Network Working Group
Internet-Draft
Intended status: Informational
Expires: March 21, 2013

C. Percival
Tarsnap
S. Josefsson
SJD AB

September 17, 2012

The scrypt Password-Based Key Derivation Function
draft-josefsson-scrypt-kdf-00

Abstract

This document specify the password-based key derivation function scrypt. The function is used to derive one or more secret keys from a secret string. It is based on memory-hard functions which offers some added protection against attacks using custom hardware. The document also provide an ASN.1 schema.

Status of this Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at http://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on March 21, 2013.

Copyright Notice

described in the Simplified BSD License.

Table of Contents

1. Introduction

Password-based key derivation functions are used in cryptography for deriving one or more secret keys from a secret value. Over the years, several password-based key derivation functions have been used, including the original DES-based UNIX Crypt-function, FreeBSD MD5 crypt, PKCS#5 PBKDF2 [RFC2898] (typically used with SHA-1), GNU SHA-256/512 crypt, Windows NT LAN Manager (NTLM) hash, and Blowfish-based bcrypt. These algorithms are based on similar techniques that employ a cryptographic primitive, salting and/or iteration. The iteration count is used to slow down the computation.

Providing that the number of iterations used is increased as computer systems get faster, this allows legitimate users to spend a constant amount of time on key derivation without losing ground to attackers' ever-increasing computing power - as long as attackers are limited to the same software implementations as legitimate users. However, as Bernstein famously pointed out in the context of integer factorization, while parallelized hardware implementations may not change the number of operations performed compared to software implementations, this does not prevent them from dramatically changing the asymptotic cost, since in many contexts - including the embarrassingly parallel task of performing a brute-force search for a passphrase - dollar-seconds are the most appropriate units for measuring the cost of a computation. As semiconductor technology develops, circuits do not merely become faster; they also become smaller, allowing for a larger amount of parallelism at the same cost. Consequently, using existing key derivation algorithms, even if the iteration count is increased such that the time taken to verify a password remains constant, the cost of finding a password by using a brute force attack implemented in hardware drops each year.

The scrypt function aims to reduce the advantage which attackers can gain by using custom-designed parallel circuits for breaking password-based key derivation functions.

For further background, see the original scrypt paper [SCRYPT].

The rest of this document is divided into sections that each describe algorithms needed for the final "scrypt" algorithm.

2. The Salsa20/8 core Function

Salsa20/8 core is a round-reduced variant of the Salsa20 core. It is a hash function from 64-octet strings to 64-octet strings. Note that these functions are not cryptographic hash function since they are not collision-resistant. See [SALSA20CORE] for the specification.

Below is reference code for the Salsa20/8 core function, for illustration.

```
#define R(a,b) (((a) << (b)) | ((a) >> (32 - (b))))
void salsa208_word_specification(uint32 out[16],uint32 in[16])
{
  int i;
  uint32 x[16];
  for (i = 0;i < 16;++i) x[i] = in[i];
  for (i = 8;i > 0;i -= 2) {
    x[ 4] ^= R(x[ 0]+x[12], 7);  x[ 8] ^= R(x[ 4]+x[ 0], 9);
    x[12] ^= R(x[ 8]+x[ 4],13);  x[ 0] ^= R(x[12]+x[ 8],18);
    x[ 9] ^= R(x[ 5]+x[ 1], 7);  x[13] ^= R(x[ 9]+x[ 5], 9);
    x[ 1] ^= R(x[13]+x[ 9],13);  x[ 5] ^= R(x[ 1]+x[13],18);
    x[14] ^= R(x[10]+x[ 6], 7);  x[ 2] ^= R(x[14]+x[10], 9);
    x[ 6] ^= R(x[ 2]+x[14],13);  x[10] ^= R(x[ 6]+x[ 2],18);
    x[ 3] ^= R(x[15]+x[11], 7);  x[ 7] ^= R(x[ 3]+x[15], 9);
    x[11] ^= R(x[ 7]+x[ 3],13);  x[15] ^= R(x[11]+x[ 7],18);
    x[ 1] ^= R(x[ 0]+x[ 3], 7);  x[ 2] ^= R(x[ 1]+x[ 0], 9);
    x[ 3] ^= R(x[ 2]+x[ 1],13);  x[ 0] ^= R(x[ 3]+x[ 2],18);
    x[ 6] ^= R(x[ 5]+x[ 4], 7);  x[ 7] ^= R(x[ 6]+x[ 5], 9);
    x[ 4] ^= R(x[ 7]+x[ 6],13);  x[ 5] ^= R(x[ 4]+x[ 7],18);
    x[11] ^= R(x[10]+x[ 9], 7);  x[ 8] ^= R(x[11]+x[10], 9);
    x[ 9] ^= R(x[ 8]+x[11],13);  x[10] ^= R(x[ 9]+x[ 8],18);
    x[12] ^= R(x[15]+x[14], 7);  x[13] ^= R(x[12]+x[15], 9);
    x[14] ^= R(x[13]+x[12],13);  x[15] ^= R(x[14]+x[13],18);
  }
  for (i = 0;i < 16;++i) out[i] = x[i] + in[i];
}
```

3. The scryptBlockMix Algorithm

We now describe the scryptBlockMix algorithm. scryptBlockMix is the same as the BlockMix function described in [SCRYPT] but with the Salsa20/8 core function used as the hash function H. Below, Salsa(T) corresponds to the Salsa20/8 core function applied to the octet vector T.

Algorithm scryptBlockMix

Parameters:
r Block size parameter.

Input:
B[0], ..., B[2 * r - 1]
Input vector of 2 * r 64-octet blocks.

Output:
B'[0], ..., B'[2 * r - 1]
Output vector of 2 * r 64-octet blocks.

Steps:

1. X = B[2 * r - 1]

2. for i = 0 to 2 * r - 1 do
   T = X xor B[i]
   X = Salsa (T)
   Y[i] = X
   end for

3. B' = (Y[0], Y[2], ..., Y[2 * r - 2],
Y[1], Y[3], ..., Y[2 * r - 1])

4. The scryptROMix Algorithm

We now describe the scryptROMix algorithm. scryptROMix is the same as the ROMix function described in [SCRYPT] but with the scryptBlockMix algorithm used as the hash function H and the Integerify function explained inline.

Algorithm scryptROMix

Input:

r	Block size parameter.
B	Input octet vector of length 128 * r octets.
N	CPU/Memory cost parameter, must be larger than 1, a power of 2 and less than 2^(128 * r / 8).

Output:

B'	Output octet vector of length 128 * r octets.

Steps:

1. X = B

```
2. for i = 0 to N - 1 do
     V[i] = X
     X = scryptBlockMix (X)
   end for

3. for i = 0 to N - 1 do
     j = Integerify (X) mod N
            where Integerify (B[0] ... B[2 * r - 1]) is defined
            as the result of interpreting B[2 * r - 1] as a
            little-endian integer.
     T = X xor V[j]
     X = scryptBlockMix (T)
   end for

4. B' = X
```

5. The scrypt Algorithm

We now describe the scrypt algorithm.

The PBKDF2-HMAC-SHA-256 function used below denote the PBKDF2 algorithm [RFC2898] used with HMAC-SHA-256 [RFC6234] as the PRF. The HMAC-SHA-256 function generates 32 octet outputs.

Algorithm scrypt

Input:

P	Passphrase, an octet string.
S	Salt, an octet string.
r	Block size parameter.
N	CPU/Memory cost parameter, must be larger than 1, a power of 2 and less than 2^(128 * r / 8).

```
          p       Parallelization parameter, a positive integer
                  less than or equal to ((2^32-1) * hLen) / MFLen
                  where hLen is 32 and MFlen is 128 * r.
          dkLen   Intended output length in octets of the derived
                  key; a positive integer less than or equal to
                  (2^32 - 1) * hLen where hLen is 32.
```

Output:

```
          DK      Derived key, of length dkLen octets.
```

Steps:

```
  1. B[0] || B[1] || ... || B[p - 1] =
       PBKDF2-HMAC-SHA256 (P, S, 1, p * 128 * r)

  2. for i = 0 to p - 1 do
       B[i] = scryptROMix (r, B[i], N)
     end for

  3. DK = PBKDF2-HMAC-SHA256 (P, B[0] || B[1] || ... || B[p - 1],
                                 1, dkLen)
```

6. ASN.1 Syntax

This section defines ASN.1 syntax for the scrypt key derivation function. The intended application of these definitions includes PKCS #8 and other syntax for key management. (Various aspects of ASN.1 are specified in several ISO/IEC standards.)

The object identifier id-scrypt identifies the scrypt key derivation function.

```
id-scrypt OBJECT IDENTIFIER ::= {1 3 6 1 4 1 11591 4 11}
```

The parameters field associated with this OID in an AlgorithmIdentifier shall have type scrypt-params:


```
scrypt-params ::= SEQUENCE {
```

```
        salt OCTET STRING,
        blockSize INTEGER (1..MAX),
        costParameter INTEGER (1..MAX),
        parallelizationParameter INTEGER (1..MAX),
        keyLength INTEGER (1..MAX) OPTIONAL }
```

The fields of type scrypt-params have the following meanings:

- salt specifies the salt value. It shall be an octet string.

- blockSize specifies the block size parameter r.

- costParameter specifies the CPU/Memory cost parameter.

- parallelizationParameter specifies the parallelization parameter.

- keyLength, an optional field, is the length in octets of the derived key. The maximum key length allowed depends on the implementation; it is expected that implementation profiles may further constrain the bounds. The field is provided for convenience only; the key length is not cryptographically protected.

6.1. ASN.1 Module

For reference purposes, the ASN.1 syntax is presented as an ASN.1 module here.

```
-- scrypt ASN.1 Module

scrypt-0 {1 3 6 1 4 1 11591 4 10}

DEFINITIONS ::= BEGIN

id-scrypt OBJECT IDENTIFIER ::= {1 3 6 1 4 1 11591 4 11}

scrypt-params ::= SEQUENCE {
    salt OCTET STRING,
    blockSize INTEGER (1..MAX),
    costParameter INTEGER (1..MAX),
    parallelizationParameter INTEGER (1..MAX),
    keyLength INTEGER (1..MAX) OPTIONAL
}

END
```

7. Test Vectors for Salsa20/8 core

The value is hex encoded and whitespace is inserted for readability. The value correspond to the first input and output pair generated by the first scrypt test vector below.

```
INPUT:
7e879a21 4f3ec986 7ca940e6 41718f26
baee555b 8c61c1b5 0df84611 6dcd3b1d
ee24f319 df9b3d85 14121e4b 5ac5aa32
76021d29 09c74829 edebc68d b8b8c25e

OUTPUT:
a41f859c 6608cc99 3b81cacb 020cef05
044b2181 a2fd337d fd7b1c63 96682f29
b4393168 e3c9e6bc fe6bc5b7 a06d96ba
e424cc10 2c91745c 24ad673d c7618f81
```

8. Test Vectors for scryptBlockMix

The following test vector use a r value of 1. The value is hex encoded and whitespace is inserted for readability. The value correspond to the first input and output pair generated by the first scrypt test vector below.

```
INPUT
B[0] =  f7ce0b65 3d2d72a4 108cf5ab e912ffdd
        777616db bb27a70e 8204f3ae 2d0f6fad
        89f68f48 11d1e87b cc3bd740 0a9ffd29
        094f0184 639574f3 9ae5a131 5217bcd7

B[1] =  89499144 7213bb22 6c25b54d a86370fb
        cd984380 374666bb 8ffcb5bf 40c254b0
        67d27c51 ce4ad5fe d829c90b 505a571b
        7f4d1cad 6a523cda 770e67bc eaaf7e89

OUTPUT
B'[0] = a41f859c 6608cc99 3b81cacb 020cef05
        044b2181 a2fd337d fd7b1c63 96682f29
        b4393168 e3c9e6bc fe6bc5b7 a06d96ba
        e424cc10 2c91745c 24ad673d c7618f81

B'[1] = 20edc975 323881a8 0540f64c 162dcd3c
        21077cfe 5f8d5fe2 b1a4168f 953678b7
        7d3b3d80 3b60e4ab 920996e5 9b4d53b6
```

```
          5d2a2258 77d5edf5 842cb9f1 4eefe425
```


9. Test Vectors for scryptROMix

The following test vector use a r value of 1 and N value of 16. The value is hex encoded and whitespace is inserted for readability. The value correspond to the first input and output pair generated by the first scrypt test vector below.

```
INPUT:
B = f7ce0b65 3d2d72a4 108cf5ab e912ffdd
    777616db bb27a70e 8204f3ae 2d0f6fad
    89f68f48 11d1e87b cc3bd740 0a9ffd29
    094f0184 639574f3 9ae5a131 5217bcd7
    89499144 7213bb22 6c25b54d a86370fb
    cd984380 374666bb 8ffcb5bf 40c254b0
    67d27c51 ce4ad5fe d829c90b 505a571b
    7f4d1cad 6a523cda 770e67bc eaaf7e89

OUTPUT:
B = 79ccc193 629debca 047f0b70 604bf6b6
    2ce3dd4a 9626e355 fafc6198 e6ea2b46
    d5841367 3b99b029 d665c357 601fb426
    a0b2f4bb a200ee9f 0a43d19b 571a9c71
    ef1142e6 5d5a266f ddca832c e59faa7c
    ac0b9cf1 be2bffca 300d01ee 387619c4
    ae12fd44 38f203a0 e4e1c47e c314861f
    4e9087cb 33396a68 73e8f9d2 539a4b8e
```

10. Test Vectors for PBKDF2 with HMAC-SHA-256

The test vectors below can be used to verify the PBKDF2-HMAC-SHA-256 [RFC2898] function. The password and salt strings are passed as sequences of ASCII [ANSI.X3-4.1986] octets.

```
PBKDF2-HMAC-SHA-256 (P="passwd", S="salt",
                     c=1, dkLen=64) =
55 ac 04 6e 56 e3 08 9f ec 16 91 c2 25 44 b6 05
f9 41 85 21 6d de 04 65 e6 8b 9d 57 c2 0d ac bc
49 ca 9c cc f1 79 b6 45 99 16 64 b3 9d 77 ef 31
```

```
7c 71 b8 45 b1 e3 0b d5 09 11 20 41 d3 a1 97 83
```

```
PBKDF2-HMAC-SHA-256 (P="Password", S="NaCl",
                     c=80000, dkLen=64) =
4d dc d8 f6 0b 98 be 21 83 0c ee 5e f2 27 01 f9
64 1a 44 18 d0 4c 04 14 ae ff 08 87 6b 34 ab 56
a1 d4 25 a1 22 58 33 54 9a db 84 1b 51 c9 b3 17
6a 27 2b de bb a1 d0 78 47 8f 62 b3 97 f3 3c 8d
```


11. Test Vectors for scrypt

For reference purposes, we provide the following test vectors for scrypt, where the password and salt strings are passed as sequences of ASCII [ANSI.X3-4.1986] octets.

The parameters to the scrypt function below are, in order, the password (octet string), the salt (octet string), the CPU/Memory cost parameter N, the block size parameter r, and the parallelization parameter p, and the output size dkLen. The output is hex encoded and whitespace is inserted for readability.

```
scrypt (P="", S="",
        r=16, N=1, p=1, dklen=64) =
77 d6 57 62 38 65 7b 20 3b 19 ca 42 c1 8a 04 97
f1 6b 48 44 e3 07 4a e8 df df fa 3f ed e2 14 42
fc d0 06 9d ed 09 48 f8 32 6a 75 3a 0f c8 1f 17
e8 d3 e0 fb 2e 0d 36 28 cf 35 e2 0c 38 d1 89 06
```

```
scrypt (P="password", S="NaCl",
        r=1024, N=8, p=16, dkLen=64) =
fd ba be 1c 9d 34 72 00 78 56 e7 19 0d 01 e9 fe
7c 6a d7 cb c8 23 78 30 e7 73 76 63 4b 37 31 62
2e af 30 d9 2e 22 a3 88 6f f1 09 27 9d 98 30 da
c7 27 af b9 4a 83 ee 6d 83 60 cb df a2 cc 06 40
```

```
scrypt (P="pleaseletmein", S="SodiumChloride",
        r=16384, N=8, p=1, dkLen=64) =
70 23 bd cb 3a fd 73 48 46 1c 06 cd 81 fd 38 eb
fd a8 fb ba 90 4f 8e 3e a9 b5 43 f6 54 5d a1 f2
```

```
d5 43 29 55 61 3f 0f cf 62 d4 97 05 24 2a 9a f9
e6 1e 85 dc 0d 65 1e 40 df cf 01 7b 45 57 58 87


scrypt (P="pleaseletmein", S="SodiumChloride",
        r=1048576, N=8, p=1, dkLen=64) =
21 01 cb 9b 6a 51 1a ae ad db be 09 cf 70 f8 81
ec 56 8d 57 4a 2f fd 4d ab e5 ee 98 20 ad aa 47
8e 56 fd 8f 4b a5 d0 9f fa 1c 6d 92 7c 40 f4 c3
37 30 40 49 e8 a9 52 fb cb f4 5c 6f a7 7a 41 a4
```

12. Acknowledgements

Text in this document was borrowed from [SCRYPT] and [RFC2898].

13. IANA Considerations

None.

14. Security Considerations

This document specify a cryptographic algorithm. The reader must follow cryptographic research to notice published attacks. ROMix has been proven sequential memory-hard under the Random Oracle model for the hash function. The security of scrypt relies on the assumption that BlockMix with Salsa20/8 does not exhibit any "shortcuts" which would allow it to be iterated more easily than a random oracle. For other claims about the security properties see [SCRYPT].

Passwords and other sensitive data, such as intermediate values, may continue to be stored in memory, core dumps, swap areas, etc, a long time after the implementation has finished processing them. This can make attacks on the implementation easier. Thus, implementation should consider storing sensitive data in protected memory areas. How to achieve that is system dependent.

By nature and depending on parameters, running the scrypt algorithm may require large amounts of memory. Systems should protect against a denial of service attack resulting from attackers presenting

unreasonable large parameters.

15. References

15.1. Normative References

[RFC2898] Kaliski, B., "PKCS #5: Password-Based Cryptography Specification Version 2.0", RFC 2898, September 2000.

[RFC6234] Eastlake, D. and T. Hansen, "US Secure Hash Algorithms (SHA and SHA-based HMAC and HKDF)", RFC 6234, May 2011.

[SALSA20CORE] Bernstein, D., "The Salsa20 core", WWW http://cr.yp.to/salsa20.html, March 2005.

15.2. Informative References

[ANSI.X3-4.1986] American National Standards Institute, "Coded Character Set - 7-bit American Standard Code for Information Interchange", ANSI X3.4, 1986.

[SCRYPT] Percival, C., "Stronger key derivation via sequential memory-hard functions", BSDCan'09 http://www.tarsnap.com/scrypt/scrypt.pdf, May 2009.

Authors' Addresses

Colin Percival
Tarsnap

Email: cperciva@tarsnap.com

Simon Josefsson
SJD AB

Email: simon@josefsson.org

URI: http://josefsson.org/