Network Working Group
Internet-Draft
Intended status: Informational
Expires: 30 September 2026

F. Denis
Individual Contributor

29 March 2026

The DNSCrypt Protocol
draft-denis-dprive-dnscrypt-08

Abstract

The DNSCrypt protocol is designed to encrypt and authenticate DNS traffic between clients and resolvers. This document specifies the protocol and its implementation, providing a standardized approach to securing DNS communications. DNSCrypt improves confidentiality, integrity, and resistance to attacks affecting the original DNS protocol while maintaining compatibility with existing DNS infrastructure.

About This Document

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

The latest revision of this draft can be found at https://dnscrypt.github.io/dnscrypt-protocol/. 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 30 September 2026.

Table of Contents

1. Introduction

The Domain Name System (DNS) [RFC1035] is a critical component of Internet infrastructure, but its original design did not include security features to protect the confidentiality and integrity of queries and responses. This fundamental security gap exposes DNS traffic to eavesdropping, tampering, and various attacks that can compromise user privacy and network security.

To address these vulnerabilities, this document defines the DNSCrypt protocol, which encrypts and authenticates DNS queries and responses, providing strong 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 existing DNS infrastructure, offering a practical solution for securing DNS communications without requiring significant changes to current systems.

The following sections detail the protocol's design, starting with an overview of its operation and then progressing through the technical specifications needed for implementation.

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.

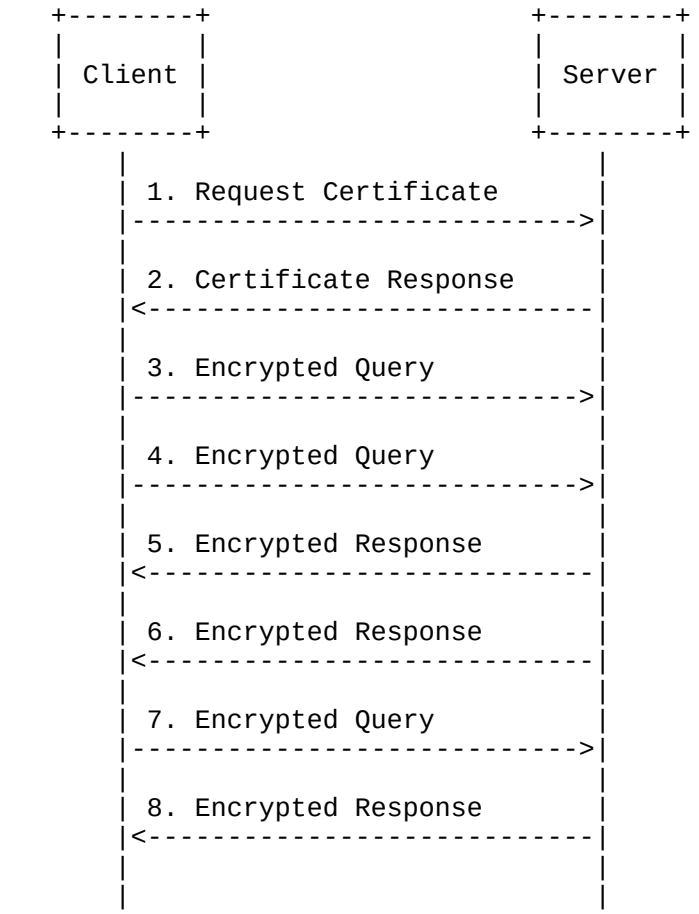
3. Protocol Flow

The DNSCrypt protocol consists of two distinct phases:

1. *Initial Setup Phase* (one-time):
    * The client requests the server's certificate
    * The server responds with its certificate containing public keys
2. *Ongoing Communication Phase* (repeated as needed):
    * The client sends encrypted DNS queries

* The server responds with encrypted DNS responses

The following diagram illustrates the complete protocol flow:



The initial setup phase (steps 1-2) occurs only when:

* A client first starts using a DNSCrypt server

* The client’s cached certificate expires

* The client detects a certificate with a higher serial number

After the initial setup, the client and server engage in the ongoing communication phase (steps 3-8), where encrypted queries and responses are exchanged as needed. This phase can be repeated indefinitely until the certificate expires or a new certificate is available.

The ongoing communication phase operates with several important characteristics that distinguish it from traditional DNS:

1. *Stateless Operation*: Each query and response is independent. The server does not maintain state between queries.

2. *Out-of-Order Responses*: Responses may arrive in a different order than the queries were sent. Each response is self-contained and can be processed independently.

3. *Concurrent Queries*: A client can send multiple queries without waiting for earlier responses, and responses can be processed independently as they arrive.

With this understanding of the protocol flow, we can now examine the specific components that make up DNSCrypt packets and their structure.

4. Protocol Components

The DNSCrypt protocol defines specific packet structures for both client queries and server responses. These components work together to provide the security properties described in the previous section.

Definitions for client queries:

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

* <client-magic>: an 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> is determined by 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.

* <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.

* AE: the authenticated encryption function.

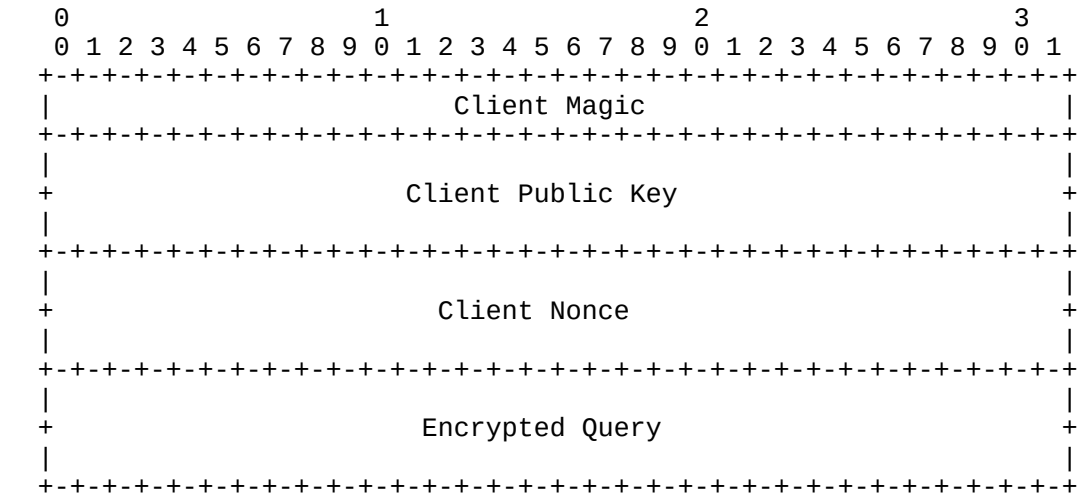
* <encrypted-response>: AE(<shared-key>, <nonce>, <resolver-response> <resolver-response-pad>)

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

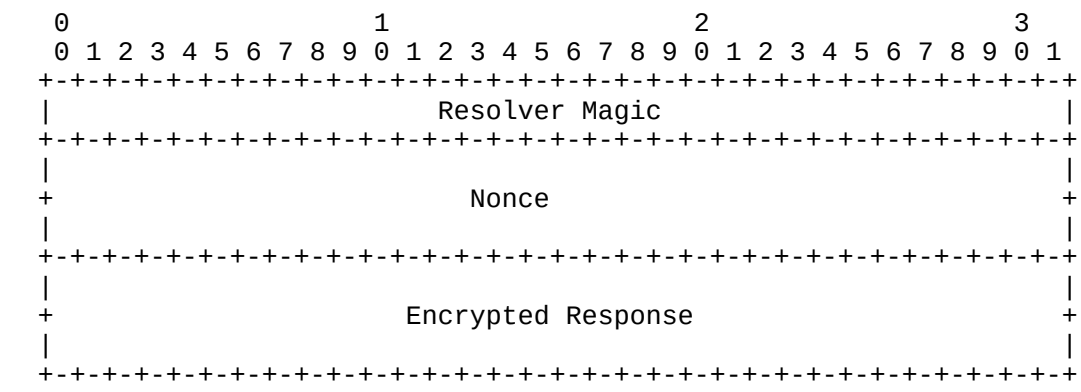
* <resolver-response>: the unencrypted resolver response. The response is not modified; in particular, the query flags are not altered.

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

The following diagram shows the structure of a DNSCrypt query packet:



The following diagram shows the structure of a DNSCrypt response packet:



These packet structures form the foundation for the protocol operations described in the next section, which details how clients and servers use these components to establish secure communications.

5. Protocol Description

5.1. Overview

Building on the protocol flow and components described earlier, this section provides a detailed examination of how the DNSCrypt protocol operates. The protocol follows a well-defined sequence of steps:

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 transmitted to the server via standard DNS transport mechanisms [RFC1035].

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 shared key and a nonce made of the client's nonce followed by the resolver'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 the shared key and the nonce included in the response. If the response was truncated, the client MAY adjust internal parameters and retry over TCP [RFC7766]. If not, the output is a regular DNS response that can be directly forwarded to applications and stub resolvers.

Key features of the DNSCrypt protocol include:

* Stateless operation: Every query can be processed independently from other queries, with no session identifiers required.

* Flexible key management: Clients can replace their keys whenever they want, without extra interactions with servers.

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

* Shared infrastructure: Recursive DNS servers can accept DNSCrypt queries on the same IP address and port used for regular DNS traffic.

* Attack mitigation: DNSCrypt mitigates two common security vulnerabilities in regular DNS over UDP: amplification [RFC5358] and fragmentation attacks.

These key features enable DNSCrypt to provide robust security while maintaining practical deployability. The protocol's transport characteristics further support these goals.

5.2. Transport

The DNSCrypt protocol can use the UDP and TCP transport protocols. DNSCrypt clients and resolvers SHOULD support the protocol via UDP, and MUST support it over TCP.

Both TCP and UDP connections using DNSCrypt SHOULD employ port 443 by default.

The choice of port 443 helps DNSCrypt traffic blend with HTTPS traffic, providing some protection against traffic analysis. Once transport is established, the next step is session establishment through certificate exchange.

5.3. Session Establishment

From the client's perspective, a DNSCrypt session is initiated when the client sends an unauthenticated DNS query to a DNSCrypt-capable resolver. This DNS query contains encoded information about the certificate versions supported by the client and a public identifier of the desired provider.

The resolver sends back a collection of signed certificates that the client MUST verify using the pre-distributed 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.

Resolvers have the ability to support various algorithms and can concurrently advertise multiple short-term public keys (resolver public keys). The client picks the one with the highest serial number among the currently valid ones that match a supported protocol version.

Every certificate contains a unique magic number that the client MUST include at the beginning of their queries. This allows the resolver to identify which certificate the client selected for crafting a particular 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.

With the knowledge of the chosen certificate and corresponding secret key, along with the client's public key, the resolver is able to verify, decrypt the query, and then encrypt the response utilizing identical parameters.

Once the session is established through certificate exchange, the ongoing query processing follows specific rules for different transport protocols and padding requirements.

5.4. Query Processing

5.4.1. Padding For Client Queries Over UDP

Before encryption takes place, queries are padded according to the ISO/IEC 7816-4 standard. Padding begins with a single byte holding the value 0x80, followed by any number of NUL bytes.

<client-query> <client-query-pad> MUST be at least <min-query-len> bytes. In this context, <client-query> represents the original client query, while <client-query-pad> denotes the added padding.

Should the client query's length fall short of <min-query-len> bytes, the padding length MUST be adjusted in order to satisfy the length requirement.

<min-query-len> is a variable length, initially set to 256 bytes, and MUST be a multiple of 64 bytes. It represents the minimum permitted length for a client query, inclusive of padding.

5.4.2. Client Queries Over UDP

UDP-based client queries need to follow the padding guidelines outlined in the previous section.

Each UDP packet MUST hold one query, with the complete content comprising the <dnscrypt-query> structure specified in the Protocol Components section.

UDP packets employing the DNSCrypt protocol have the capability to be split into distinct IP packets sharing the same source port.

Upon receiving a query, the resolver may choose to either disregard it or send back a response encrypted using DNSCrypt.

The client MUST authenticate and, if authentication succeeds, decrypt the response with the help of the resolver's public key, the shared secret, and the obtained nonce. In case the response fails verification, it MUST be disregarded by the client.

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

1. send the query again using TCP [RFC7766]

2. set the new minimum query length as:

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

<min-query-len> denotes the minimum permitted length for a client query, including padding. That value 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.

While UDP queries require careful length management due to truncation concerns, TCP queries follow different padding rules due to the reliable nature of the transport.

5.4.3. Padding For Client Queries Over TCP

Queries MUST undergo padding using the ISO/IEC 7816-4 format before being encrypted. The padding starts with a byte valued 0x80 followed by a variable number of NUL bytes.

The length of <client-query-pad> is selected randomly, ranging from 1 to 256 bytes, including the initial byte valued at 0x80. 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)

5.4.4. Client Queries Over TCP

The sole differences between encrypted client queries transmitted via TCP and those sent using UDP lie in the padding length calculation and the inclusion of a two-byte big-endian length prefix for the encrypted DNSCrypt packet.

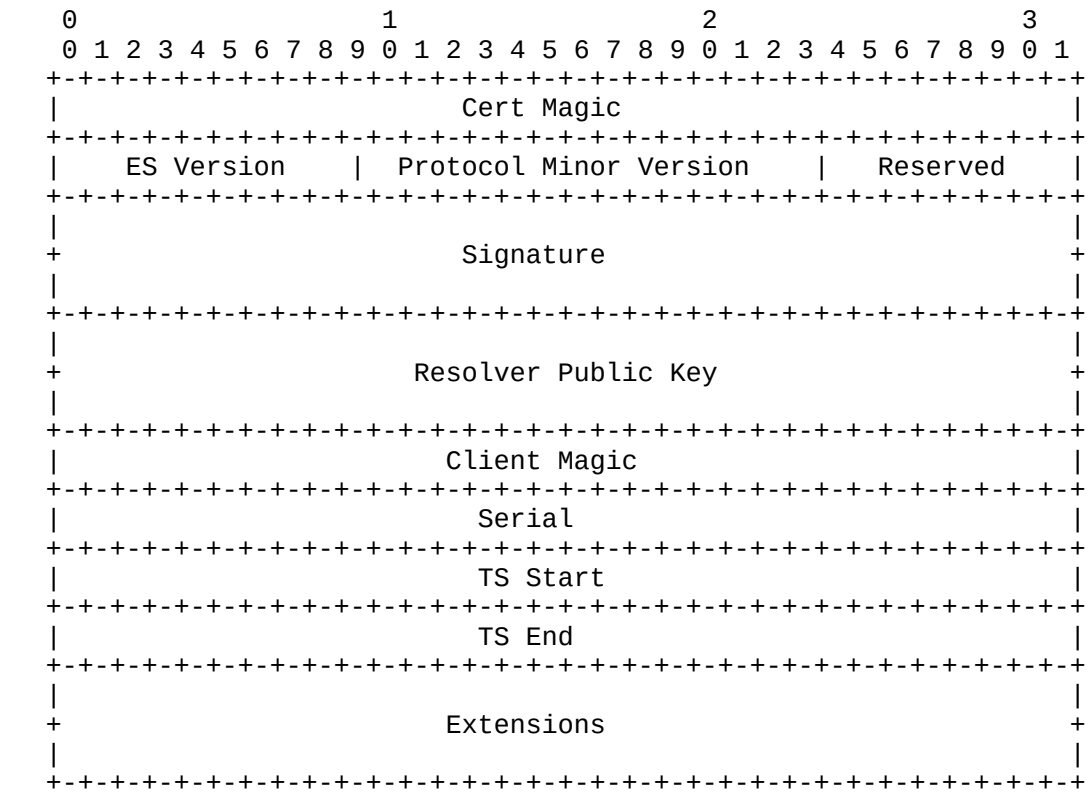
Unlike UDP queries, a query sent over TCP can be shorter than the response.

After having received a response from the resolver, the client and the resolver MUST close the TCP connection to ensure security and comply with this revision of the protocol, which prohibits multiple transactions over the same TCP connection.

The query processing rules described above depend on the certificate information obtained during session establishment. The certificate format and management procedures are critical to the protocol’s security.

5.5. Certificates

The following diagram shows the structure of a DNSCrypt certificate:



To initiate a DNSCrypt session, a client transmits an ordinary unencrypted TXT DNS query to the resolver's IP address and DNSCrypt port. The attempt is first made using UDP; if unsuccessful due to failure, timeout, or truncation, the client then proceeds with 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 single provider public key can be employed 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.

It is RECOMMENDED that certificates are signed using specialized hardware rather than directly on the resolvers themselves. Once signed, resolvers SHOULD make these certificates available to clients. Signing certificates on dedicated hardware helps ensure security and integrity, as it isolates the process from potential vulnerabilities present in the resolver’s system.

A successful response to a 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 confusion with the QUIC protocol [RFC9000].

* <serial>: a 4-byte serial number in big-endian format. If more than one certificate is 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 verification of the signature MUST include the extensions. An implementation not supporting these extensions MUST ignore them.

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

After receiving 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 anymore,

or

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

The certificate management system ensures that cryptographic keys remain fresh and that clients can smoothly transition to updated certificates. With the core protocol mechanics now established, we can examine implementation considerations.

6. Implementation Status

Note: This section is to be removed before publishing as an RFC.

Multiple implementations of the protocol described in this document have been developed and verified for interoperability. A comprehensive list of known implementations can be found at https://dnscrypt.info/implementations.

The successful deployment of multiple interoperable implementations demonstrates the protocol’s maturity. However, proper implementation requires careful attention to security considerations.

7. Security Considerations

This section discusses security considerations for the DNSCrypt protocol.

7.1. Protocol Security

The DNSCrypt protocol provides several security benefits:

1. *Confidentiality*: DNS queries and responses are encrypted using XChaCha20-Poly1305 [RFC8439], preventing eavesdropping of DNS traffic. For example, a query for "example.com" would be encrypted and appear as random data to an observer.

2. *Integrity*: Message authentication using Poly1305 [RFC8439] ensures that responses cannot be tampered with in transit. Any modification to the encrypted response would be detected and rejected by the client.

3. *Authentication*: The use of X25519 [RFC7748] for key exchange and Ed25519 for certificate signatures provides strong authentication of resolvers. Clients can verify they are communicating with the intended resolver and not an impostor.

4. *Short-Term Resolver Keys*: Resolver certificates carry short-term public keys, limiting the impact of key compromise and enabling regular key rotation.

These fundamental security properties depend on correct implementation practices. Several implementation-specific security aspects require particular attention.

7.2. Implementation Security

Implementations should consider the following security aspects:

1. *Key Management*:
    * Resolvers MUST rotate their short-term key pairs at most every 24 hours
    * Previous secret keys MUST be discarded after rotation
    * Provider secret keys used for certificate signing SHOULD be stored in hardware security modules (HSMs)
    * Example: A resolver might generate new key pairs daily at midnight UTC
2. *Nonce Management*:
    * Nonces MUST NOT be reused for a given shared secret
    * Clients can include timestamps in their nonces in order to quickly discard stale responses
    * Example: A nonce might be constructed as: timestamp || random_bytes
3. *Padding*:
    * Implementations MUST use the specified padding scheme to prevent traffic analysis
    * The minimum query length SHOULD be adjusted based on network conditions
    * Example: A 50-byte query might be padded to 256 bytes to prevent size-based fingerprinting
4. *Certificate Validation*:

* Clients MUST verify certificate signatures using the provider's public key

* Certificates MUST be checked for validity periods

* Clients MUST prefer certificates with higher serial numbers

* Example: A client might cache valid certificates and check for updates hourly

Proper implementation of these security measures provides the foundation for the protocol's attack mitigation capabilities.

7.3. Attack Mitigation

DNSCrypt provides protection against several types of attacks:

1. *DNS Spoofing*: The use of authenticated encryption prevents spoofed responses. An attacker cannot forge responses without the server's secret key.

2. *Amplification Attacks*: The padding requirements and minimum query length help prevent amplification attacks [RFC5358]. For example, a 256-byte minimum query size limits the amplification factor.

3. *Fragmentation Attacks*: The protocol mitigates fragmentation risks by padding queries, allowing truncated UDP responses, and retrying over TCP when necessary.

4. *Replay Exposure Reduction*: Nonces make responses query-specific, and clients can use timestamps in client nonces to quickly discard stale responses.

While DNSCrypt effectively mitigates these attacks, implementers should also be aware of privacy considerations that extend beyond basic protocol security.

7.4. Privacy Considerations

While DNSCrypt encrypts DNS traffic, there are some privacy considerations:

1. *Resolver Knowledge*: Resolvers can still see the client's IP address unless Anonymized DNSCrypt is used. This can reveal the client's location and network.

2. *Query Patterns*: Even with encryption, the size and timing of queries may reveal information. Padding helps mitigate this but doesn't eliminate it completely.

3. *Certificate Requests*: Initial certificate requests are unencrypted and may reveal client capabilities. This is a one-time exposure per session.

These privacy considerations complement the security measures and should inform operational practices for DNSCrypt deployments.

7.5. Operational Security

Operators should consider:

1. *Key Distribution*: Provider public keys should be distributed securely to clients. This might involve:
   * Publishing keys on secure websites
   * Using DNSSEC-signed records
   * Including keys in software distributions
2. *Certificate Management*: Certificates should be signed on dedicated hardware, not on resolvers. This provides:
   * Better key protection
   * Centralized certificate management
   * Reduced attack surface
3. *Access Control*: Resolvers may implement access control based on client public keys. This can:
   * Prevent abuse
   * Enable service differentiation
   * Support business models
4. *Monitoring*: Operators should monitor for unusual patterns that may indicate attacks:
   * High query rates from single clients
   * Unusual query patterns

* Certificate request anomalies

These operational security practices work together with the technical security measures to provide comprehensive protection. Additional operational considerations extend beyond security to include practical deployment aspects.

8. 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. This 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 DNSSEC-signed TXT records [RFC1035], 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.

While DNSCrypt provides strong encryption and authentication, some use cases require additional privacy protection. The Anonymized DNSCrypt extension addresses scenarios where hiding client IP addresses from resolvers is necessary.

9. Anonymized DNSCrypt

While DNSCrypt encrypts DNS traffic, DNS server operators can still observe client IP addresses. Anonymized DNSCrypt is an extension to the DNSCrypt protocol that allows queries and responses to be relayed by an intermediate server, hiding the client's IP address from the resolver.

This extension maintains all the security properties of standard DNSCrypt while adding an additional layer of privacy protection.

9.1. Protocol Overview

Anonymized DNSCrypt works by having the client send encrypted queries to a relay server, which then forwards them to the actual DNSCrypt resolver. The relay server cannot decrypt the queries or responses, and the resolver only sees the relay's IP address.

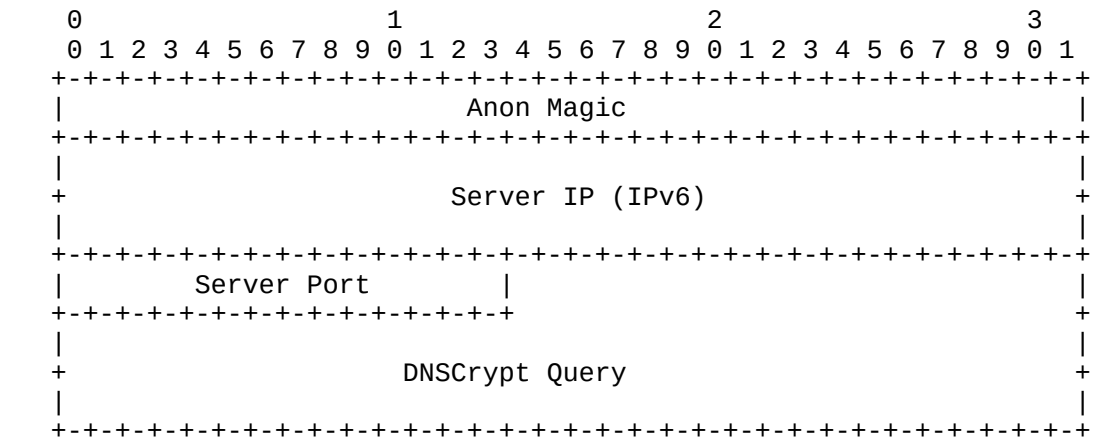
```
[Client]----(encrypted query)--->[Relay]----(encrypted query)--->[Server]

[Client]<--(encrypted response)--[Relay]<--(encrypted response)--[Server]
```

Key properties of Anonymized DNSCrypt:

* The relay cannot decrypt or modify queries and responses
* The resolver only sees the relay's IP address, not the client's
* A DNSCrypt server can simultaneously act as a relay
* The protocol works over both UDP and TCP

9.2. Client Queries

The following diagram shows the structure of an Anonymized DNSCrypt query packet:



An Anonymized DNSCrypt query is a standard DNSCrypt query prefixed with information about the target server:

```
<anondnscrypt-query> ::= <anon-magic> <server-ip> <server-port> <dnscrypt-query>
```

Where:

* <anon-magic>: 0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xff 0x00 0x00

* <server-ip>: 16 bytes encoded IPv6 address (IPv4 addresses are mapped to IPv6 using ::ffff:<ipv4 address> [RFC4291])

* <server-port>: 2 bytes in big-endian format

* <dnscrypt-query>: standard DNSCrypt query

For example, a query for a server at 192.0.2.1:443 would be prefixed with:

```
0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xff 0x00 0x00
0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0xff 0xff 0xc0 0x00 0x02 0x01 0x01 0xbb
```

9.3. Relay Behavior

Relays MUST:

1. Accept queries over both TCP and UDP

2. Communicate with upstream servers over UDP, even if client queries were sent over TCP

3. Validate incoming packets:

    * Check that the target IP is not in a private range [RFC1918]

    * Verify the port number is in an allowed range

    * Ensure the DNSCrypt query doesn’t start with <anon-magic>

    * Verify the query doesn’t start with 7 zero bytes (to avoid confusion with QUIC [RFC9000])

4. Forward valid queries unmodified to the server

5. Verify server responses:

    * Check that the response is smaller than the query

    * Validate the response format (either starts with resolver magic or is a certificate response)

    * Forward valid responses unmodified to the client

These relay requirements ensure that anonymization does not compromise the security properties of the underlying DNSCrypt protocol. Proper deployment requires additional operational considerations.

9.4. Operational Considerations

When using Anonymized DNSCrypt:

1. Clients should choose relays and servers operated by different entities

2. Having relays and servers on different networks is recommended

3. Relay operators should:

    * Refuse forwarding to reserved IP ranges [RFC1918]

    * Restrict allowed server ports (typically only allowing port 443)

    * Monitor for abuse

These operational guidelines help ensure that Anonymized DNSCrypt deployments provide the intended privacy benefits while maintaining security and preventing abuse.

10. IANA Considerations

This document has no IANA actions.

11. Appendix 1: The Box-XChaChaPoly Algorithm

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

11.1. Conventions and Definitions

* x[a..]: the subarray of x starting at index a, and extending to the last index of x

* x[a..b]: the subarray of x starting at index a and ending at index b.

* LOAD32_LE(p): returns a 32-bit unsigned integer from the 4-byte array p

* STORE32_LE(p, x): stores the 32-bit unsigned integer x into the 4-byte array p

11.2. HChaCha20

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

The HChaCha20 function takes the following input parameters:

* <k>: secret key

* <in>: a 128-bit input

and returns a 256-bit keyed hash.

The function can be implemented using an existing IETF-compliant ChaCha20 implementation as follows:

```
   block_bytes = ChaCha20(msg={0}**64, nonce=in[4..16],
                          counter=LOAD32_LE(in[0..4]), key=k)

   block_out[0] = LOAD32_LE(block_bytes[ 0..][0..4]) - 0x61707865
   block_out[1] = LOAD32_LE(block_bytes[ 4..][0..4]) - 0x3320646e
   block_out[2] = LOAD32_LE(block_bytes[ 8..][0..4]) - 0x79622d32
   block_out[3] = LOAD32_LE(block_bytes[12..][0..4]) - 0x6b206574
   block_out[4] =
      LOAD32_LE(block_bytes[48..][0..4]) - LOAD32_LE(in[ 0..][0..4])
   block_out[5] =
      LOAD32_LE(block_bytes[52..][0..4]) - LOAD32_LE(in[ 4..][0..4])
   block_out[6] =
      LOAD32_LE(block_bytes[56..][0..4]) - LOAD32_LE(in[ 8..][0..4])
   block_out[7] =
      LOAD32_LE(block_bytes[60..][0..4]) - LOAD32_LE(in[12..][0..4])

   for i in 0..8:
       STORE32_LE(out[i * 4..][0..4], block_out[i])

   return out
```

11.3. Test Vector For The HChaCha20 Block Function

```
   k:    000102030405060708090a0b0c0d0e0f101112131415161718191a1b1c1d1e1f

   in:   000102030405060708090a0b0c0d0e0f

   out:  51e3ff45a895675c4b33b46c64f4a9ace110d34df6a2ceab486372bacbd3eff6
```

11.4. ChaCha20_DJB

As opposed to the version standardized for IETF protocols, ChaCha20 was originally designed to have a 8 byte nonce.

For the needs of TLS, [RFC8439] changed this by setting 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.

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

11.5. XChaCha20_DJB

XChaCha20_DJB can be constructed from an existing ChaCha20 implementation and the HChaCha20 function.

All that needs to be done 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.

11.6. 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 the 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

* <ct>: XChaCha20_DJB(<k>, <m>)

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

11.7. 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>: recipient’s public key

* <sk>: sender’s secret key

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

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

12. 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>.

[RFC7766] Dickinson, J., Dickinson, S., Bellis, R., Mankin, A., and D. Wessels, "DNS Transport over TCP - Implementation Requirements", RFC 7766, DOI 10.17487/RFC7766, March 2016, <https://www.rfc-editor.org/info/rfc7766>.

[RFC5358] Damas, J. and F. Neves, "Preventing Use of Recursive Nameservers in Reflector Attacks", BCP 140, RFC 5358, DOI 10.17487/RFC5358, October 2008, <https://www.rfc-editor.org/info/rfc5358>.

[RFC9000] Iyengar, J., Ed. and M. Thomson, Ed., "QUIC: A UDP-Based Multiplexed and Secure Transport", RFC 9000, DOI 10.17487/RFC9000, May 2021, <https://www.rfc-editor.org/info/rfc9000>.

[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>.

[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>.

[RFC4291] Hinden, R. and S. Deering, "IP Version 6 Addressing Architecture", RFC 4291, DOI 10.17487/RFC4291, February 2006, <https://www.rfc-editor.org/info/rfc4291>.

[RFC1918] Rekhter, Y., Moskowitz, B., Karrenberg, D., de Groot, G. J., and E. Lear, "Address Allocation for Private Internets", BCP 5, RFC 1918, DOI 10.17487/RFC1918, February 1996, <https://www.rfc-editor.org/info/rfc1918>.

Author’s Address

Frank Denis
Individual Contributor
Email: fde@00f.net