = 2e d5 59 dc d7 28 9c 0e 98 41 78 7e 04 65 6e 22
66 d8 a7 97 e1 7a 66 81 fa f6 7e 57 19 17 07 a9
c3 b5 18 93 19 43 20 8a 6a b8 8a 52 ce 8c bb fd
ba d4 78 cd 93 d1 7b 00 8d fc d7 67 0a 37 27 cc

DK = 0a 77 a1 b3 b1 d8 07 57 a2 19 05 99 85 80 77 df
c0 13 ce b6 60 d1 dc 40 0a e9 86 73 80 b6 37 72

Password: “qwerty12345”
Salt: “AE 0F 55 70 C2 BD 3E 90 24 CA 76 7F B8 6D 10 87”
c: 1000
dCostLen: 256
dkLen: 32

DK = 29 a2 92 ef 5c 55 9d a5 75 ca e6 b7 49 69 73 85
78 e0 16 2d 1a 93 08 b1 c3 b1 3f 80 bf 62 5c e2

Password: “qwerty12345”
Salt: “AE 0F 55 70 C2 BD 3E 90 24 CA 76 7F B8 6D 10 87”
c: 1000000
dCostLen: 256
dkLen: 32

DK = fc f0 be d8 26 37 74 49 11 1f b9 51 40 09 e5 6b
2e ca 84 fb 65 7b f7 2c 76 74 0d 8c ea 44 f0 ca

Password: “qwerty12345”

Salt: "AE 0F 55 70 C2 BD 3E 90 24 CA 76 7F B8 6D 10 87"
c: 0
dCostLen: 32
dkLen: 32

```
DK =        00 69 ba 8f cc 03 4b 6e 2e ae cc 71 e9 c7 00 3b
            25 44 05 be c3 49 02 b9 d1 4d a3 8e b8 96 c7 44
```

Password: "qwerty12345"
Salt: "AE 0F 55 70 C2 BD 3E 90 24 CA 76 7F B8 6D 10 87"
c: 0
dCostLen: 32
dkLen: 64

```
DK =        85 4e 0f b2 be ce c5 85 87 56 2e fc 1f 83 36 02
            a9 95 e4 cf 46 cf 7e 39 fa 68 6e 75 a0 11 65 21
            c9 f2 e6 31 d7 fb 41 0f 04 b4 3a 19 1b 6a a4 a9
            9a 01 a6 ae e2 9a 38 9c 3b 08 c5 7c 30 79 b9 4d
```

7. Speed-Up

There are two tests result: speed for random passwords length 8 and speed for random passwords length 64 (c = 0, dCostLen = 256 in both tests).

```
#####################    test MCS_PSW speed    #####################

##########    password length = 8, test numbers = 100000    ##########

#####################    Time = 29.250000 sec.    #####################


#####################    test MCS_PSW speed    #####################

##########    password length = 64, test numbers = 100000    ##########

#####################    Time = 29.530001 sec.    #####################
```

References

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

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