Workgroup: Network Working Group
Internet-Draft: draft-denis-dprive-dnscrypt-00
Published: 9 March 2023
Intended Status: Informational
Expires: 10 September 2023
Authors: F. Denis
Individual Contributor

The DNSCrypt protocol

Abstract

The DNSCrypt protocol is designed to encrypt and authenticate DNS traffic between clients and resolvers. This document specifies the protocol and its implementation.

About This Document

This note is to be removed before publishing as an RFC.

Status information for this document may be found at https://datatracker.ietf.org/doc/draft-denis-dprive-dnscrypt/.

Source for this draft and an issue tracker can be found at https://github.com/DNSCrypt/dnscrypt-protocol.

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 https://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 10 September 2023.

Table of Contents

1. Introduction
2. Conventions and Definitions
3. Protocol overview
4. Key management
5. Session Establishment
6. Transport
7. Padding for client queries over UDP
8. Client queries over UDP
9. Padding for client queries over TCP
10. Client queries over TCP
11. Authenticated encryption and key exchange algorithm
12. Certificates
13. Security considerations
14. Operational considerations
15. IANA Considerations
16. Appendix 1: The Box-XChaChaPoly algorithm
    - 16.1. HChaCha20
    - 16.2. Test Vector for the HChaCha20 Block Function
    - 16.3. ChaCha20_DJB
    - 16.4. XChaCha20_DJB
    - 16.5. XChaCha20_DJB-Poly1305
    - 16.6. The Box-XChaChaPoly algorithm
17. Normative References

Author's Address

1. Introduction

The document defines a specific protocol, DNSCrypt, that encrypts and authenticates DNS [RFC1035] queries and responses, improving confidentiality, integrity, and resistance to attacks affecting the original DNS protocol.

The protocol is designed to be lightweight, extensible, and simple to implement securely on top of an existing DNS client, server or proxy.

DNS packets don't need to be parsed nor rewritten. DNSCrypt simply wraps them in a secure, encrypted container. Encrypted packets are then exchanged the same way as regular packets, using the standard DNS transport mechanisms. Queries and responses are sent over UDP, falling back to TCP for large responses only if necessary.

DNSCrypt is stateless. Every query can be processed independently from other queries. There are no session identifiers. Clients can replace their keys whenever they want, without extra interactions with servers.

DNSCrypt packets can securely be proxied without having to be decrypted, allowing client IP addresses to be hidden from resolvers ("Anonymized DNSCrypt").

A recursive DNS server can accept DNSCrypt queries on the same IP address and port as regular DNS. Similarly, DNSCrypt and DoH can also share the same IP address and TCP port.

Finally, DNSCrypt addresses two security issues inherent to regular DNS over UDP: amplification and fragment attacks.

2. Conventions and Definitions

The key words "**MUST**", "**MUST NOT**", "**REQUIRED**", "**SHALL**", "**SHALL NOT**", "**SHOULD**", "**SHOULD NOT**", "**RECOMMENDED**", "**NOT RECOMMENDED**", "**MAY**", and "**OPTIONAL**" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

Definitions for client queries:

*<dnscrypt-query>: <client-magic> <client-pk> <client-nonce> <encrypted-query>

*<client-magic>: a 8 byte identifier for the resolver certificate chosen by the client.

*<client-pk>: the client's public key, whose length depends on the encryption algorithm defined in the chosen certificate.

*<client-sk>: the client's secret key.

*<resolver-pk>: the resolver's public key.

*<client-nonce>: a unique query identifier for a given (<client-sk>, <resolver-pk>) tuple. The same query sent twice for the same (<client-sk>, <resolver-pk>) tuple must use two distinct <client-nonce> values. The length of <client-nonce> depends on the chosen encryption algorithm.

*AE: the authenticated encryption function.

*<encrypted-query>: AE(<shared-key> <client-nonce> <client-nonce-pad>, <client-query> <client-query-pad>)

*<shared-key>: the shared key derived from <resolver-pk> and <client-sk>, using the key exchange algorithm defined in the chosen certificate. -<client-query>: the unencrypted client query. The query is not modified; in particular, the query flags are not altered and the query length must be kept in queries prepared to be sent over TCP.

*<client-nonce-pad>: <client-nonce> length is half the nonce length required by the encryption algorithm. In client queries, the other half, <client-nonce-pad> is filled with NUL bytes.

*<client-query-pad>: the variable-length padding.

Definitions for server responses:

*<dnscrypt-response>: <resolver-magic> <nonce> <encrypted-response>

*<resolver-magic>: the 0x72 0x36 0x66 0x6e 0x76 0x57 0x6a 0x38 byte sequence

*<nonce>: <client-nonce> <resolver-nonce>

*<client-nonce>: the nonce sent by the client in the related query.

*<client-pk>: the client's public key.

*<resolver-sk>: the resolver's secret key.

*<resolver-nonce>: a unique response identifier for a given (<client-pk>, <resolver-sk>) tuple. The length of <resolver-nonce> depends on the chosen encryption algorithm.

*DE: the authenticated decryption function.

*<encrypted-response>: DE(<shared-key>, <nonce>, <resolver-response> <resolver-response-pad>)

*<shared-key>: the shared key derived from <resolver-sk> and <client-pk>, using the key exchange algorithm defined in the chosen certificate.

*<resolver-response>: the unencrypted resolver response. The response is not modified; in particular, the query flags are not altered and the response length must be kept in responses prepared to be sent over TCP.

*<resolver-response-pad>: the variable-length padding.

3. Protocol overview

The protocol operates as follows:

1. The DNSCrypt client sends a DNS query to a DNSCrypt server to retrieve the server's public keys.

2. The client generates its own key pair.

3. The client encrypts unmodified DNS queries using a server's public key, padding them as necessary, and concatenates them to a nonce and a copy of the client's public key. The resulting output is sent using standard DNS transport mechanisms.

4. Encrypted queries are decrypted by the server using the attached client public key and the server's own secret key. The output is a regular DNS packet that doesn't require any special processing.

5. To send an encrypted response, the server adds padding to the unmodified response, encrypts the result using the client's public key and the client's nonce, and truncates the response if necessary. The resulting packet, truncated or not, is sent to the client using standard DNS mechanisms.

6. The client authenticates and decrypts the response using its secret key, the server's public key, the attached nonce, and its own nonce. If the response was truncated, the client may adjust internal parameters and retry over TCP. If not, the

output is a regular DNS response that can be directly forwarded to applications and stub resolvers.

4. Key management

Both the client and the resolver initially generate a short-term key pair for each supported encryption system.

The client generates a key pair for each resolver it communicates with, and the resolver generates a key pair for each client it communicates with. The resolver also generates a public key for each supported encryption system.

5. Session Establishment

From a client perspective, a DNSCrypt session begins with the client sending a non-authenticated DNS query to a DNSCrypt-enabled resolver. This DNS query encodes the certificate versions supported by the client, as well as a public identifier of the provider requested by the client.

The resolver responds with a public set of signed certificates that must be verified by the client using a previously distributed public key, known as the provider public key. Each certificate includes a validity period, a serial number, a version that defines a key exchange mechanism, an authenticated encryption algorithm and its parameters, as well as a short-term public key, known as the resolver public key.

A resolver can support multiple algorithms and advertise multiple resolver public keys simultaneously. The client picks the one with the highest serial number among the currently valid ones that match a supported protocol version.

Each certificate includes a magic number that the client must prefix its queries with, in order for the resolver to know what certificate was chosen by the client to construct a given query.

The encryption algorithm, resolver public key, and client magic number from the chosen certificate are then used by the client to send encrypted queries. These queries include the client public key.

Using this client public key, and knowing which certificate was chosen by the client as well as the relevant secret key, the resolver verifies and decrypts the query and encrypts the response using the same parameters.

6. Transport

The DNSCrypt protocol can use the UDP and TCP transport protocols. DNSCrypt Clients and resolvers should support the protocol over UDP and must support it over TCP.

The default port for this protocol should be 443, both for TCP and UDP.

7. Padding for client queries over UDP

Prior to encryption, queries are padded using the ISO/IEC 7816-4 format. The padding starts with a byte valued 0x80 followed by a variable number of NUL bytes.

<client-query> <client-query-pad> must be at least <min-query-len> bytes. If the length of the client query is less than <min-query-len>, the padding length must be adjusted in order to satisfy this requirement.

<min-query-len> is a variable length, initially set to 256 bytes, and must be a multiple of 64 bytes.

8. Client queries over UDP

Client queries sent using UDP must be padded as described in section 3.

A UDP packet can contain a single query, whose entire content is the <dnscrypt-query> construction documented in section 2.

UDP packets using the DNSCrypt protocol can be fragmented into multiple IP packets and can use a single source port.

After having received a query, the resolver can either ignore the query or reply with a DNSCrypt-encapsulated response.

The client must verify and decrypt the response using the resolver's public key, the shared secret and the received nonce. If the response cannot be verified, the response must be discarded.

If the response has the TC flag set, the client must:

1. send the query again using TCP

2. set the new minimum query length as:

<min-query-len> ::= min(<min-query-len> + 64, <max-query-len>)

<min-query-len> must be capped so that the full length of a DNSCrypt packet doesn't exceed the maximum size required by the transport layer.

The client may decrease <min-query-len>, but the length must remain a multiple of 64 bytes.

9. Padding for client queries over TCP

Prior to encryption, queries are padded using the ISO/IEC 7816-4 format. The padding starts with a byte valued 0x80 followed by a variable number of NUL bytes.

The length of <client-query-pad> is randomly chosen between 1 and 256 bytes (including the leading 0x80), but the total length of <client-query> <client-query-pad> must be a multiple of 64 bytes.

For example, an originally unpadded 56-bytes DNS query can be padded as:

<56-bytes-query> 0x80 0x00 0x00 0x00 0x00 0x00 0x00 0x00

or

<56-bytes-query> 0x80 (0x00 * 71)

or

<56-bytes-query> 0x80 (0x00 * 135)

or

<56-bytes-query> 0x80 (0x00 * 199)

10. Client queries over TCP

Encrypted client queries over TCP only differ from queries sent over UDP by the padding length computation and by the fact that they are prefixed with their length, encoded as two big-endian bytes.

Cleartext DNS query payloads are not prefixed by their length, even when sent over TCP.

Unlike UDP queries, a query sent over TCP can be shorter than the response.

After having received a response from the resolver, the client and the resolver must close the TCP connection. Multiple transactions over the same TCP connections are not allowed by this revision of the protocol.

11. Authenticated encryption and key exchange algorithm

The Box-XChaChaPoly construction, and the way to use it described in this section, must be referenced in certificates as version 2 of the public-key authenticated encryption system.

The construction, originally implemented in the libsodium cryptographic library and exposed under the name "crypto_box_curve25519xchacha20poly1305", uses the Curve25119 elliptic curve in Montgomery form and the hchacha20 hash function for key exchange, the XChaCha20 stream cipher, and Poly1305 for message authentication.

The public and secret keys are 32 bytes long in storage. The MAC is 16 bytes long, and is prepended to the ciphertext.

When using Box-XChaChaPoly, this construction requires a 24 bytes nonce, that must not be reused for a given shared secret.

With a 24 bytes nonce, a question sent by a DNSCrypt client must be encrypted using the shared secret, and a nonce constructed as follows: 12 bytes chosen by the client followed by 12 NUL (0x00) bytes.

A response to this question must be encrypted using the shared secret, and a nonce constructed as follows: the bytes originally chosen by the client, followed by bytes chosen by the resolver.

The resolver's half of the nonce should be randomly chosen.

The client's half of the nonce can include a timestamp in addition to a counter or to random bytes, so that when a response is received, the client can use this timestamp to immediately discard responses to queries that have been sent too long ago, or dated in the future.

12. Certificates

The client begins a DNSCrypt session by sending a regular unencrypted TXT DNS query to the resolver IP address, on the DNSCrypt port, first over UDP, then, in case of failure, timeout or truncation, over TCP.

Resolvers are not required to serve certificates both on UDP and TCP.

The name in the question (<provider name) must follow this scheme:

<protocol-major-version> . dnscrypt-cert . <zone>

A major protocol version has only one certificate format.

A DNSCrypt client implementing the second version of the protocol must send a query with the TXT type and a name of the form:

2.dnscrypt-cert.example.com

The zone must be a valid DNS name, but may not be registered in the DNS hierarchy.

A single provider name can be shared by multiple resolvers operated by the same entity, and a resolver can respond to multiple provider names, especially to support multiple protocol versions simultaneously.

In order to use a DNSCrypt-enabled resolver, a client must know the following information:

- The resolver IP address and port
- The provider name
- The provider public key

The provider public key is a long-term key whose sole purpose is to verify the certificates. It is never used to encrypt or verify DNS queries. A unique provider public key can be used to sign multiple certificates.

For example, an organization operating multiple resolvers can use a unique provider name and provider public key across all resolvers, and just provide a list of IP addresses and ports. Each resolver may

have its unique set of certificates that can be signed with the same key.

Certificates should be signed on dedicated hardware and not on the resolvers. Resolvers must serve the certificates, provided that they have already been signed.

A successful response to certificate request contains one or more TXT records, each record containing a certificate encoded as follows:

* <cert>: <cert-magic> <es-version> <protocol-minor-version> <signature> <resolver-pk> <client-magic> <serial> <ts-start> <ts-end> <extensions>

* <cert-magic>: 0x44 0x4e 0x53 0x43

* <es-version>: the cryptographic construction to use with this certificate.

For Box-XChaChaPoly, <es-version> must be 0x00 0x02.

* <protocol-minor-version>: 0x00 0x00

* <signature>: a 64-byte signature of (<resolver-pk> <client-magic> <serial> <ts-start> <ts-end> <extensions>) using the Ed25519 algorithm and the provider secret key. Ed25519 must be used in this version of the protocol.

* <resolver-pk>: the resolver short-term public key, which is 32 bytes when using X25519.

* <client-magic>: the first 8 bytes of a client query that was built using the information from this certificate. It may be a truncated public key. Two valid certificates cannot share the same <client-magic>.

* <client-magic> must not start with 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 (seven all-zero bytes) in order to avoid a confusion with the QUIC protocol.

* <serial>: a 4 byte serial number in big-endian format. If more than one certificates are valid, the client must prefer the certificate with a higher serial number.

* <ts-start>: the date the certificate is valid from, as a big-endian 4-byte unsigned Unix timestamp.

* <ts-end>: the date the certificate is valid until (inclusive), as a big-endian 4-byte unsigned Unix timestamp.

* <extensions>: empty in the current protocol version, but may contain additional data in future revisions, including minor versions. The computation and the verification of the signature must include the extensions. An implementation not supporting these extensions must ignore them.

Certificates made of these information, without extensions, are 116 bytes long. With the addition of the cert-magic, es-version and protocol-minor-version, the record is 124 bytes long.

After having received a set of certificates, the client checks their validity based on the current date, filters out the ones designed for encryption systems that are not supported by the client, and chooses the certificate with the higher serial number.

DNSCrypt queries sent by the client must use the <client-magic> header of the chosen certificate, as well as the specified encryption system and public key.

The client must check for new certificates every hour, and switch to a new certificate if:

*the current certificate is not present or not valid any more

or

*a certificate with a higher serial number than the current one is available.

13. Security considerations

DNSCrypt does not protect against attacks on DNS infrastructure.

14. Operational considerations

Special attention should be paid to the uniqueness of the generated secret keys.

Client public keys can be used by resolvers to authenticate clients, link queries to customer accounts, and unlock business-specific features such as redirecting specific domain names to a sinkhole.

Resolvers accessible from any client IP address can also opt for only responding to a set of whitelisted public keys.

Resolvers accepting queries from any client must accept any client public key. In particular, an anonymous client can generate a new key pair for every session, or even for every query.

his mitigates the ability for a resolver to group queries by client public keys, and discover the set of IP addresses a user might have been operating.

Resolvers must rotate the short-term key pair every 24 hours at most, and must throw away the previous secret key.

After a key rotation, a resolver must still accept all the previous keys that haven't expired.

Provider public keys may be published as a DNSSEC-signed TXT records, in the same zone as the provider name.

For example, a query for the TXT type on the name "2.pubkey.example.com" may return a signed record containing a

hexadecimal-encoded provider public key for the provider name "2.dnscrypt-cert.example.com".

As a client is likely to reuse the same key pair many times, servers are encouraged to cache shared keys instead of performing the X25519 operation for each query. This makes the computational overhead of DNSCrypt negligible compared to plain DNS.

15. IANA Considerations

This document has no IANA actions.

16. Appendix 1: The Box-XChaChaPoly algorithm

The Box-XChaChaPoly algorithm combines the X25519 [RFC7748] key exchange mechanism with a variant of the ChaCha20-Poly1305 constrution defined in [RFC8439].

16.1. HChaCha20

HChaCha20 is an intermediary step based on the construction and security proof used to create XSalsa20, an extended-nonce Salsa20 variant.

HChaCha20 is initialized the same way as the ChaCha20 cipher defined in [RFC8439], except that HChaCha20 uses a 128-bit nonce and has no counter. Instead, the block counter is replaced by the first 32 bits of the nonce.

Consider the two figures below, where each non-whitespace character represents one nibble of information about the ChaCha states (all numbers little-endian):

```
               cccccccc  cccccccc  cccccccc  cccccccc
               kkkkkkkk  kkkkkkkk  kkkkkkkk  kkkkkkkk
               kkkkkkkk  kkkkkkkk  kkkkkkkk  kkkkkkkk
               bbbbbbbb  nnnnnnnn  nnnnnnnn  nnnnnnnn

       ChaCha20 State: c=constant k=key b=blockcount n=nonce


               cccccccc  cccccccc  cccccccc  cccccccc
               kkkkkkkk  kkkkkkkk  kkkkkkkk  kkkkkkkk
               kkkkkkkk  kkkkkkkk  kkkkkkkk  kkkkkkkk
               nnnnnnnn  nnnnnnnn  nnnnnnnn  nnnnnnnn

              HChaCha20 State: c=constant k=key n=nonce
```

After initialization, proceed through the ChaCha rounds as usual.

Once the 20 ChaCha rounds have been completed, the first 128 bits and last 128 bits of the ChaCha state (both little-endian) are concatenated, and this 256-bit subkey is returned.

16.2. Test Vector for the HChaCha20 Block Function

- Key = 00:01:02:03:04:05:06:07:08:09:0a:0b:0c:0d:0e:0f:10:11:12:13:14:15:16:17:18:19:1a:1b:1c:1d:1e:1f. The key is a sequence of octets with no particular structure before we copy it into the HChaCha state.

- Nonce = (00:00:00:09:00:00:00:4a:00:00:00:00:31:41:59:27)

After setting up the HChaCha state, it looks like this:

```
61707865 3320646e 79622d32 6b206574
03020100 07060504 0b0a0908 0f0e0d0c
13121110 17161514 1b1a1918 1f1e1d1c
09000000 4a000000 00000000 27594131
```

ChaCha state with the key setup.

After running 20 rounds (10 column rounds interleaved with 10 "diagonal rounds"), the HChaCha state looks like this:

```
423b4182 fe7bb227 50420ed3 737d878a
0aa76448 7954cdf3 846acd37 7b3c58ad
77e35583 83e77c12 e0076a2d bc6cd0e5
d5e4f9a0 53a8748a 13c42ec1 dcecd326
```

HChaCha state after 20 rounds

HChaCha20 will then return only the first and last rows, in little endian, resulting in the following 256-bit key:

```
82413b42 27b27bfe d30e4250 8a877d73
a0f9e4d5 8a74a853 c12ec413 26d3ecdc
```

Resultant HChaCha20 subkey

16.3. ChaCha20_DJB

ChaCha20 was originally designed to have a 8 byte nonce.

For the needs of TLS, [RFC8439] changed this to set N_MIN and N_MAX to 12, at the expense of a smaller internal counter.

DNSCrypt uses ChaCha20 as originally specified, with N_MIN = N_MAX = 8.

We refer to this variant as ChaCha20_DJB.

Common implementations may just refer to it as ChaCha20 and the IETF version as ChaCha20-IETF.

The internal counter in ChaCha20_DJB is 4 bytes larger than ChaCha20. There are no other differences between ChaCha20_DJB and ChaCha20.

16.4. XChaCha20_DJB

XChaCha20_DJB can be constructed from ChaCha20 implementation and HChaCha20.

All one needs to do is:

1. Pass the key and the first 16 bytes of the 24-byte nonce to HChaCha20 to obtain the subkey.

2. Use the subkey and remaining 8 byte nonce with ChaCha20_DJB.

16.5. XChaCha20_DJB-Poly1305

XChaCha20 is a stream cipher and offers no integrity guarantees without being combined with a MAC algorithm (e.g. Poly1305).

XChaCha20_DJB-Poly1305 adds an authentication tag to ciphertext encrypted with XChaCha20_DJB.

The Poly1305 key is computed as in [RFC8439], by encrypting an empty block.

Finally, the output of the Poly1305 function is prepended to the ciphertext:

*<k>: encryption key

*<m>: message to encrypt

*XChaCha20_DJB-Poly1305(<k>, <m>): Poly1305(XChaCha20_DJB(<k>, <m>)) || XChaCha20_DJB(<k>, <m>)

16.6. The Box-XChaChaPoly algorithm

The Box-XChaChaPoly algorithm combines the key exchange mechanism X25519 defined [RFC7748] with the XChaCha20_DJB-Poly1305 authenticated encryption algorithm.

*<k>: encryption key

*<m>: message to encrypt

*<pk>: recipent's public key

*<sk>: sender's secret key

*sk: HChaCha20(X25519(<pk>, <sk>))

*Box-XChaChaPoly(pk, sk, m): XChaCha20_DJB-Poly1305(<sk>, <m>)

17. Normative References

[RFC1035] Mockapetris, P., "Domain names - implementation and specification", STD 13, RFC 1035, DOI 10.17487/RFC1035, November 1987, <https://www.rfc-editor.org/info/rfc1035>.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/info/rfc2119>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/info/rfc8174>.

[RFC7748] Langley, A., Hamburg, M., and S. Turner, "Elliptic Curves for Security", RFC 7748, DOI 10.17487/RFC7748, January 2016, <https://www.rfc-editor.org/info/rfc7748>.

[RFC8439] Nir, Y. and A. Langley, "ChaCha20 and Poly1305 for IETF Protocols", RFC 8439, DOI 10.17487/RFC8439, June 2018, <https://www.rfc-editor.org/info/rfc8439>.

Author's Address

Frank Denis
Individual Contributor

Email: fde@00f.net