Abstract

This document describes the use of the ChaCha stream cipher with HMAC-SHA1 and Poly1305 in Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS) protocols.

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 July 28, 2014.

Copyright Notice

Copyright (c) 2014 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust’s Legal Provisions Relating to IETF Documents (http://trustee.ietf.org/license-info) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect
to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Introduction ................................................. 3
2. The ChaCha Cipher .......................................... 4
3. The Poly1305 Authenticator ................................. 6
4. ChaCha20 Cipher Suites ...................................... 7
   4.1. ChaCha20 Cipher Suites with HMAC-SHA1 ................. 7
   4.2. ChaCha20 Cipher Suites with Poly1305 ................. 7
5. Updates to the TLS Standard Stream Cipher .................. 10
6. Updates to DTLS ............................................. 11
7. Acknowledgements ........................................... 12
8. IANA Considerations ......................................... 13
9. Security Considerations ..................................... 14
10. References .................................................. 15
   10.1. Normative References ................................. 15
   10.2. Informative References ............................... 15
Appendix A. Test vectors ....................................... 17
   A.1. ChaCha20 .............................................. 17
   A.2. Poly1305 .............................................. 18
   A.3. AEAD_CHACHA20_POLY1305 ............................. 18
Authors’ Addresses ............................................. 20
1. Introduction

This document describes the use of the ChaCha stream cipher in the Transport Layer Security (TLS) version 1.0 [RFC2246], TLS version 1.1 [RFC4346], and TLS version 1.2 [RFC5246] protocols, as well as in the Datagram Transport Layer Security (DTLS) versions 1.0 [RFC4347] and 1.2 [RFC6347]. It can also be used with Secure Sockets Layer (SSL) version 3.0 [RFC6101].

ChaCha [CHACHA] is a stream cipher that has been designed for high performance in software implementations. The cipher has compact implementation and uses few resources and inexpensive operations that makes it suitable for implementation on a wide range of architectures. It has been designed to prevent leakage of information through side channel analysis, has a simple and fast key setup and provides good overall performance. It is a variant of Salsa20 [SALSA20SPEC] which is one of the selected ciphers in the eSTREAM portfolio [ESTREAM].

Recent attacks [CBC-ATTACK] have indicated problems with CBC-mode cipher suites in TLS and DTLS as well as issues with the only supported stream cipher (RC4) [RC4-ATTACK]. While the existing AEAD (AES-GCM) ciphersuites address some of these issues, concerns about the performance and ease of software implementation are sometimes raised.

Therefore, a new stream cipher to replace RC4 and address all the previous issues is needed. It is the purpose of this document to describe a secure stream cipher for both TLS and DTLS that is comparable to RC4 in speed on a wide range of platforms and can be implemented easily without being vulnerable to software side-channel attacks.
2. The ChaCha Cipher

ChaCha [CHACHA] is a stream cipher developed by D. J. Bernstein in 2008. It is a refinement of Salsa20 and was used as the core of the SHA-3 finalist, BLAKE.

The variant of ChaCha used in this document is ChaCha with 20 rounds and a 256 bit key, which will be referred to as ChaCha20 in the rest of this document. This is the conservative variant (with respect to security) of the ChaCha family.

ChaCha maps 16, 32-bit input words to 16, 32-bit output words. By convention, 8 of the input words consist of a 256-bit key, 4 are constants and the remaining four are a nonce and block counter. The output words are converted to bytes and XORed with the plaintext to produce ciphertext. In order to generate sufficient output bytes to XOR with the whole plaintext, the block counter is incremented and ChaCha is run again, as many times as needed, for up to $2^{70}$ bytes of output.

ChaCha operates on a state of 16, 32-bit words which are initialised from the input words. The first four input words are constants:

$$(0x61707865, 0x3320646e, 0x79622d32, 0x6b206574)$$

Input words 4 through 11 are taken from the 256-bit key by reading the bytes in little-endian order, in 4-byte chunks. Input words 12 and 13 are a block counter, with word 12 overflowing into word 13. Lastly, words 14 and 15 are taken from an 8-byte nonce, again by reading the bytes in little-endian order, in 4-byte chunks. The block counter words are initially zero.

ChaCha20 consists of 20 rounds, alternating between "column" rounds and "diagonal" rounds. Each round applies the following "quarter-round" function four times, to a different set of words each time. The quarter-round function updates 4, 32-bit words $(a, b, c, d)$ as follows, where $<<<$ is a bitwise, left rotation:

$$
a += b; d ^= a; d <<<= 16;
b += c; b <<<= 12;
a += b; d ^= a; d <<<= 8;
c += d; b ^= c; b <<<= 7;
$$

The 16 words are conceptually arranged in a four by four grid with the first word in the top-left position and the fourth word in the top-right position. The "column" rounds then apply the quarter-round function to the four columns, from left to right. The "diagonal" rounds apply the quarter-round to the top-left, bottom-right
diagonal, followed by the pattern shifted one place to the right, for three more quarter-rounds.

Specifically, a column round applies the quarter-round function to the following indexes: (0, 4, 8, 12), (1, 5, 9, 13), (2, 6, 10, 14), (3, 7, 11, 15). A diagonal round applies it to these indexes: (0, 5, 10, 15), (1, 6, 11, 12), (2, 7, 8, 13), (3, 4, 9, 14).

After 20 rounds of the above processing, the original 16 input words are added to the 16 words to form the 16 output words.

The 64 output bytes are generated from the 16 output words by serializing them in little-endian order and concatenating the results.

Test vectors for this cipher can be found in Appendix A.1.
3. The Poly1305 Authenticator

Poly1305 [POLY1305] is a Wegman-Carter, one-time authenticator designed by D. J. Bernstein. Poly1305 takes a 32-byte, one-time key and a message and produces a 16-byte tag that authenticates the message such that an attacker has a negligible chance of producing a valid tag for an inauthentic message.

The first 16 bytes of the one-time key form an integer, \( r \), as follows: the top four bits of the bytes at indexes 3, 7, 11 and 15 are cleared, the bottom 2 bits of the bytes at indexes 4, 8 and 12 are cleared and the 16 bytes are taken as a little-endian value.

An accumulator is set to zero. For each chunk of 16 bytes from the input message, a byte with value 1 is appended and the 17 bytes are treated as a little-endian number. If the last chunk has less than 16 bytes then zero bytes are appended after the 1 byte is appended until there are 17 bytes. The value is added to the accumulator and then the accumulator is multiplied by \( r \), all mod \( 2^{130} - 5 \).

Finally the last 16 bytes of the one-time key are treated as a little-endian number and added to the accumulator, mod \( 2^{128} \). The result is serialised as a little-endian number, producing the 16 byte tag. Note that the original specification of Poly1305 used a different construction with AES to generate the constant term of the polynomial from a counter nonce. For a more recent treatment that avoids the use of a block cipher in this fashion, as is done here, see section 9 of the NaCl specification [NACLCRYPTO].

Test vectors for this authenticator can be found in Appendix A.2.
4. ChaCha20 Cipher Suites

In the next sections different ciphersuites are defined that utilize the ChaCha20 cipher combined with various message authentication methods.

In all cases, the pseudorandom function (PRF) for TLS 1.2 is the TLS PRF with SHA-256 as the hash function. When used with TLS versions prior to 1.2, the PRF is calculated as specified in the appropriate version of the TLS specification.

The RSA, DHE_RSA, ECDHE_RSA, ECDHE_ECDSA, PSK, DHE_PSK, RSA_PSK, ECDHE_PSK key exchanges are performed as defined in [RFC5246], [RFC4492], and [RFC5489].

4.1. ChaCha20 Cipher Suites with HMAC-SHA1

The following CipherSuites are defined.

- TLS_RSA_WITH_CHACHA20_SHA = {0xTBD, 0xTBD} {0xCC, 0x20}
- TLS_ECDHE_RSA_WITH_CHACHA20_SHA = {0xTBD, 0xTBD} {0xCC, 0x21}
- TLS_ECDHE_ECDSA_WITH_CHACHA20_SHA = {0xTBD, 0xTBD} {0xCC, 0x22}
- TLS_DHE_RSA_WITH_CHACHA20_SHA = {0xTBD, 0xTBD} {0xCC, 0x23}
- TLS_DHE_PSK_WITH_CHACHA20_SHA = {0xTBD, 0xTBD} {0xCC, 0x24}
- TLS_PSK_WITH_CHACHA20_SHA = {0xTBD, 0xTBD} {0xCC, 0x25}
- TLS_ECDHE_PSK_WITH_CHACHA20_SHA = {0xTBD, 0xTBD} {0xCC, 0x26}
- TLS_RSA_PSK_WITH_CHACHA20_SHA = {0xTBD, 0xTBD} {0xCC, 0x27}

Note that ChaCha20 requires a 64-bit nonce. That nonce is updated on the encryption of every TLS record, and is set to be the 64-bit TLS record sequence number. In case of DTLS the 64-bit nonce is formed as the concatenation of the 16-bit epoch with the 48-bit sequence number.

The MAC algorithm used in the ciphersuites above is HMAC-SHA1 [RFC6234].

4.2. ChaCha20 Cipher Suites with Poly1305

The ChaCha20 and Poly1305 primitives are built into an AEAD algorithm [RFC5116], AEAD_CHACHA20_POLY1305, that takes a 32 byte key and 8 byte nonce as follows.

ChaCha20 is run with the given key and nonce and with the two counter words set to zero. The first 32 bytes of the 64 byte output are saved to become the one-time key for Poly1305. The remainder of the
output is discarded. The first counter input word is set to one and
the plaintext is encrypted by XORing it with the output of
invocations of the ChaCha20 function as needed, incrementing the
first counter word after each block and overflowing into the second.
The limits on the TLS plaintext size mean that the first counter word
will never overflow in practice.

The reason for generating the Poly1305 key like this rather than
using key material from the handshake is that handshake key material
is per-session, but for a polynomial MAC, a unique, secret key is
needed per-record.

The Poly1305 key is used to calculate a tag for the following input:
the concatenation of the additional data, the number of bytes of
additional data, the ciphertext and the number of bytes of
ciphertext. Numbers are represented as 8-byte, little-endian values.
The resulting tag is appended to the ciphertext, resulting in the
output of the AEAD operation.

Authenticated decryption is largely the reverse of the encryption
process: generate one block of ChaCha20 keystream and use the first
32 bytes as a Poly1305 key. Feed Poly1305 the additional data and
ciphertext, with the length suffixing as described above. Verify, in
constant time, that the calculated Poly1305 authenticator matches the
final 16 bytes of the input. If not, the input can be rejected
immediately. Otherwise, run ChaCha20, starting with a counter value
of one, to decrypt the ciphertext.

When used in TLS, the "record_iv_length" is zero and the nonce is the
sequence number for the record, as an 8-byte, big-endian number. The
additional data is seq_num + TLSCompressed.type +
TLSCompressed.version + TLSCompressed.length, where "+" denotes
concatenation.

In DTLS, the nonce is formed as the concatenation of the 16-bit epoch
with the 48-bit sequence number.

In accordance with section 4 of RFC 5116 [RFC5116], the constants for
this AEAD algorithm are as follows: K_LEN is 32 bytes, N_MIN and
N_MAX are 8 bytes, P_MAX and A_MAX are 2^64, C_MAX is 2^64+16. An
AEAD_CHACHA20_POLY1305 ciphertext is exactly 16 octets longer than
its corresponding plaintext.

Test vectors for this authenticator can be found in Appendix A.3.

The following CipherSuites are defined.
TLS_RSA_WITH_CHACHA20_POLY1305  = {0xTBD, 0xTBD} {0xCC, 0x12}
TLS_ECDHE_RSA_WITH_CHACHA20_POLY1305 = {0xTBD, 0xTBD} {0xCC, 0x13}
TLS_ECDHE_ECDSA_WITH_CHACHA20_POLY1305 = {0xTBD, 0xTBD} {0xCC, 0x14}
TLS_DHE_RSA_WITH_CHACHA20_POLY1305  = {0xTBD, 0xTBD} {0xCC, 0x15}
TLS_DHE_PSK_WITH_CHACHA20_POLY1305  = {0xTBD, 0xTBD} {0xCC, 0x16}
TLS_PSK_WITH_CHACHA20_POLY1305      = {0xTBD, 0xTBD} {0xCC, 0x17}
TLS_ECDHE_PSK_WITH_CHACHA20_POLY1305 = {0xTBD, 0xTBD} {0xCC, 0x18}
TLS_RSA_PSK_WITH_CHACHA20_POLY1305  = {0xTBD, 0xTBD} {0xCC, 0x19}
5. Updates to the TLS Standard Stream Cipher

The ChaCha20 ciphersuites with HMAC-SHA1 defined in this document differ from the TLS RC4 ciphersuites that have been the basis for the definition of Standard Stream Cipher. Unlike RC4, ChaCha20 requires a nonce per record. This however, does not affect the description of the Standard Stream Cipher if one assumes that a nonce is optional and depends on the cipher’s characteristics.

Hence, this document modifies the Standard Stream Cipher by adding an implicit nonce of 8-bytes, which is set to be the 64-bit TLS record sequence number. If the stream cipher needs more than 8 byte of nonce, it can obtain additional bytes for the implicit nonce from the client_write_iv and server_write_iv of the key_block.

Stream ciphers that don’t require a nonce such as RC4 shall ignore it. Other stream ciphers that require a nonce, such as ChaCha20 with HMAC-SHA1, will use the nonce and reset their state on each record.

Note that in case of DTLS the 8-byte nonce is formed as the concatenation of the 16-bit epoch with the 48-bit sequence number, which are sent as part of the record.
6. Updates to DTLS

The DTLS protocol requires the cipher in use to introduce no dependencies between TLS Records to allow lost or rearranged records. For that it explicitly bans stream ciphers (see Section 3.1 of [RFC6347]).

As the stream cipher described in this document, unlike RC4, does not require dependencies between records, this ban of stream ciphers is lifted with this document. Stream ciphers can be used with DTLS if they introduce no dependencies between records.
7. Acknowledgements

The authors would like to thank Zooko Wilcox-O’Hearn and Samuel Neves.
8. IANA Considerations

IANA is requested to assign a value for AEAD_CHACHA20_POLY1305 in the registry of AEAD algorithms [RFC5116], and also allocate the following Cipher Suites in the TLS Cipher Suite Registry (note that the third column contains the suggested ciphersuite numbers):

- TLS_RSA_WITH_CHACHA20_POLY1305 = {0xTBD, 0xTBD} {0xCC, 0x12}
- TLS_ECDHE_RSA_WITH_CHACHA20_POLY1305 = {0xTBD, 0xTBD} {0xCC, 0x13}
- TLS_ECDHE_ECDSA_WITH_CHACHA20_POLY1305 = {0xTBD, 0xTBD} {0xCC, 0x14}
- TLS_DHE_RSA_WITH_CHACHA20_POLY1305 = {0xTBD, 0xTBD} {0xCC, 0x15}
- TLS_DHE_PSK_WITH_CHACHA20_POLY1305 = {0xTBD, 0xTBD} {0xCC, 0x16}
- TLS_RSA_PSK_WITH_CHACHA20_POLY1305 = {0xTBD, 0xTBD} {0xCC, 0x17}
- TLS_ECDHE_PSK_WITH_CHACHA20_POLY1305 = {0xTBD, 0xTBD} {0xCC, 0x18}
- TLS_RSA_PSK_WITH_CHACHA20_SHA = {0xTBD, 0xTBD} {0xCC, 0x19}
- TLS_RSA_WITH_CHACHA20_SHA = {0xTBD, 0xTBD} {0xCC, 0x20}
- TLS_ECDHE_RSA_WITH_CHACHA20_SHA = {0xTBD, 0xTBD} {0xCC, 0x21}
- TLS_ECDHE_ECDSA_WITH_CHACHA20_SHA = {0xTBD, 0xTBD} {0xCC, 0x22}
- TLS_DHE_RSA_WITH_CHACHA20_SHA = {0xTBD, 0xTBD} {0xCC, 0x23}
- TLS_DHE_PSK_WITH_CHACHA20_SHA = {0xTBD, 0xTBD} {0xCC, 0x24}
- TLS_PSK_WITH_CHACHA20_SHA = {0xTBD, 0xTBD} {0xCC, 0x25}
- TLS_RSA_PSK_WITH_CHACHA20_SHA = {0xTBD, 0xTBD} {0xCC, 0x26}
- TLS_RSA_PSK_WITH_CHACHA20_SHA = {0xTBD, 0xTBD} {0xCC, 0x27}
9. Security Considerations

ChaCha20 follows the same basic principle as Salsa20, a cipher with significant security review [SALSA20-SECURITY][ESTREAM]. At the time of writing this document, there are no known significant security problems with either cipher, and ChaCha20 is shown to be more resistant in certain attacks than Salsa20 [SALSA20-ATTACK]. Furthermore ChaCha20 was used as the core of the BLAKE hash function, a SHA3 finalist, that had received considerable cryptanalytic attention [NIST-SHA3].

Poly1305 is designed to ensure that forged messages are rejected with a probability of 1-(n/2^102) for a 16*n byte message, even after sending 2^64 legitimate messages.

The cipher suites described in this document require that an nonce is never repeated under the same key. The design presented ensures that by using the TLS sequence number which is unique and does not wrap [RFC5246].

This document should not introduce any other security considerations than those that directly follow from the use of the stream cipher ChaCha20, the AEAD_CHACHA20_POLY1305 construction, and those that directly follow from introducing any set of stream cipher suites into TLS and DTLS.
10. References

10.1. Normative References


10.2. Informative References


Internet-Draft                 chacha-tls                   January 2014


Appendix A. Test vectors

A.1. ChaCha20

The following blocks contain test vectors for ChaCha20. The first line contains the 256-bit key, the second the 64-bit nonce and the last line contains a prefix of the resulting ChaCha20 key-stream.

KEY:       0000000000000000000000000000000000000000000000000000000000000000
NONCE:     0000000000000000
KEYSTREAM: 76b8e0da0f13d90405d6ae55386bd28bddd219b8a08ded1aa836efcc8b770dc7da41597c5157488d7724e03fb8d84a376a43b8f41518a11c387b669b2ee6586

KEY:       0000000000000000000000000000000000000000000000000000000000000001
NONCE:     0000000000000000
KEYSTREAM: 4540f05a9f1fb296d7736e7b208e3c96eb4fe1834688d2604f450952ed432d41bbe2a0b6ea7566d2a5d1e7e20d42af2c53d792b1c43fe817e9ad275ae546963

KEY:       0000000000000000000000000000000000000000000000000000000000000000
NONCE:     0000000000000001
KEYSTREAM: de9ca7bf3d69ef5e786dc63973f653a0b49e015adbf7134fc7b7df137821031e85a050278a7084527214f73efc7fa5b5277062eb7a0433e445f41e3

KEY:       0000000000000000000000000000000000000000000000000000000000000000
NONCE:     0000000000000000
KEYSTREAM: ef3fdefc6c61578eb5cf35bd3dd3bd3b8009631634d21e42ac33960bd138e50d3211e4cafc237ee53ca8ad6426194a88545ddc497a0b466e7d6bbdb0041b2f586b
A.2. Poly1305

The following blocks contain test vectors for Poly1305. The first line contains a variable length input. The second contains the 256-bit key and the last contains the resulting, 128-bit tag.

INPUT: 000000000000000000000000000000000000000000000000000000000000000
KEY: 746869732069732033322d62797465206b657920666f7220506f6c7931333035
TAG: 49ec78090e481ec6c26b33b91ccc0307

INPUT: 48656c6c6f20776f726c6421
KEY: 746869732069732033322d62797465206b657920666f7220506f6c7931333035
TAG: af6f745008f81c916a20dcc74eef2bf0

A.3. AEAD_CHACHA20_POLY1305

The following block contains a test vector for the AEAD_CHACHA20_POLY1305 algorithm. The first four lines consist of the standard inputs to an AEAD algorithm and the last line contains the encrypted result and authenticated tag.

KEY: 4290bcb154173531f314af57f3be3b5006da371ece272afa1b5dbdd1100a1007
INPUT: 86d09974840bded2a5ca
NONCE: cd7cf67be39c794a
AD: 87e229d4500845a079c0
OUTPUT: e3e446f7ede9a19b62a4677dabf4e3d24b876bb284753896e1d6

To aid implementations, the next block contains some intermediate values in the AEAD_CHACHA20_POLY1305 algorithm. The first line
contains the Poly1305 key that is derived and the second contains the raw bytes that are authenticated by Poly1305.

KEY: 9052a6335505b6d507341169783dccac0e26f84ea84906b1558c05bf48150fbe
INPUT: 87e229d4500845a079c00a0000000000000000e3e446f7ede9a19b62a40a000000000000
Authors’ Addresses

Adam Langley
Google Inc
Email: agl@google.com

Wan-Teh Chang
Google Inc
Email: wtc@google.com

Nikos Mavrogiannopoulos
Red Hat
Email: nmav@redhat.com

Joachim Strombergson
Secworks Sweden AB
Email: joachim@secworks.se
URI: http://secworks.se/

Simon Josefsson
SJD AB
Email: simon@josefsson.org
URI: http://josefsson.org/