NeoScrypt,

a Strong Memory Intensive Key Derivation Function

John Doering <ghostlander@phoenixcoin.org>

ABSTRACT. Hereby presented a new password based memory intensive cryptographic solution designed for general purpose computer hardware. A particular 32-bit implementation is described and evaluated.

1. INTRODUCTION

Password based key derivation function (KDF) is a deterministic algorithm used to derive a cryptographic key from an input datum known as a password. An additional input datum known as a salt may be employed in order to increase strength of the algorithm against attacks using pre-computed hashes also known as rainbow tables. The derived key length may be specified usually, and one of the most popular uses of KDFs is key stretching. It increases effective length of a user password by constructing an enhanced key to provide with a better resistance against brute force attacks. Another popular use is password storage. Keeping user passwords in unencrypted form is very undesired as it may be possible for an attacker to gain access to the password file and retrieve the passwords stored immediately. Brute force attacks may be the only possible approach against strong KDFs. This kind of attack can be parallelised usually to a great extent. High requirements on computational resources such as processor time and memory space allow to reduce parallelisation efficiency and keep these attacks expensive far beyond reasonable limits.

As the name suggests, NeoScrypt is a further development of Scrypt as described in Percival [1]. It is aimed at increased security and better performance on general purpose computer hardware while maintaining comparable costs and requirements. This document focuses on functional differences between NeoScrypt and Scrypt.

2. SCRYPT SPECIFICATIONS

The most popular implementation of Scrypt employed by many cryptocurrencies since 2011 is N = 1024, r = 1, p = 1 abbreviated usually to (1024, 1, 1). N is the primary parameter defining number of memory segments used and must be a power of 2. May be also described through Nfactor.

N = (1 << (Nfactor + 1))

Nfactor = lb(N) - 1

The default memory segment size for the 32-bit implementation is 128 bytes. r is the segment size multiplier. p is the computational multiplier. They may be also described through rfactor and pfactor respectively.

r = (1 << rfactor)
p = (1 << pfactor)

A single instance of Scrypt utilises (N + 2) * r * 128 bytes of memory space, i.e. 128.25Kb for the (1024, 1, 1) configuration. Actual data mixing in memory is performed by Salsa20, a stream cipher introduced by Bernstein [2]. A reduced strength 8-round implementation has been chosen (Salsa20/8). Every run of the Scrypt core engine executes it 4 * r * N times, i.e. 4096 times for the (1024, 1, 1) configuration. Every execution of Salsa20 mixes one half of a memory segment with itself.

The Scrypt core engine has no provisions for key stretching or compressing as well as salting, therefore additional cryptographic functions need to be deployed. In case of cryptocurrencies, a typical configuration operates with 80 bytes of input data (block header) which is also a salt. It is passed to PBKDF2, a password based KDF [3] capable of deriving variable length keys with salting. It works with SHA-256, a cryptographic hash function delivering digests up to 32 bytes in size through 64 internal rounds. It doesn't support keyed hashing, therefore a pseudorandom function (PRF) such as HMAC [4] is required, and the whole big endian construction may be called PBKDF2-HMAC-SHA256. It feeds r * 128 bytes of derived data to the Scrypt core and receives it back after mixing to be used as a salt for another PBKDF2-HMAC-SHA256 run which compresses 80 bytes of input data into 32 bytes of hash.

3. NEOSCRYPT SPECIFICATIONS

Although a very innovative design back in time, Scrypt has developed certain vulnerabilities. The first announced differential cryptanalysis of Salsa20/8 by Tsunoo et al. [5] in 2007 did not deliver any advantage over 256-bit brute force attack, but the following research by Aumasson et al. [6] reduced time complexity to break it from 2^{255} to 2^{251} with 50% success probability. It was improved by Shi et. al [7] in 2012 to 2^{250}. Although this is not critical yet, better attacks on Salsa20/8 may be developed in the future.

PBKDF2 is a very popular KDF and may be configured to require considerably large amounts of processor time, but it does not require complex logic or significant amounts of memory to operate. Therefore brute force attacks can be carried out on general purpose hardware such as GPUs or custom designs (ASICs) with reasonably low costs. SHA-256 also allows numerous performance optimisations in this context. It is also worth to mention that Scrypt relies very little on PBKDF2-HMAC-SHA256 strength as it is configured to run in the fastest 1-iteration mode even though 1000-iteration minimum advised in general [3].

NeoScrypt addresses these issues. The core engine is configured to employ non-reduced Salsa20 of 20 rounds (Salsa20/20) as well as non-reduced ChaCha20 of 20 rounds (ChaCha20/20) [8]. Both of them are used to produce the final salt as their outputs are XOR'ed into it. They may be configured to run either in series or parallel depending on application objectives. The default NeoScrypt configuration is (128, 2, 1). A single instance of NeoScrypt utilises (N + 3) * r * 128 bytes of memory space, i.e. 32.75Kb, in series mode or (2 * N + 3) * r * 128 bytes, i.e. 64.75Kb, in parallel mode. Every run of the NeoScrypt core engine executes Salsa20/20 and ChaCha20/20 1024 times each which might seem inferior to 4096 times of Salsa20/8 of the Scrypt core engine. However NeoScrypt operates with double the memory segment size requiring larger temporal buffers, also with higher round count of each stream cipher iteration as explained above. If approximated to abstract load/store units, NeoScrypt is 1.25 times more memory intensive than Scrypt.

There are no known successful attacks on non-reduced Salsa20 and ChaCha20 other than exhaustive brute force search.

NeoScrypt replaces SHA-256 with BLAKE2s [9] which is a further development of BLAKE-256 [10], one of 5 NIST SHA-3 contest finalists. Based upon ChaCha20 , operates with a lower round count of

10, supports keyed hashing, is native little endian and faster significantly than SHA-256 and even BLAKE-256. It could be interfaced directly to PBKDF2 with no need of HMAC. However PBKDF2 constructs derived keys using blocks. It means a minor change in an input datum, such as nonce increment, may not result in an entirely different derived key. A replacement KDF has been developed to address this issue.

FastKDF is a buffered password based KDF which also supports salting. It operates with 2 primary buffers for password and salt each. They must be a power of 2 in size and not less than any input (password, salt) or output (derived key) data. The default configuration works with 256-byte buffers. Password and salt are loaded initially into these buffers in a repetitive manner until the end of buffer is reached. The salt buffer is modified through operations while the password buffer remains constant. The buffer pointers are set to zero (start) on the first run. When a PRF chosen delivers a digest, a sum of all its bytes modulo buffer size defines the next buffer pointer. The digest is XOR'ed into the salt buffer at the new buffer pointer and the next iteration starts. If a read or write operation goes past a buffer end, it is continued from the buffer start. BLAKE2s is configured to operate with 64-byte input (password), 32-byte key (salt) and 32-byte output (digest). When the final FastKDF iteration is completed, the password buffer using zero buffer pointer is XOR'ed into the salt buffer using the last buffer pointer to produce the derived key of length required which is copied into the output buffer. FastKDF-BLAKE2s is configured to run through 32 iterations by default. It is little endian for easier deployment and additional minor performance advantage on popular general purpose computer hardware.

4. CONCLUSIONS

The primary functionality of NeoScrypt and Scrypt has been described and evaluated briefly without much mathematical detail to a cryptography amateur. Certain disadvantages of Scrypt have been outlined. Please refer to the source code and the original Scrypt documentation [1] for additional information should you need any.

REFERENCES

1. Colin Percival. Stronger Key Derivation via Sequential Memory-Hard Functions, May 2009

2. Daniel J. Bernstein. The Salsa20 family of stream ciphers, December 2007

3. IETF RFC 2898. PKCS #5: Password-based Cryptography Specification Version 2.0, September 2000

4. FIPS 198-1. The Keyed-Hash Message Authentication Code (HMAC), July 2008

5. Yukiyasu Tsunoo, Teruo Saito, Hiroyasu Kubo, Tomoyasu Suzaki and Hiroki Nakashima. Differential Cryptanalysis of Salsa20/8, January 2007

6. Jean-Philippe Aumasson, Simon Fischer, Shahram Khazaei, Willi Meier and Christian Rechberger. New Features of Latin Dances: Analysis of Salsa, ChaCha, and Rumba, December of 2007

7. Zhenqing Shi, Bin Zhang, Dengguo Feng and Wenling Wu. Improved Key Recovery Attacks on Reduced-Round Salsa20 and ChaCha, November 2012

8. Daniel J. Bernstein. ChaCha, a variant of Salsa20, January 2008

9. Jean-Philippe Aumasson, Samuel Neves, Zooko Wilcox-O'Hearn and Christian Winnerlein. BLAKE2: simpler, smaller, fast as MD5, January 2013.

10. Jean-Philippe Aumasson, Luca Henzen, Willi Meier and Raphael C.-W. Phan. SHA-3 proposal BLAKE. Submission to NIST (Round 1/2), 2008.