Workgroup: Independent Stream
Internet-Draft: draft-schanzen-gns-21
Published: 7 August 2022
Intended Status: Informational
Expires: 8 February 2023
Authors: M. Schanzenbach
Fraunhofer AISEC
C. Grothoff
Berner Fachhochschule
B. Fix
GNUnet e.V.

The GNU Name System

Abstract

This document contains the GNU Name System (GNS) technical specification. GNS is a decentralized and censorship-resistant domain name resolution protocol that provides a privacy-enhancing alternative to the Domain Name System (DNS) protocols.

This document defines the normative wire format of resource records, resolution processes, cryptographic routines and security considerations for use by implementers.

This specification was developed outside the IETF and does not have IETF consensus. It is published here to inform readers about the function of GNS, guide future GNS implementations, and ensure interoperability among implementations including with the pre-existing GNUnet implementation.

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 8 February 2023.

Table of Contents

1. Introduction

This specification describes the GNU Name System (GNS), a censorship-resistant, privacy-preserving and decentralized domain name resolution protocol. GNS can bind names to any kind of cryptographically secured token, enabling it to double in some respects as an alternative to some of today's public key infrastructures.

In the terminology of the Domain Name System (DNS) [RFC1035], GNS roughly follows the idea of a local root zone deployment (see [RFC8806]), with the difference that the design encourages alternative roots and does not expect all deployments use the same or any specific root zone. In the GNS reference implementation, users can autonomously and freely delegate control of names to zones through their local configurations.

Name resolution and zone dissemination is based on the principle of a petname system where users can assign local names to zones. The GNS has its roots in ideas from the Simple Distributed Security Infrastructure [SDSI], enabling the decentralized mapping of secure identifiers to memorable names. A first academic description of the cryptographic ideas behind GNS can be found in [GNS].

This document defines the normative wire format of resource records, resolution processes, cryptographic routines and security considerations for use by implementers.

This specification was developed outside the IETF and does not have IETF consensus. It is published here to guide implementers of GNS and to ensure interoperability among implementations.

1.1. Requirements Notation

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.

2. Terminology

Apex Label This type of label is used to publish resource records in a zone that can be resolved without providing a specific label. It is the GNS method to provide what is the "zone apex" in DNS [RFC4033]. The apex label is represented using the character U+0040 ("@" without the quotes).

Application A component which uses a GNS implementation to resolve names into records and processes its contents.

Blinded Zone Key The key derived from a zone key and a label. The zone key and the blinded zone key are unlinkable without knowledge of the label.

Extension Label The primary use for the extension label is in redirections where the redirection target is defined relative to the authoritative zone of the redirection record (Section 5.2). The extension label is represented using the character U+002B ("+" without the quotes).

Label Separator Labels in a name are separated using the label separator U+002E ("." without the quotes). In GNS, with the exceptions of zone Top-Level Domains (see below) and boxed records (see Section 5.3.3), every separator label in a name delegates to another zone.

Label A GNS label is a label as defined in [RFC8499]. Labels are UTF-8 strings in Unicode Normalization Form C (NFC) [Unicode-UAX15]. The apex label, label separator and the extension label have special purposes in the resolution protocol which are defined in the rest of the document. Zone administrators **MAY** disallow certain labels that might be easily confused with other labels through registration policies (see also Section 9.4).

Name
: A name in GNS is a domain name as defined in [RFC8499] as an ordered list of labels. Names are UTF-8 [RFC3629] strings consisting of the list of labels concatenated with a label separator. Names are resolved starting from the rightmost label. GNS does not impose length restrictions on names or labels. However, applications **MAY** ensure that name and label lengths are compatible with DNS and in particular IDNA [RFC5890]. In the spirit of [RFC5895], applications **MAY** preprocess names and labels to ensure compatibility with DNS or support specific user expectations, for example according to [Unicode-UTS46]. A GNS name may be indistinguishable from a DNS name and care must be taken by applications and implementors when handling GNS names (see Section 9.10).

Resolver The component of a GNS implementation which provides the recursive name resolution logic defined in Section 7.

Resource Record A GNS resource record is the information associated with a label in a GNS zone. A GNS resource record contains information as defined by its resource record type.

Start Zone In order to resolve any given GNS name an initial start zone must be determined for this name. The start zone can be explicitly defined through a zTLD. Otherwise, it is determined through a local suffix-to-zone mapping (see Section 7.1).

Top-Level Domain The rightmost part of a GNS name is a GNS Top-Level Domain (TLD). A GNS TLD can consist of one or more labels. Unlike DNS Top-Level Domains (defined in [RFC8499]), GNS does not expect all users to use the same global root zone. Instead, with the exception of Zone Top-Level Domains (see below), GNS TLDs are typically part of the configuration of the local resolver (see Section 7.1), and might thus not be globally unique.

Zone A GNS zone contains authoritative information (resource records). A zone is uniquely identified by its zone key. Unlike

DNS zones, a GNS zone does not need to have a SOA record under the apex label.

Zone Key A key which uniquely identifies a zone. It is usually a public key of an asymmetric key pair.

Zone Key Derivation Function The zone key derivation function (ZKDF) blinds a zone key using a label.

Zone Master The component of a GNS implementation which provides local zone management and publication as defined in Section 6.

Zone Owner The holder of the secret (typically a private key) that (together with a label and a value to sign) allows the creation of zone signatures that can be validated against the respective blinded zone key.

Zone Top-Level Domain A GNS Zone Top-Level Domain (zTLD) is a sequence of GNS labels at the end of a GNS name which encodes a zone type and zone key of a zone. Due to the statistical uniqueness of zone keys, zTLDs are also globally unique. A zTLD label sequence can only be distinguished from ordinary TLD label sequences by attempting to decode the labels into a zone type and zone key.

Zone Type The type of a GNS zone determines the cipher system and binary encoding format of the zone key, blinded zone keys, and signatures.

3. Overview

GNS exhibits the three properties that are commonly used to describe a petname system:

1. Global names through the concept of zone top-level domains (zTLDs): As zones can be uniquely identified by their zone key and are statistically unique, zTLDs are globally unique mappings to zones. Consequently, GNS domain names with a zTLD suffix are also globally unique. Names with zTLDs suffixes are not human-readable.

2. Memorable petnames for zones: Users can configure local, human-readable references to zones. Such petnames serve as zTLD monikers in order to support human-readable domain names. The petnames may also be published in order to delegate namespaces of zones.

3. A secure mapping from names to records: GNS allows zone owners to map petnames to resource records or to delegate authority of the petname to other zones and publish this information. The

mappings are signed and encrypted using keys derived from local labels. When names are resolved, resource records including delegations can be verified by the implementation.

It follows from the above that GNS does not support names which are simultaneously global, secure and human-readable. Instead, names are either global and not human-readable or not globally unique and human-readable. An example for a global name pointing to the record "example" in a zone is:

```
example.000G006K2TJNMD9VTCYRX7BRVV3HAEPS15E6NHDXKPJA1KAJJEG9AFF884
```

Now consider the petname "pet" for the example zone of the name above. The following name would point to the same record as the globally unique name above but it is only valid locally:

```
example.pet
```

The delegation of petnames and subsequent resolution of delegation builds on ideas from the Simple Distributed Security Infrastructure [SDSI]. In GNS, any user can create and manage one or more zones (Section 4) as part of a zone master implementation. The zone type determines the respective set of cryptographic operations and the wire formats for encrypted data, public keys and signatures. A zone can be populated with mappings from labels to resource records by its owner (Section 5). A label can be mapped to a delegation record which results in the corresponding subdomain being delegated to another zone. Circular delegations are explicitly allowed, including delegating a subdomain to its immediate parent zone. In order to support (legacy) applications as well as to facilitate the use of petnames, GNS defines auxiliary record types in addition to supporting existing DNS records.

Zone contents are encrypted and signed before being published in a key-value storage (Section 6) as illustrated in Figure 1. In this process, unique zone identification is hidden from the network through the use of key blinding. Key blinding allows the creation of signatures for zone contents using a blinded public/private key pair. This blinding is realized using a deterministic key derivation from the original zone key and corresponding private key using record label values as blinding factors. Specifically, the zone owner can derive blinded private keys for each record set published under a label, and a resolver can derive the corresponding blinded public keys. It is expected that GNS implementations use distributed or decentralized storages such as distributed hash tables (DHT) in order to facilitate availability within a network without the need

for dedicated infrastructure. Specification of such a distributed or decentralized storage is out of scope of this document, but possible existing implementations include those based on [RFC7363], [Kademlia] or [R5N].

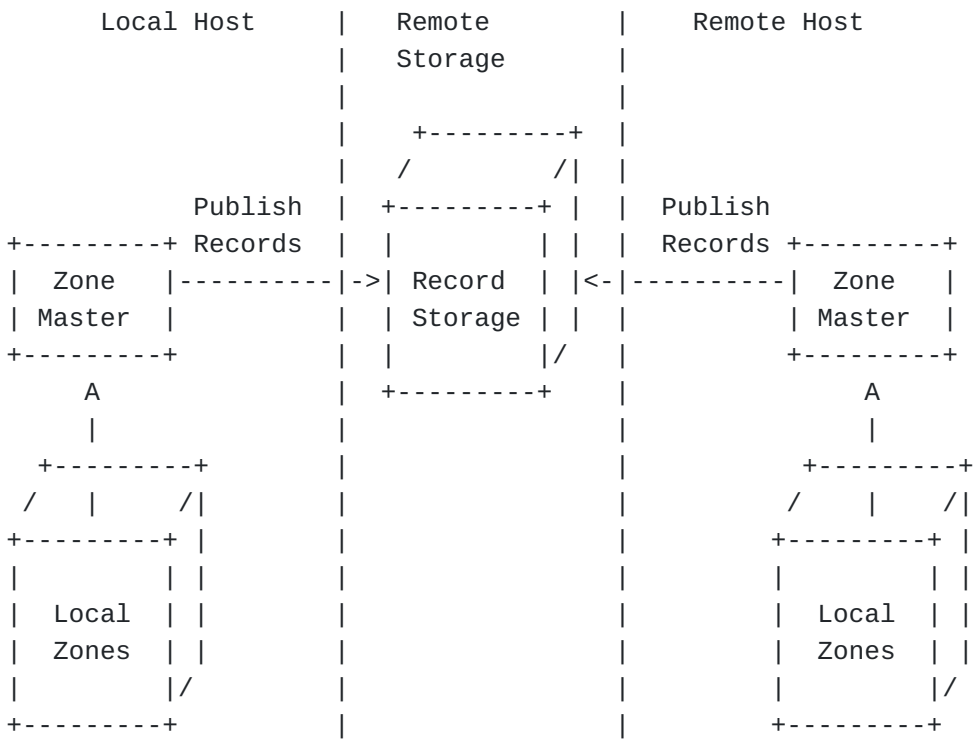

Figure 1: An example diagram of two hosts publishing GNS zones.

Applications use the resolver to lookup GNS names. Starting from a configurable start zone, names are resolved by following zone delegations recursively as illustrated in Figure 2. For each label in a name, the recursive GNS resolver fetches the respective record from the storage layer (Section 7). Without knowledge of the label values and the zone keys, the different derived keys are unlinkable both to the original zone key and to each other. This prevents zone enumeration (except via impractical online brute force attacks) and requires knowledge of both the zone key and the label to confirm affiliation of a query or the corresponding encrypted record set with a specific zone. At the same time, the blinded zone key provides resolvers with the ability to verify the integrity of the published information without disclosing the originating zone.

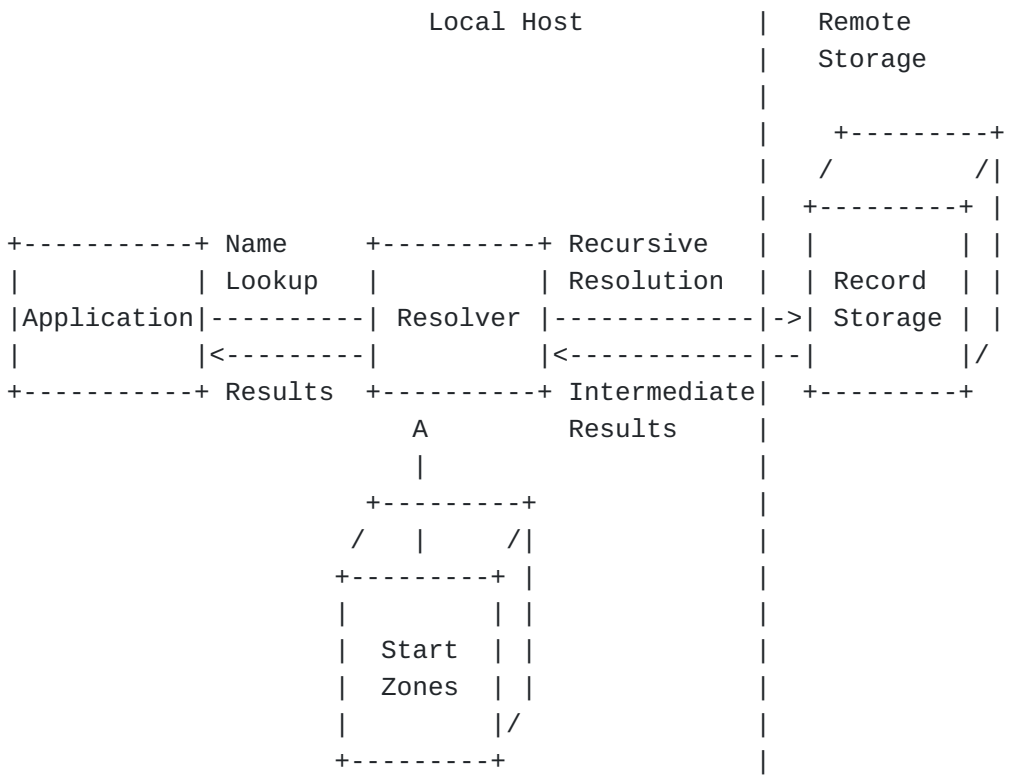

Figure 2: High-level view of the GNS resolution process.

In the remainder of this document, the "implementer" refers to the developer building a GNS implementation including the resolver, zone master, and supporting configuration such as start zones (Section 7.1).

4. Zones

A zone master implementation **SHOULD** enable the zone owners to create and manage zones. If this functionality is not implemented, names can still be resolved if zone keys for the initial step in the name resolution are available (see Section 7).

A zone in GNS is uniquely identified by its zone type and zone key. Each zone can be represented by a Zone Top-Level Domain (zTLD) string. A zone type (ztype) is a unique 32-bit number. This number corresponds to a resource record type number identifying a delegation record type in the GNUnet Assigned Numbers Authority [GANA]. The ztype is a unique identifier for the set cryptographic functions of the zone and the format of the delegation record type. Any ztype **MUST** define the following set of cryptographic functions:

KeyGen() -> d, zk
: is a function to generate a new private key d and the corresponding public zone key zk.

ZKDF(zk,label) -> zk'
: is a zone key derivation function which blinds a zone key zk using a label. zk and zk' must be unlinkable. Furthermore, blinding zk with different values for the label must result in different, unlinkable zk' values.

S-Encrypt(zk,label,expiration,message) -> ciphertext
: is a symmetric encryption function which encrypts the record data based on key material derived from the zone key, a label, and an expiration timestamp. In order to leverage performance-enhancing caching features of certain underlying storages, in particular DHTs, a deterministic encryption scheme is recommended.

S-Decrypt(zk,label,expiration,ciphertext) -> message
: is a symmetric decryption function which decrypts the encrypted record data based on key material derived from the zone key, a label, and an expiration timestamp.

Sign(d,message) -> signature
: is a function to sign a message using the private key d, yielding an unforgeable cryptographic signature. In order to leverage performance-enhancing caching features of certain underlying storages, in particular DHTs, a deterministic signature scheme is recommended.

Verify(zk,message,signature) -> boolean
: is a function to verify the signature was created using the private key d corresponding to the zone key zk where d,zk := Keygen(). The function returns a boolean value of "TRUE" if the signature is valid, and otherwise "FALSE".

SignDerived(d,label,message) -> signature
: is a function to sign a message (typically encrypted record data) that can be verified using the derived zone key zk' := ZKDF(zk,label). In order to leverage performance-enhancing caching features of certain underlying storages, in particular DHTs, a deterministic signature scheme is recommended.

VerifyDerived(zk,label,message,signature) -> boolean
: is function to verify the signature using the derived zone key zk' := ZKDF(zk,label). The function returns a boolean value of "TRUE" if the signature is valid, and otherwise "FALSE".

The cryptographic functions of the default ztypes are specified with their corresponding delegation records in Section 5.1. In order to support cryptographic agility, additional ztypes **MAY** be defined in the future which replace or update the default ztypes defined in this document. All ztypes **MUST** be registered as dedicated zone

delegation record types in the GNU Name System Record Types registry (see Section 10). When defining new record types the cryptographic security considerations of this document apply, in particular Section 9.3.

4.1. Zone Top-Level Domain

The zTLD is the Zone Top-Level Domain. It is a string which encodes the zone type and zone key into a domain name. The zTLD is used as a globally unique reference to a specific zone in the process of name resolution. It is created by encoding a binary concatenation of the zone type and zone key (see Figure 3). The used encoding is a variation of the Crockford Base32 encoding [CrockfordB32] called Base32GNS. The encoding and decoding symbols for Base32GNS including this modification are defined in the table found in Figure 29. The functions for encoding and decoding based on this table are called Base32GNS-Encode and Base32GNS-Decode, respectively.

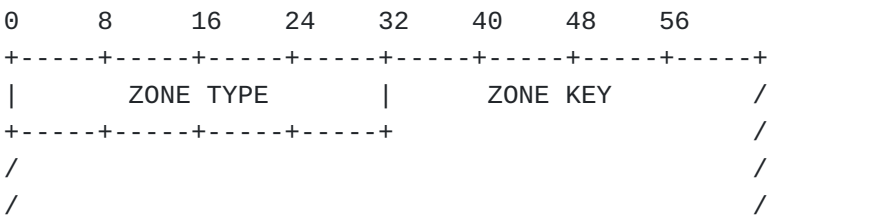

Figure 3: The decoded binary representation of the zTLD

Consequently, a zTLD is encoded and decoded as follows:

```
zTLD := Base32GNS-Encode(ztype||zkey)
ztype||zkey := Base32GNS-Decode(zTLD)
```

where "||" is the concatenation operator.

The zTLD can be used as-is as a rightmost label in a GNS name. If an application wants to ensure DNS compatibility of the name, it **MAY** also represent the zTLD as follows: If the zTLD is less than or equal to 63 characters, it can be used as a zTLD as-is. If the zTLD is longer than 63 characters, the zTLD is divided into smaller labels separated by the label separator. Here, the most significant bytes of the "ztype||zkey" concatenation must be contained in the rightmost label of the resulting string and the least significant bytes in the leftmost label of the resulting string. This allows the resolver to determine the ztype and zTLD length from the rightmost label and to subsequently determine how many labels the zTLD should span. A GNS implementation **MUST** support the division of zTLDs in DNS compatible label lengths. For example, assuming a zTLD of 130 characters, the division is:

zTLD[126..129].zTLD[63..125].zTLD[0..62]

4.2. Zone Revocation

In order to revoke a zone key, a signed revocation message **MUST** be published. This message **MUST** be signed using the private key. The revocation message is broadcast to the network. The specification of the broadcast mechanism is out of scope for this document. A possible broadcast mechanism for efficient flooding in a distributed network is implemented in [GNUnet]. Alternatively, revocation messages could also be distributed via a distributed ledger or a trusted central server. To prevent flooding attacks, the revocation message **MUST** contain a proof of work (PoW). The revocation message including the PoW **MAY** be calculated ahead of time to support timely revocation.

For all occurrences below, "Argon2id" is the Password-based Key Derivation Function as defined in [RFC9106]. For the PoW calculations the algorithm is instantiated with the following parameters:

S The salt. Fixed 16-byte string: "GnsRevocationPow".

t Number of iterations: 3

m Memory size in KiB: 1024

T Output length of hash in bytes: 64

p Parallelization parameter: 1

v Algorithm version: 0x13

y Algorithm type (Argon2id): 2

X Unused

K Unused

Figure 4 illustrates the format of the data "P" on which the PoW is calculated.

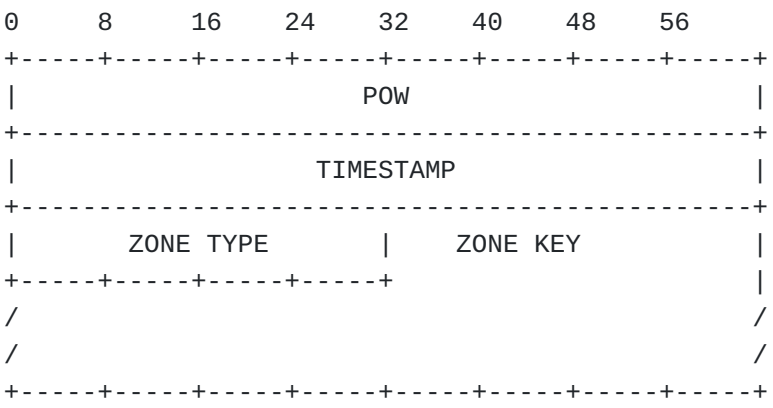

Figure 4: The Format of the PoW Data.

POW A 64-bit value that is a solution to the PoW. In network byte order.

TIMESTAMP denotes the absolute 64-bit date when the revocation was computed. In microseconds since midnight (0 hour), January 1, 1970 UTC in network byte order.

ZONE TYPE is the 32-bit zone type.

ZONE KEY is the 256-bit public key zk of the zone which is being revoked. The wire format of this value is defined by the ZONE TYPE.

Usually, PoW schemes require to find one POW value such that a specific number of leading zeroes are found in the hash result. This number is then referred to as the difficulty of the PoW. In order to reduce the variance in time it takes to calculate the PoW, a valid GNS revocation requires that a number Z different PoWs must be found that on average have D leading zeroes.

The resulting proofs are ready for dissemination. The concrete dissemination and publication methods are out of scope of this document. Given an average difficulty of D, the proofs have an expiration time of EPOCH. With each additional bit difficulty, the lifetime of the proof is prolonged for another EPOCH. Consequently, by calculating a more difficult PoW, the lifetime of the proof can be increased on demand by the zone owner.

The parameters are defined as follows:

Z
The number of PoWs that are required. Its value is fixed at 32.

D The lower limit of the average difficulty. Its value is fixed at 22.

EPOCH A single epoch. Its value is fixed at 365 days in microseconds.

The revocation message wire format is illustrated in Figure 5.

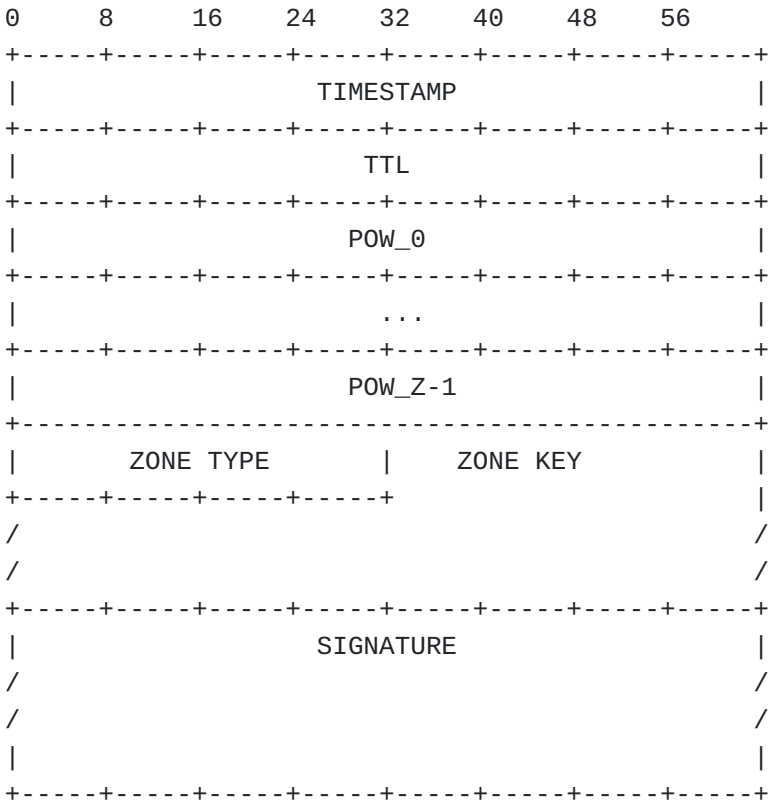

Figure 5: The Revocation Message Wire Format.

TIMESTAMP denotes the absolute 64-bit date when the revocation was computed. In microseconds since midnight (0 hour), January 1, 1970 UTC in network byte order. This is the same value as the time stamp used in the individual PoW calculations.

TTL denotes the relative 64-bit time to live of the record in microseconds in network byte order. The field **SHOULD** be set to EPOCH * 1.1. Given an average number of leading zeros D', then the field value **MAY** be increased up to (D'-D+1) * EPOCH * 1.1. Validators **MAY** reject messages with lower or higher values when received. The EPOCH is extended by 10% in order to deal with unsynchronized clocks.

POW_i
: The values calculated as part of the PoW, in network byte order. Each POW_i **MUST** be unique in the set of POW values. To facilitate fast verification of uniqueness, the POW values must be given in strictly monotonically increasing order in the message.

ZONE TYPE The 32-bit zone type corresponding to the zone key.

ZONE KEY is the public key zk of the zone which is being revoked and the key to be used to verify SIGNATURE.

SIGNATURE A signature over a time stamp and the zone zk of the zone which is revoked and corresponds to the key used in the PoW. The signature is created using the Sign() function of the cryptosystem of the zone and the private key (see Section 4).

The signature over the public key covers a 32-bit header prefixed to the time stamp and public key fields. The header includes the key length and signature purpose. The wire format is illustrated in Figure 6.

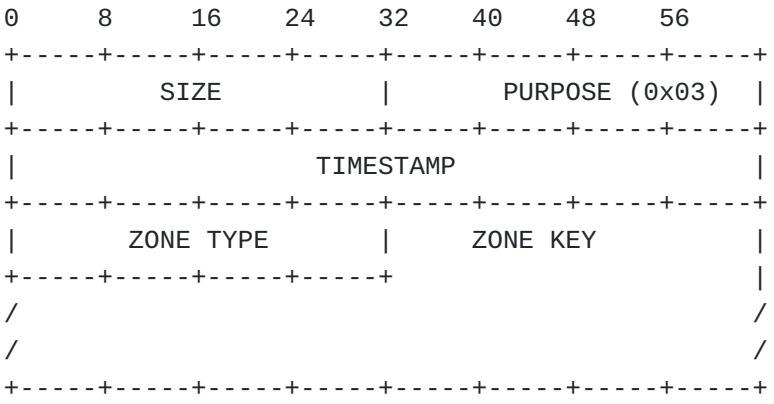

Figure 6: The Wire Format of the Revocation Data for Signing.

SIZE A 32-bit value containing the length of the signed data in bytes in network byte order.

PURPOSE A 32-bit signature purpose flag. The value of this field **MUST** be 3. The value is encoded in network byte order. It defines the context in which the signature is created so that it cannot be reused in other parts of the protocol including possible future extensions. The value of this field corresponds to an entry in the GANA "GNUnet Signature Purpose" registry Section 10.

TIMESTAMP Field as defined in the revocation message above.

ZONE TYPE Field as defined in the revocation message above.

ZONE KEY
: Field as defined in the revocation message above.

In order to validate a revocation the following steps **MUST** be taken:

1. The signature **MUST** be verified against the zone key.

2. The set of POW values **MUST** NOT contain duplicates which **MUST** be checked by verifying that the values are strictly monotonically increasing.

3. The average number of leading zeroes D' resulting from the provided POW values **MUST** be greater than or equal to D. Implementers **MUST NOT** use an integer data type to calculate or represent D'.

The TTL field in the revocation message is informational. A revocation **MAY** be discarded without checking the POW values or the signature if the TTL (in combination with TIMESTAMP) indicates that the revocation has already expired. The actual validity period of the revocation **MUST** be determined by examining the leading zeroes in the POW values.

The validity period of the revocation is calculated as (D'-D+1) * EPOCH * 1.1. The EPOCH is extended by 10% in order to deal with unsynchronized clocks. The validity period added on top of the TIMESTAMP yields the expiration date. If the current time is after the expiration date, the revocation is considered stale.

Verified revocations **MUST** be stored locally. The implementation **MAY** discard stale revocations and evict then from the local store at any time.

Implementations **MUST** broadcast received revocations if they are valid and not stale. Should the calculated validity period differ from the TTL field value, the calculated value **MUST** be used as TTL field value when forwarding the revocation message. Systems might disagree on the current time, so implementations **MAY** use stale but otherwise valid revocations but **SHOULD NOT** broadcast them. Forwarded stale revocations **MAY** be discarded.

Any locally stored revocation **MUST** be considered during delegation record processing (Section 7.3.4).

5. Resource Records

A GNS implementation **SHOULD** provide a mechanism to create and manage local zones as well as a persistence mechanism such as a database for resource records. A new local zone is established by selecting a zone type and creating a zone key pair. If this mechanism is not

implemented, no zones can be published in the storage (Section 6) and name resolution is limited to non-local start zones (Section 7.1).

A GNS resource record holds the data of a specific record in a zone. The resource record format is defined in Figure 7.

Figure 7: The Resource Record Wire Format.

EXPIRATION denotes the absolute 64-bit expiration date of the record. In microseconds since midnight (0 hour), January 1, 1970 UTC in network byte order.

SIZE denotes the 16-bit size of the DATA field in bytes and in network byte order.

FLAGS is a 16-bit resource record flags field (see below).

TYPE is the 32-bit resource record type. This type can be one of the GNS resource records as defined in Section 5 or a DNS record type as defined in [RFC1035] or any of the complementary standardized DNS resource record types. This value must be stored in network byte order. Note that values below 2^16 are reserved for 16-bit DNS Resorce Record types allocated by IANA [RFC6895]. Values above 2^16 are allocated by the GNUnet Assigned Numbers Authority [GANA].

DATA the variable-length resource record data payload. The content is defined by the respective type of the resource record.

Flags indicate metadata surrounding the resource record. An application creating resource records **MUST** set all bits to 0 unless it wants to set the respective flag. As additional flags can be defined in future protocol versions, if an application or implementation encounters a flag which it does not recognize, it **MUST** be ignored. Any combination of the flags specified below are valid. Figure 8 illustrates the flag distribution in the 16-bit flag field of a resource record:

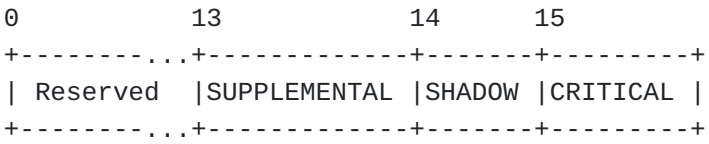

Figure 8: The Resource Record Flag Wire Format.

CRITICAL If this flag is set, it indicates that processing is critical. Implementations that do not support the record type or are otherwise unable to process the record **MUST** abort resolution upon encountering the record in the resolution process.

SHADOW If this flag is set, this record **MUST** be ignored by resolvers unless all (other) records of the same record type have expired. Used to allow zone publishers to facilitate good performance when records change by allowing them to put future values of records into the storage. This way, future values can propagate and can be cached before the transition becomes active.

SUPPLEMENTAL This is a supplemental record. It is provided in addition to the other records. This flag indicates that this record is not explicitly managed alongside the other records under the respective name but might be useful for the application.

5.1. Zone Delegation Records

This section defines the initial set of zone delegation record types. Any implementation **SHOULD** support all zone types defined here and **MAY** support any number of additional delegation records defined in the GNU Name System Record Types registry (see Section 10). Not supporting some zone types will result in resolution failures in case the respective zone type is encountered. This is be a valid choice if some zone delegation record types have been determined to be cryptographically insecure. Zone delegation records **MUST NOT** be stored and published under the apex label. A zone delegation record type value is the same as the respective ztype value. The ztype defines the cryptographic primitives for the zone that is being delegated to. A zone delegation record payload contains the public key of the zone to delegate to. A zone delegation record **MUST** have the CRITICAL flag set and **MUST** be the only non-supplemental record under a label. There **MAY** be inactive records of the same type which have the SHADOW flag set in order to facilitate smooth key rollovers.

In the following, "||" is the concatenation operator of two byte strings. The algorithm specification uses character strings such as GNS labels or constant values. When used in concatenations or as

input to functions the null-terminator of the character strings **MUST NOT** be included.

5.1.1. PKEY

In GNS, a delegation of a label to a zone of type "PKEY" is represented through a PKEY record. The PKEY DATA entry wire format can be found in Figure 9.

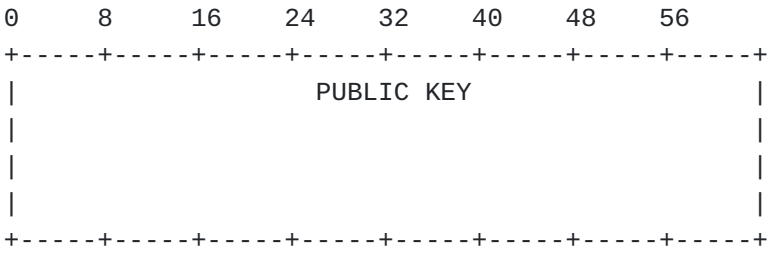

Figure 9: The PKEY Wire Format.

PUBLIC KEY A 256-bit Ed25519 public key.

For PKEY zones the zone key material is derived using the curve parameters of the twisted Edwards representation of Curve25519 [RFC7748] (a.k.a. Ed25519) with the ECDSA scheme [RFC6979]. The following naming convention is used for the cryptographic primitives of PKEY zones:

d is a 256-bit Ed25519 private key (private scalar).

zk is the Ed25519 public zone key corresponding to d.

p is the prime of edwards25519 as defined in [RFC7748], i.e. 2^255 - 19.

G is the group generator (X(P),Y(P)) of edwards25519 as defined in [RFC7748].

L is the order of the prime-order subgroup of edwards25519 in [RFC7748].

KeyGen() The generation of the private scalar d and the curve point zk := d*G (where G is the group generator of the elliptic curve) as defined in Section 2.2. of [RFC6979] represents the KeyGen() function.

The zone type and zone key of a PKEY are 4 + 32 bytes in length. This means that a zTLD will always fit into a single label and does not need any further conversion. Given a label, the output zk' of the ZKDF(zk,label) function is calculated as follows for PKEY zones:

```
ZKDF(zk,label):
  PRK_h := HKDF-Extract ("key-derivation", zk)
  h := HKDF-Expand (PRK_h, label || "gns", 512 / 8)
  zk' := (h mod L) * zk
  return zk'
```

The PKEY cryptosystem uses a hash-based key derivation function (HKDF) as defined in [RFC5869], using SHA-512 [RFC6234] for the extraction phase and SHA-256 [RFC6234] for the expansion phase. PRK_h is key material retrieved using an HKDF using the string "key-derivation" as salt and the zone key as initial keying material. h is the 512-bit HKDF expansion result and must be interpreted in network byte order. The expansion information input is a concatenation of the label and the string "gns". The multiplication of zk with h is a point multiplication, while the multiplication of d with h is a scalar multiplication.

The Sign() and Verify() functions for PKEY zones are implemented using 512-bit ECDSA deterministic signatures as specified in [RFC6979]. The same functions can be used for derived keys:

```
SignDerived(d,label,message):
  zk := d * G
  PRK_h := HKDF-Extract ("key-derivation", zk)
  h := HKDF-Expand (PRK_h, label || "gns", 512 / 8)
  d' := (h * d) mod L
  return Sign(d',message)
```

A signature (R,S) is valid if the following holds:

```
VerifyDerived(zk,label,message,signature):
  zk' := ZKDF(zk,label)
  return Verify(zk',message,signature)
```

The S-Encrypt() and S-Decrypt() functions use AES in counter mode as defined in [MODES] (CTR-AES-256):

```
S-Encrypt(zk,label,expiration,plaintext):
  PRK_k := HKDF-Extract ("gns-aes-ctx-key", zk)
  PRK_n := HKDF-Extract ("gns-aes-ctx-iv", zk)
  K := HKDF-Expand (PRK_k, label, 256 / 8)
  NONCE := HKDF-Expand (PRK_n, label, 32 / 8)
  IV := NONCE || expiration || 0x0000000000000001
  return CTR-AES256(K, IV, plaintext)

S-Decrypt(zk,label,expiration,ciphertext):
  PRK_k := HKDF-Extract ("gns-aes-ctx-key", zk)
  PRK_n := HKDF-Extract ("gns-aes-ctx-iv", zk)
  K := HKDF-Expand (PRK_k, label, 256 / 8)
  NONCE := HKDF-Expand (PRK_n, label, 32 / 8)
  IV := NONCE || expiration || 0x0000000000000001
  return CTR-AES256(K, IV, ciphertext)
```

The key K and counter IV are derived from the record label and the zone key zk using a hash-based key derivation function (HKDF) as defined in [RFC5869]. SHA-512 [RFC6234] is used for the extraction phase and SHA-256 [RFC6234] for the expansion phase. The output keying material is 32 bytes (256 bits) for the symmetric key and 4 bytes (32 bits) for the nonce. The symmetric key K is a 256-bit AES [RFC3826] key.

The nonce is combined with a 64-bit initialization vector and a 32-bit block counter as defined in [RFC3686]. The block counter begins with the value of 1, and it is incremented to generate subsequent portions of the key stream. The block counter is a 32-bit integer value in network byte order. The initialization vector is the expiration time of the resource record block in network byte order. The resulting counter (IV) wire format can be found in Figure 10.

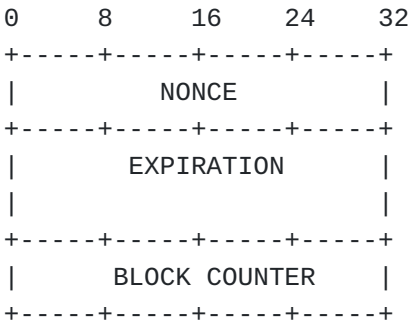

Figure 10: The Block Counter Wire Format.

5.1.2. EDKEY

In GNS, a delegation of a label to a zone of type "EDKEY" is represented through a EDKEY record. The EDKEY DATA entry wire format is illustrated in Figure 11.

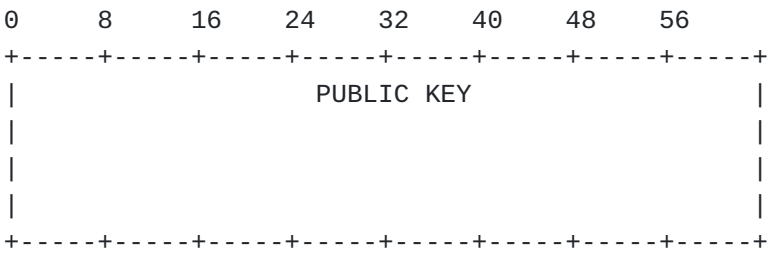

Figure 11: The EDKEY DATA Wire Format.

PUBLIC KEY A 256-bit EdDSA zone key.

For EDKEY zones the zone key material is derived using the curve parameters of the twisted edwards representation of Curve25519 [RFC7748] (a.k.a. Ed25519) with the Ed25519 scheme [ed25519] as specified in [RFC8032]. The following naming convention is used for the cryptographic primitives of EDKEY zones:

d is a 256-bit EdDSA private key.

a is is an integer derived from d using the SHA-512 hash function as defined in [RFC8032].

zk is the EdDSA public key corresponding to d. It is defined as the curve point a*G where G is the group generator of the elliptic curve as defined in [RFC8032].

p is the prime of edwards25519 as defined in [RFC8032], i.e. 2^255 - 19.

G is the group generator (X(P),Y(P)) of edwards25519 as defined in [RFC8032].

L is the order of the prime-order subgroup of edwards25519 in [RFC8032].

KeyGen() The generation of the private key d and the associated public key zk := a*G where G is the group generator of the elliptic curve and a is an integer derived from d using the SHA-512 hash function as defined in Section 5.1.5 of [RFC8032] represents the KeyGen() function.

The zone type and zone key of an EDKEY are 4 + 32 bytes in length. This means that a zTLD will always fit into a single label and does not need any further conversion.

The "EDKEY" ZKDF instantiation is based on [Tor224]. The calculation of a is defined in Section 5.1.5 of [RFC8032]. Given a label, the output of the ZKDF function is calculated as follows:

```
ZKDF(zk,label):
  /* Calculate the blinding factor */
  PRK_h := HKDF-Extract ("key-derivation", zk)
  h := HKDF-Expand (PRK_h, label || "gns", 512 / 8)
  /* Ensure that h == h mod L */
  h[31] &= 7

  zk' := h * zk
  return zk'
```

Implementers **SHOULD** employ a constant time scalar multiplication for the constructions above to protect against timing attacks. Otherwise, timing attacks could leak private key material if an attacker can predict when a system starts the publication process.

The EDKEY cryptosystem uses a hash-based key derivation function (HKDF) as defined in [RFC5869], using SHA-512 [RFC6234] for the extraction phase and HMAC-SHA256 [RFC6234] for the expansion phase. PRK_h is key material retrieved using an HKDF using the string "key-derivation" as salt and the zone key as initial keying material. The blinding factor h is the 512-bit HKDF expansion result. The expansion information input is a concatenation of the label and the string "gns". The result of the HKDF must be clamped and interpreted in network byte order. a is the 256-bit integer corresponding to the 256-bit private key d. The multiplication of zk with h is a point multiplication, while the division and multiplication of a and a1 with the co-factor are integer operations.

The Sign(d,message) and Verify(zk,message,signature) procedures **MUST** be implemented as defined in [RFC8032].

Signatures for EDKEY zones use a derived private scalar d' which is not compliant with [RFC8032]. As the corresponding private key to the derived private scalar is not known, it is not possible to deterministically derive the signature part R according to [RFC8032]. Instead, signatures **MUST** be generated as follows for any given message and private zone key: A nonce is calculated from the highest 32 bytes of the expansion of the private key d and the blinding factor h. The nonce is then hashed with the message to r. This way, the full derivation path is included in the calculation of the R value of the signature, ensuring that it is never reused for two different derivation paths or messages.

```
SignDerived(d,label,message):
  /* Key expansion */
  dh := SHA-512 (d)
  /* EdDSA clamping */
  a := dh[0..31]
  a[0] &= 248
  a[31] &= 127
  a[31] |= 64
  /* Calculate zk corresponding to d */
  zk := a * G

  /* Calculate blinding factor */
  PRK_h := HKDF-Extract ("key-derivation", zk)
  h := HKDF-Expand (PRK_h, label || "gns", 512 / 8)
  /* Ensure that h == h mod L */
  h[31] &= 7

  zk' := h * zk
  a1 := a >> 3
  a2 := (h * a1) mod L
  d' := a2 << 3
  nonce := SHA-256 (dh[32..63] || h)
  r := SHA-512 (nonce || message)
  R := r * G
  S := r + SHA-512(R || zk' || message) * d' mod L
  return (R,S)
```

A signature (R,S) is valid if the following holds:

```
VerifyDerived(zk,label,message,signature):
  zk' := ZKDF(zk,label)
  (R,S) := signature
  return S * G == R + SHA-512(R, zk', message) * zk'
```

The S-Encrypt() and S-Decrypt() functions use XSalsa20 as defined in [XSalsa20] (XSalsa20-Poly1305):

```
S-Encrypt(zk,label,expiration,message):
  PRK_k := HKDF-Extract ("gns-xsalsa-ctx-key", zk)
  PRK_n := HKDF-Extract ("gns-xsalsa-ctx-iv", zk)
  K := HKDF-Expand (PRK_k, label, 256 / 8)
  NONCE := HKDF-Expand (PRK_n, label, 128 / 8)
  IV := NONCE || expiration
  return XSalsa20-Poly1305(K, IV, message)

S-Decrypt(zk,label,expiration,ciphertext):
  PRK_k := HKDF-Extract ("gns-xsalsa-ctx-key", zk)
  PRK_n := HKDF-Extract ("gns-xsalsa-ctx-iv", zk)
  K := HKDF-Expand (PRK_k, label, 256 / 8)
  NONCE := HKDF-Expand (PRK_n, label, 128 / 8)
  IV := NONCE || expiration
  return XSalsa20-Poly1305(K, IV, ciphertext)
```

The result of the XSalsa20-Poly1305 encryption function is the encrypted ciphertext followed by the 128-bit authentication tag. Accordingly, the length of encrypted data equals the length of the data plus the 16 bytes of the authentication tag.

The key K and counter IV are derived from the record label and the zone key zk using a hash-based key derivation function (HKDF) as defined in [RFC5869]. SHA-512 [RFC6234] is used for the extraction phase and SHA-256 [RFC6234] for the expansion phase. The output keying material is 32 bytes (256 bits) for the symmetric key and 16 bytes (128 bits) for the NONCE. The symmetric key K is a 256-bit XSalsa20 [XSalsa20] key. No additional authenticated data (AAD) is used.

The nonce is combined with an 8 byte initialization vector. The initialization vector is the expiration time of the resource record block in network byte order. The resulting counter (IV) wire format is illustrated in Figure 12.

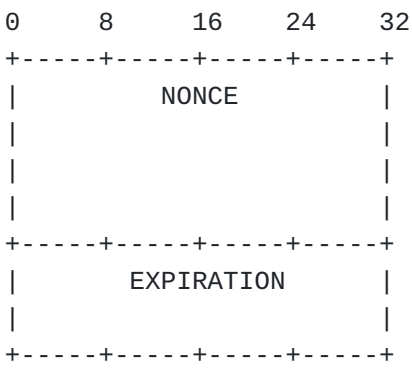

Figure 12: The Counter Block Initialization Vector.

5.2. Redirection Records

Redirect records are used to redirect resolution. Any implementation **SHOULD** support all redirection record types defined here and **MAY** support any number of additional redirection records defined in the GNU Name System Record Types registry (see Section Section 10). Redirection records **MUST** have the CRITICAL flag set. Not supporting some record types can result in resolution failures. This can be a valid choice if some redirection record types have been determined to be insecure, or if an application has reasons to not support redirection to DNS for reasons such as complexity or security. Redirection records **MUST NOT** be stored and published under the apex label.

5.2.1. REDIRECT

A REDIRECT record is the GNS equivalent of a CNAME record in DNS. A REDIRECT record **MUST** be the only non-supplemental record under a label. There **MAY** be inactive records of the same type which have the SHADOW flag set in order to facilitate smooth changes of redirection targets. No other records are allowed. Details on processing of this record is defined in Section 7.3.1. A REDIRECT DATA entry is illustrated in Figure 13.

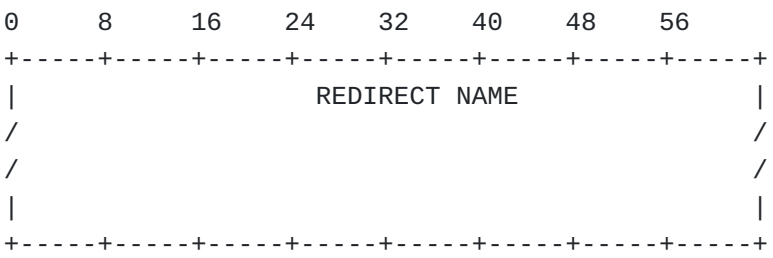

Figure 13: The REDIRECT DATA Wire Format.

REDIRECT NAME The name to continue with. The value of a redirect record can be a regular name, or a relative name. Relative GNS names are indicated by an extension label (U+002B, "+") as rightmost label. The string is UTF-8 encoded and 0-terminated.

5.2.2. GNS2DNS

It is possible to delegate a label back into DNS through a GNS2DNS record. The resource record contains a DNS name for the resolver to continue with in DNS followed by a DNS server. Both names are in the format defined in [RFC1034] for DNS names. There **MAY** be multiple GNS2DNS records under a label. There **MAY** also be DNSSEC DS records or any other records used to secure the connection with the DNS servers under the same label. There **MAY** be inactive records of the same type(s) which have the SHADOW flag set in order to facilitate

smooth changes of redirection targets. No other non-supplemental record types are allowed in the same record set. A GNS2DNS DATA entry is illustrated in Figure 14.

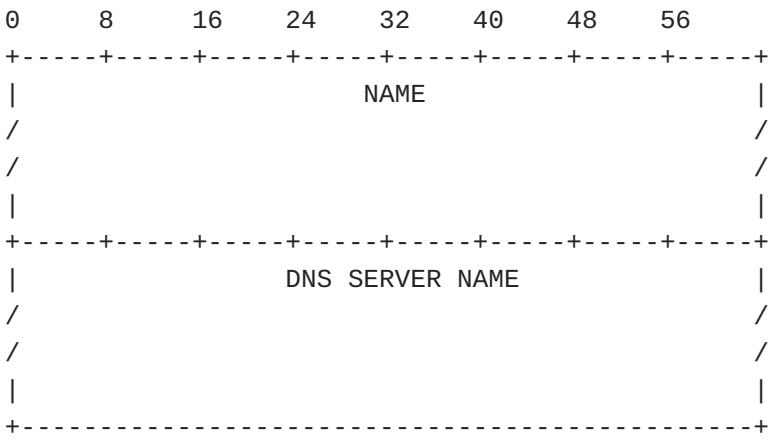

Figure 14: The GNS2DNS DATA Wire Format.

NAME The name to continue with in DNS. The value is UTF-8 encoded and 0-terminated.

DNS SERVER NAME The DNS server to use. This value can be an IPv4 address in dotted-decimal form or an IPv6 address in colon-hexadecimal form or a DNS name. It can also be a relative GNS name ending with a "+" as the rightmost label. The implementation **MUST** check the string syntactically for an IP address in the respective notation before checking for a relative GNS name. If all three checks fail, the name **MUST** be treated as a DNS name. The value is UTF-8 encoded and 0-terminated.

NOTE: If an application uses DNS names obtained from GNS2DNS records in a DNS request they **MUST** first be converted to an IDNA compliant representation [RFC5890].

5.3. Auxiliary Records

This section defines the initial set of auxiliary GNS record types. Any implementation **SHOULD** be able to process the specified record types according to Section 7.3.

5.3.1. LEHO

This record is used to provide a hint for LEgacy HOstnames: Applications can use the GNS to lookup IPv4 or IPv6 addresses of internet services. However, sometimes connecting to such services does not only require the knowledge of an address and port, but also requires the canonical DNS name of the service to be transmitted over the transport protocol. In GNS, legacy host name records

provide applications the DNS name that is required to establish a connection to such a service. The most common use case is HTTP virtual hosting and TLS Server Name Indication [RFC6066], where a DNS name must be supplied in the HTTP "Host"-header and the TLS handshake, respectively. Using a GNS name in those cases might not work as it might not be globally unique. Furthermore, even if uniqueness is not an issue, the legacy service might not even be aware of GNS.

A LEHO resource record is expected to be found together in a single resource record with an IPv4 or IPv6 address. A LEHO DATA entry is illustrated in Figure 15.

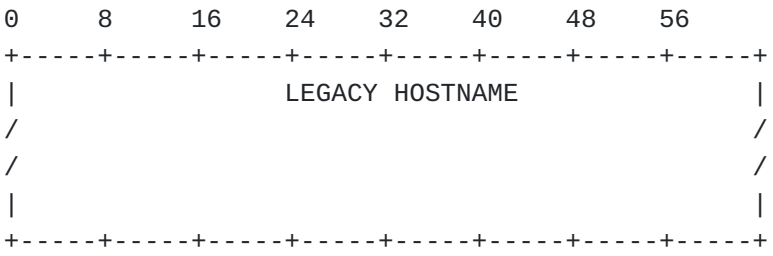

Figure 15: The LEHO DATA Wire Format.

LEGACY HOSTNAME A UTF-8 string (which is not 0-terminated) representing the legacy hostname.

NOTE: If an application uses a LEHO value in an HTTP request header (e.g. "Host:" header) it **MUST** be converted to an IDNA compliant representation [RFC5890].

5.3.2. NICK

Nickname records can be used by zone administrators to publish a label that a zone prefers to have used when it is referred to. This is a suggestion to other zones what label to use when creating a delegation record (Section 5.1) containing this zone key. This record **SHOULD** only be stored under the apex label "@" but **MAY** be returned with record sets under any label as a supplemental record. Section 7.3.5 details how a resolver must process supplemental and non-supplemental NICK records. A NICK DATA entry is illustrated in Figure 16.

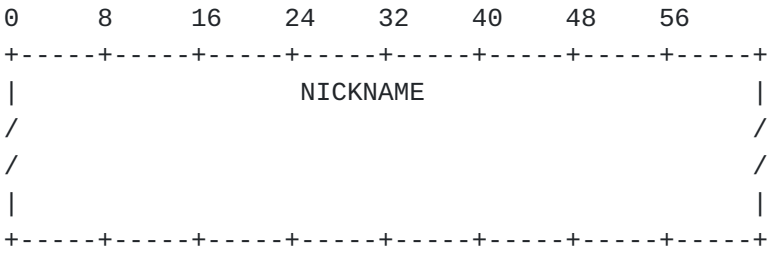

Figure 16: The NICK DATA Wire Format.

NICKNAME A UTF-8 string (which is not 0-terminated) representing the preferred label of the zone. This string **MUST** be a valid GNS label.

5.3.3. BOX

GNS lookups are expected to return all of the required useful information in one record set. This avoids unnecessary additional lookups and cryptographically ties together information that belongs together, making it impossible for an adversarial storage to provide partial answers that might omit information critical for security.

This general strategy is incompatible with the special labels used by DNS for SRV and TLSA records. Thus, GNS defines the BOX record format to box up SRV and TLSA records and include them in the record set of the label they are associated with. For example, a TLSA record for "_https._tcp.example.org" will be stored in the record set of "example.org" as a BOX record with service (SVC) 443 (https) and protocol (PROTO) 6 (tcp) and record TYPE "TLSA". For reference, see also [RFC2782]. A BOX DATA entry is illustrated in Figure 17.

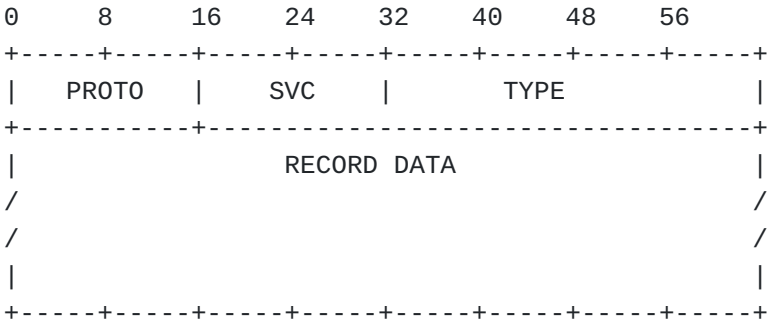

Figure 17: The BOX DATA Wire Format.

PROTO the 16-bit protocol number, e.g. 6 for TCP. Note that values below 2^8 are reserved for 8-bit Internet Protocol numbers allocated by IANA [RFC5237]. Values above 2^8 are allocated by the GNUnet Assigned Numbers Authority [GANA]. In network byte order.

SVC
the 16-bit service value of the boxed record. In case of TCP and UDP it is the port number. In network byte order.

TYPE is the 32-bit record type of the boxed record. In network byte order.

RECORD DATA is a variable length field containing the "DATA" format of TYPE as defined for the respective TYPE in DNS.

6. Record Encoding

Any API which allows storing a value under a 512-bit key and retrieving one or more values from the key can be used by an implementation for record storage. To be useful, the API **MUST** permit storing at least 176 byte values to be able to support the defined zone delegation record encodings, and **SHOULD** allow at least 1024 byte values. In the following, it is assumed that an implementation realizes two procedures on top of a storage:

```
PUT(key,value)
GET(key) -> value
```

There is no explicit delete function as the deletion of a non-expired record would require a revocation of the record. In GNS, zones can only be revoked as a whole. Records automatically expire and it is under the discretion of the storage as to when to delete the record. The GNS implementation **MUST NOT** publish expired resource records. Any GNS resolver **MUST** discard expired records returned from the storage.

Resource records are grouped by their respective labels, encrypted and published together in a single records block (RRBLOCK) in the storage under a storage key q as illustrated in Figure 18. The implementation **MUST** use the PUT storage procedure in order to update the zone contents accordingly.

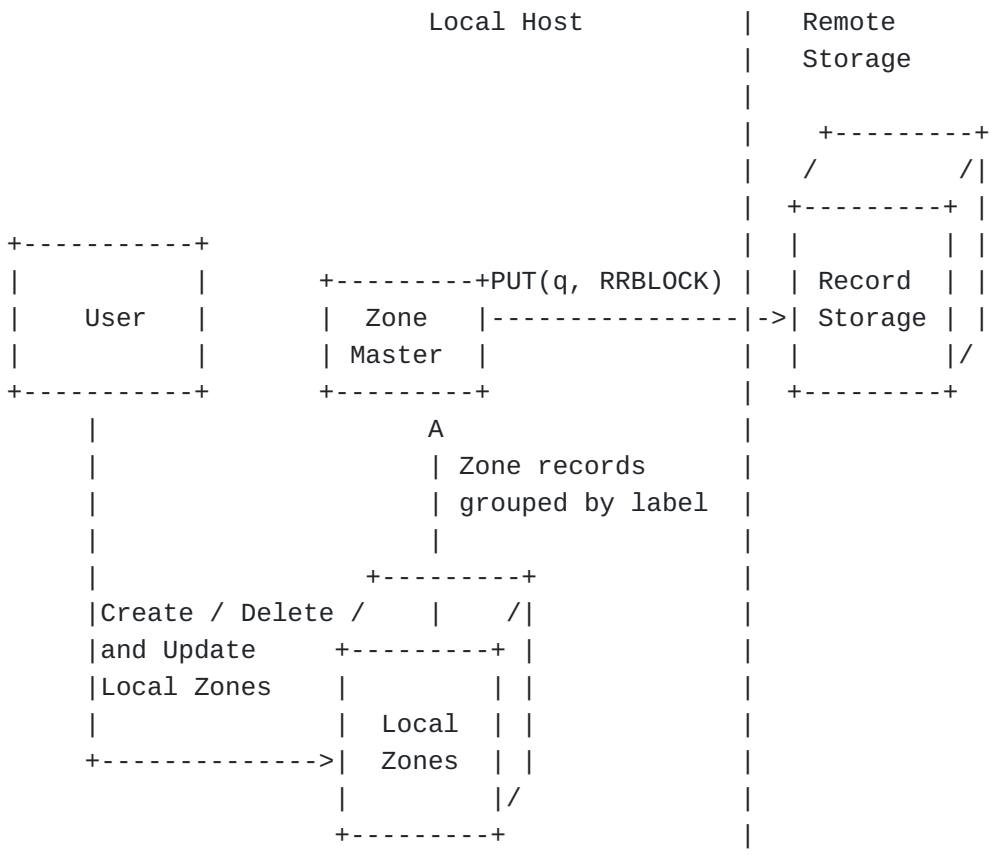

Figure 18: Management and publication of local zones in the distributed storage.

The storage key is derived from the zone key and the respective label of the contained records. The required knowledge of both zone key and label in combination with the similarly derived symmetric secret keys and blinded zone keys ensure query privacy (see [RFC8324], Section 3.5). The storage Key derivation and records block creation using is specified in the following sections and a high-level overview is illustrated in Figure 19.

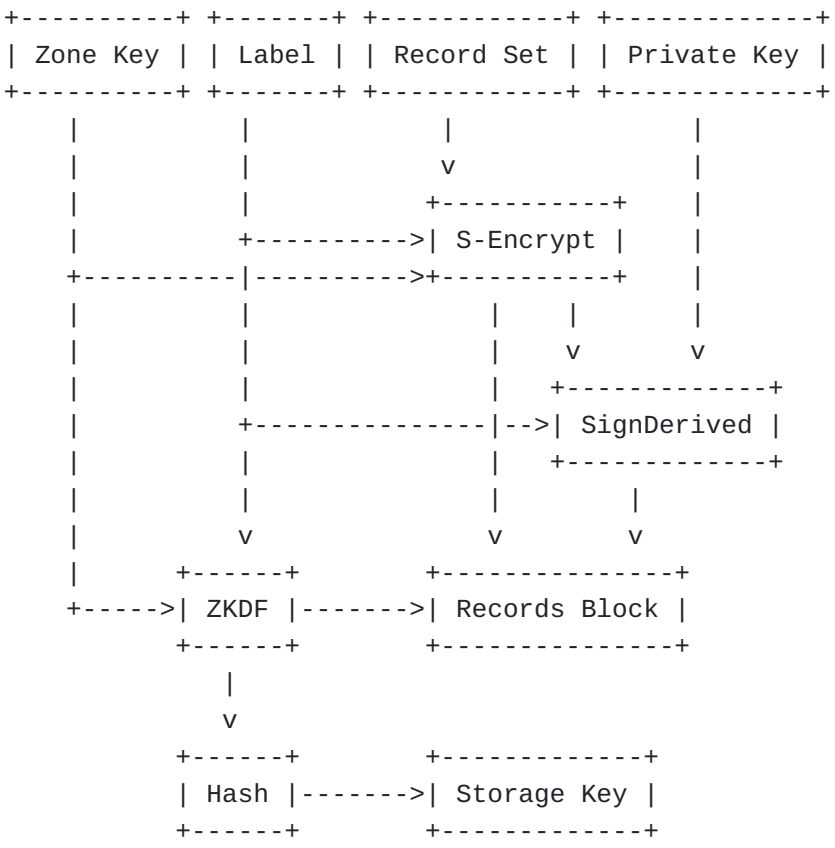

Figure 19: Storage key and records block creation overview.

6.1. The Storage Key

Given a label, the storage key q is derived as follows:

```
q := SHA-512 (ZKDF(zk, label))
```

label is a UTF-8 string under which the resource records are published.

zk is the zone key.

q Is the 512-bit storage key under which the resource records block is published. It is the SHA-512 hash [RFC6234] over the derived zone key.

6.2. The Records Block

GNS records are grouped by their labels and published as a single block in the storage. The grouped record sets **MAY** be paired with any number of supplemental records. Supplemental records **MUST** have the supplemental flag set (See Section 5). The contained resource records are encrypted using a symmetric encryption scheme. A GNS implementation publishes RRBLOCKs in accordance to the properties and recommendations of the underlying storage. This can include a

periodic refresh operation to ensure the availability of the published RRBLOCKs. The GNS RRBLOCK wire format is illustrated in Figure 20.

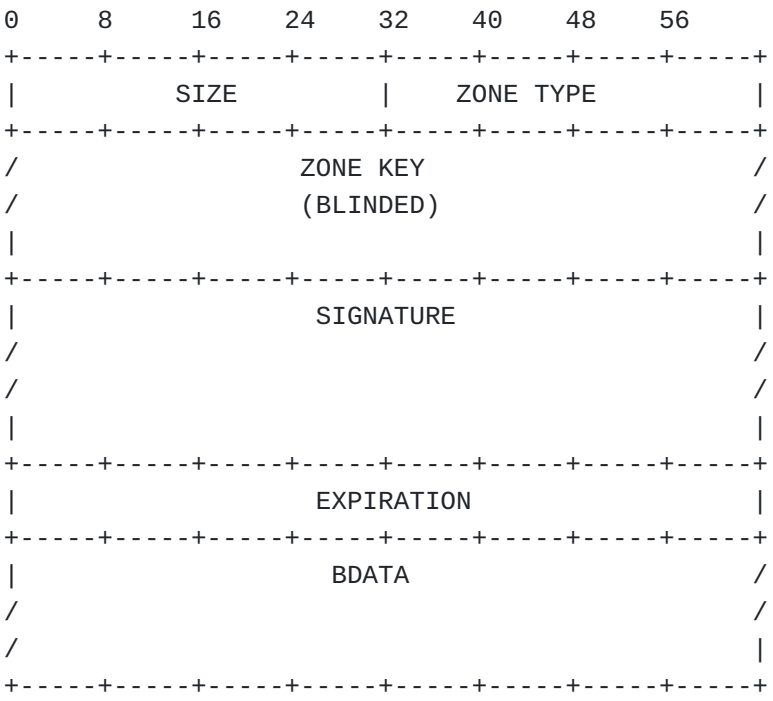

Figure 20: The RRBLOCK Wire Format.

SIZE A 32-bit value containing the length of the block in bytes. In network byte order. While a 32-bit value is used, implementations **MAY** refuse to publish blocks beyond a certain size significantly below 4 GB.

ZONE TYPE is the 32-bit ztype. In network byte order.

ZONE KEY is the blinded zone key "ZKDF(zk, label)" to be used to verify SIGNATURE. The length and format of the public key depends on the ztype.

SIGNATURE The signature is computed over the EXPIRATION and BDATA fields as detailed in Figure 21. The length and format of the signature depends on the ztype. The signature is created using the SignDerived() function of the cryptosystem of the zone (see Section 4).

EXPIRATION Specifies when the RRBLOCK expires and the encrypted block **SHOULD** be removed from the storage and caches as it is likely stale. However, applications **MAY** continue to use non-expired individual records until they expire. The value **MUST** be set to the expiration time of the resource record contained within this block with the smallest expiration time. If a records

block includes shadow records, then the maximum expiration time of all shadow records with matching type and the expiration times of the non-shadow records is considered. This is a 64-bit absolute date in microseconds since midnight (0 hour), January 1, 1970 UTC in network byte order.

BDATA The encrypted RDATA. Its size is determined by the S-Encrypt() function of the ztype.

The signature over the public key covers a 32-bit pseudo header conceptually prefixed to the EXPIRATION and the BDATA fields. The wire format is illustrated in Figure 21.

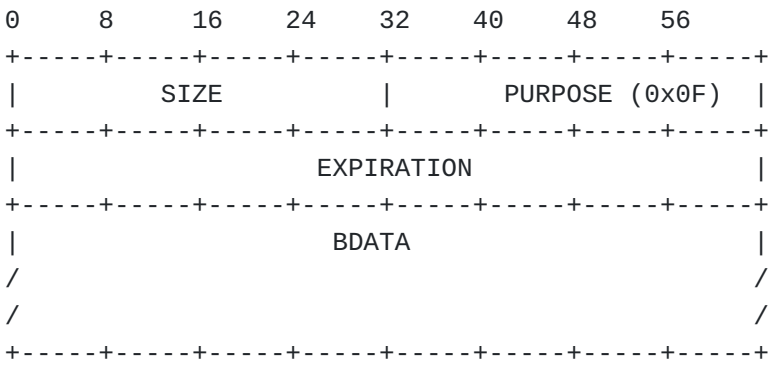

Figure 21: The Wire Format used for creating the signature of the RRBLOCK.

SIZE A 32-bit value containing the length of the signed data in bytes in network byte order.

PURPOSE A 32-bit signature purpose flag. The value of this field **MUST** be 15. The value is encoded in network byte order. It defines the context in which the signature is created so that it cannot be reused in other parts of the protocol including possible future extensions. The value of this field corresponds to an entry in the GANA "GNUnet Signature Purpose" registry Section 10.

EXPIRATION Field as defined in the RRBLOCK message above.

BDATA Field as defined in the RRBLOCK message above.

A symmetric encryption scheme is used to encrypt the resource records set RDATA into the BDATA field of a GNS RRBLOCK. The wire format of the RDATA is illustrated in Figure 22.

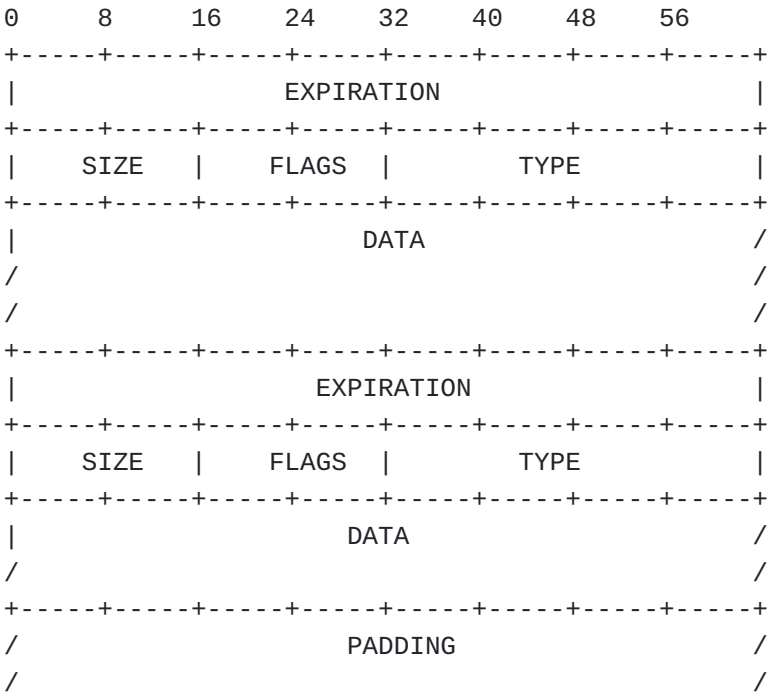

Figure 22: The RDATA Wire Format.

EXPIRATION, SIZE, TYPE, FLAGS and DATA These fields were defined in the resource record format in Section 5.

PADDING When publishing an RDATA block, the implementation **MUST** ensure that the size of the RDATA is a power of two using the padding field. The field **MUST** be set to zero and **MUST** be ignored on receipt. As a special exception, record sets with (only) a zone delegation record type are never padded. Note that a record set with a delegation record **MUST NOT** contain other records. If other records are encountered, the whole record block **MUST** be discarded.

7. Name Resolution

Names in GNS are resolved by recursively querying the record storage. Recursive in this context means that a resolver does not provide intermediate results for a query to the application. Instead, it **MUST** respond to a resolution request with either the requested resource record or an error message in case the resolution fails. Figure 23 illustrates how an application requests the lookup of a GNS name (1). The application **MAY** provide a desired record type to the resolver. Subsequently, the Start Zone is determined (2) and the recursive resolution process started. This is where the desired record type is used to guide processing. For example, if a zone delegation record type is requested, the resolution of the apex label in that zone must be skipped, as the desired record is already found. Details on how the resolution process is initiated and each

iterative result (3a,3b) in the resolution is processed are provided in the sections below. The results of the lookup are eventually returned to the application (4). The implementation **MUST NOT** filter results according to the desired record type. Filtering of record sets is typically done by the application.

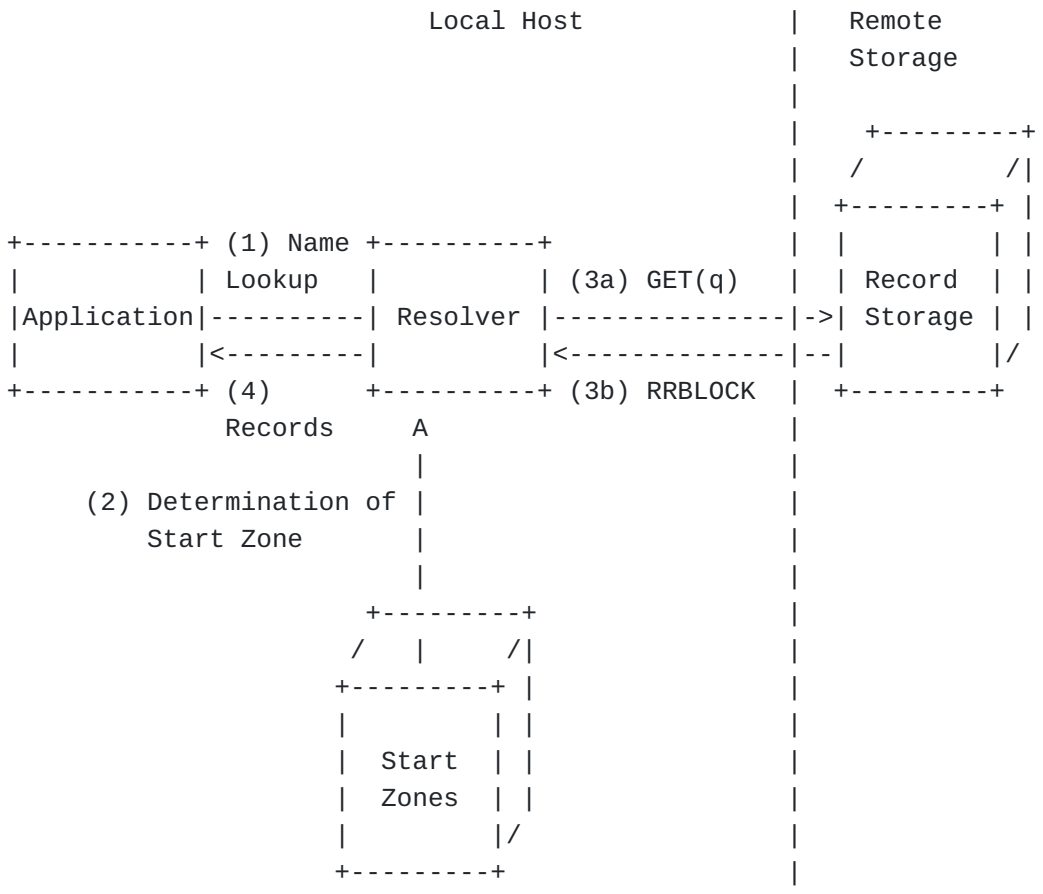

Figure 23: The recursive GNS resolution process.

7.1. Start Zones

The resolution of a GNS name starts by identifying the start zone suffix. Once the start zone suffix is identified, recursive resolution of the remainder of the name is initiated (Section 7.2). There are two types of start zone suffixes: zTLDs and local suffix-to-zone mappings. The choice of available suffix-to-zone mappings is at the sole discretion of the local system administrator or user. This property addresses the issue of a single hierarchy with a centrally controlled root and the related issue of distribution and management of root servers in DNS (see [RFC8324], Section 3.10 and 3.12).

For names ending with a zTLD the start zone is explicitly given in the suffix of the name to resolve. In order to ensure uniqueness of

names with zTLDs any implementation **MUST** use the given zone as start zone. An implementation **MUST** first try to interpret the rightmost label of the given name as the beginning of a zTLD (Section 4.1). If the rightmost label cannot be (partially) decoded or if it does not indicate a supported ztype, the name is treated as a normal name and start zone discovery **MUST** continue with finding a local suffix-to-zone mapping. If a valid ztype can be found in the rightmost label, the implementation **MUST** try to synthesize and decode the zTLD to retrieve the start zone key according to Section 4.1. If the zTLD cannot be synthesized or decoded, the resolution of the name fails and an error is returned to the application. Otherwise, the zone key **MUST** be used as the start zone:

```
Example name: www.example.<zTLD>
=> Start zone: zk of type ztype
=> Name to resolve from start zone: www.example
```

For names not ending with a zTLD the resolver **MUST** determine the start zone through a local suffix-to-zone mapping. Suffix-to-zone mappings **MUST** be configurable through a local configuration file or database by the user or system administrator. A suffix **MAY** consist of multiple GNS labels concatenated with a label separator. If multiple suffixes match the name to resolve, the longest matching suffix **MUST** be used. The suffix length of two results **MUST NOT** be equal. This indicates a misconfiguration and the implementation **MUST** return an error. The following is a non-normative example mapping of start zones:

```
Example name: www.example.org
Local suffix mappings:
org = zTLD0 := Base32GNS(ztype0||zk0)
example.org = zTLD1 := Base32GNS(ztype1||zk1)
example.com = zTLD2 := Base32GNS(ztype2||zk2)
...
=> Start zone: zk1
=> Name to resolve from start zone: www
```

The process given above **MAY** be supplemented with other mechanisms if the particular application requires a different process. If no start zone can be discovered, resolution **MUST** fail and an error **MUST** be returned to the application.

7.2. Recursion

In each step of the recursive name resolution, there is an authoritative zone zk and a name to resolve. The name **MAY** be empty. If the name is empty, it is interpreted as the apex label "@". Initially, the authoritative zone is the start zone.

From here, the following steps are recursively executed, in order:

1. Extract the right-most label from the name to look up.

2. Calculate q using the label and zk as defined in Section 6.1.

3. Perform a storage query GET(q) to retrieve the RRBLOCK.

4. Verify and process the RRBLOCK and decrypt the BDATA contained in it as defined in Section 6.2.

Upon receiving the RRBLOCK from the storage, as part of verifying the provided signature, the resolver **MUST** check that the SHA-512 hash of the derived authoritative zone key zk' from the RRBLOCK matches the query q and that the block is not yet expired. If the signature does not match or the block is expired, the RRBLOCK **MUST** be ignored and, if applicable, the storage lookup GET(q) **MUST** continue to look for other RRBLOCKs.

7.3. Record Processing

Record processing occurs once a well-formed block has been decrypted. In record processing, only the valid records obtained are considered. To filter records by validity, the resolver **MUST** at least check the expiration time and the FLAGS field of the respective record. In particular, SHADOW and SUPPLEMENTAL flags can exclude the record from being considered. If the resolver encounters a record with the CRITICAL flag set and does not support the record type the resolution **MUST** be aborted and an error **MUST** be returned. The information that the critical record could not be processed **SHOULD** be returned in the error description. The implementation **MAY** choose not to return the reason for the failure, merely complicating troubleshooting for the user.

The next steps depend on the context of the name that is being resolved:

*Case 1: If the filtered record set consists of a single REDIRECT record, the remainder of the name is prepended to the REDIRECT data and the recursion is started again from the resulting name. Details are described in Section 7.3.1.

*Case 2: If the filtered record set consists exclusively of one or more GNS2DNS records resolution continues with DNS. Details are described in Section 7.3.2.

*Case 3: If the remainder of the name to be resolved is of the format "_SERVICE._PROTO" and the record set contains one or more matching BOX records, the records in the BOX records are the

final result and the recursion is concluded as described in Section 7.3.3.

*Case 4: If the current record set consist of a single delegation record, resolution of the remainder of the name is delegated to the target zone as described in Section 7.3.4.

*Case 5: If the remainder of the name to resolve is empty the record set is the final result. If any NICK records are in the final result set, it **MUST** be processed according to Section 7.3.5. Otherwise, the final result set is returned.

*Finally, if none of the above is applicable resolution fails and the resolver **MUST** return an empty record set.

7.3.1. REDIRECT

If the remaining name is empty and the desired record type is REDIRECT, in which case the resolution concludes with the REDIRECT record. If the rightmost label of the redirect name is the extension label (U+002B, "+"), resolution continues in GNS with the new name in the current zone. Otherwise, the resulting name is resolved via the default operating system name resolution process. This can in turn trigger a GNS name resolution process depending on the system configuration. In case resolution continues in DNS, the name **MUST** first be converted to an IDNA compliant representation [RFC5890].

In order to prevent infinite loops, the resolver **MUST** implement loop detection or limit the number of recursive resolution steps. The loop detection **MUST** be effective even if a REDIRECT found in GNS triggers subsequent GNS lookups via the default operating system name resolution process.

7.3.2. GNS2DNS

When a resolver encounters one or more GNS2DNS records and the remaining name is empty and the desired record type is GNS2DNS, the GNS2DNS records are returned.

Otherwise, it is expected that the resolver first resolves the IP addresses of the specified DNS name servers. The DNS name **MUST** be converted to an IDNA compliant representation [RFC5890] for resolution in DNS. GNS2DNS records **MAY** contain numeric IPv4 or IPv6 addresses, allowing the resolver to skip this step. The DNS server names might themselves be names in GNS or DNS. If the rightmost label of the DNS server name is the extension label (U+002B, "+"), the rest of the name is to be interpreted relative to the zone of the GNS2DNS record. If the DNS server name ends in a label representation of a zone key, the DNS server name is to be resolved against the GNS zone zk.

Multiple GNS2DNS records can be stored under the same label, in which case the resolver **MUST** try all of them. The resolver **MAY** try them in any order or even in parallel. If multiple GNS2DNS records are present, the DNS name **MUST** be identical for all of them. Otherwise, it is not clear which name the resolver is supposed to follow. If multiple DNS names are present the resolution fails and an appropriate error is **SHOULD** be returned to the application.

If there are DNSSEC DS records or any other records used to secure the connection with the DNS servers stored under the label, the DNS resolver **SHOULD** use them to secure the connection with the DNS server.

Once the IP addresses of the DNS servers have been determined, the DNS name from the GNS2DNS record is appended to the remainder of the name to be resolved, and resolved by querying the DNS name server(s). The synthesized name has to be converted to an IDNA compliant representation [RFC5890] for resolution in DNS. If such a conversion is not possible, the resolution **MUST** be aborted and an error **MUST** be returned. The information that the critical record could not be processed **SHOULD** be returned in the error description. The implementation **MAY** choose not to return the reason for the failure, merely complicating troubleshooting for the user.

As the DNS servers specified are possibly authoritative DNS servers, the GNS resolver **MUST** support recursive DNS resolution and **MUST NOT** delegate this to the authoritative DNS servers. The first successful recursive name resolution result is returned to the application. In addition, the resolver **SHOULD** return the queried DNS name as a supplemental LEHO record (see Section 5.3.1) with a relative expiration time of one hour.

Once the transition from GNS into DNS is made through a GNS2DNS record, there is no "going back". The (possibly recursive) resolution of the DNS name **MUST NOT** delegate back into GNS and should only follow the DNS specifications. For example, names contained in DNS CNAME records **MUST NOT** be interpreted by resolvers that support both DNS and GNS as GNS names.

GNS resolvers **SHOULD** offer a configuration option to disable DNS processing to avoid information leakage and provide a consistent security profile for all name resolutions. Such resolvers would return an empty record set upon encountering a GNS2DNS record during the recursion. However, if GNS2DNS records are encountered in the record set for the apex label and a GNS2DNS record is explicitly requested by the application, such records **MUST** still be returned, even if DNS support is disabled by the GNS resolver configuration.

7.3.3. BOX

When a BOX record is received, a GNS resolver must unbox it if the name to be resolved continues with "_SERVICE._PROTO". Otherwise, the BOX record is to be left untouched. This way, TLSA (and SRV) records do not require a separate network request, and TLSA records become inseparable from the corresponding address records.

7.3.4. Zone Delegation Records

When the resolver encounters a record of a supported zone delegation record type (such as PKEY or EDKEY) and the remainder of the name is not empty, resolution continues recursively with the remainder of the name in the GNS zone specified in the delegation record.

Whenever a resolver encounters a new GNS zone, it **MUST** check against the local revocation list whether the respective zone key has been revoked. If the zone key was revoked, the resolution **MUST** fail with an empty result set.

Implementations **MUST NOT** allow multiple different zone delegations under a single label. Implementations **MAY** support any subset of ztypes. Handling of Implementations **MUST NOT** process zone delegation for the apex label "@". Upon encountering a zone delegation record under this label, resolution fails and an error **MUST** be returned. The implementation **MAY** choose not to return the reason for the failure, merely impacting troubleshooting information for the user.

If the remainder of the name to resolve is empty and a record set was received containing only a single delegation record, the recursion is continued with the record value as authoritative zone and the apex label "@" as remaining name. Except in the case where the desired record type as specified by the application is equal to the ztype, in which case the delegation record is returned.

7.3.5. NICK

NICK records are only relevant to the recursive resolver if the record set in question is the final result which is to be returned to the application. The encountered NICK records can either be supplemental (see Section 5) or non-supplemental. If the NICK record is supplemental, the resolver only returns the record set if one of the non-supplemental records matches the queried record type. It is possible that one record set contains both supplemental and non-supplemental NICK records.

The differentiation between a supplemental and non-supplemental NICK record allows the application to match the record to the authoritative zone. Consider the following example:

```
Query: alice.example (type=A)
Result:
A: 192.0.2.1
NICK: eve (non-Supplemental)
```

In this example, the returned NICK record is non-supplemental. For the application, this means that the NICK belongs to the zone "alice.example" and is published under the apex label along with an A record. The NICK record is interpreted as: The zone defined by "alice.example" wants to be referred to as "eve". In contrast, consider the following:

```
Query: alice.example (type=AAAA)
Result:
AAAA: 2001:DB8::1
NICK: john (Supplemental)
```

In this case, the NICK record is marked as supplemental. This means that the NICK record belongs to the zone "example" and is published under the label "alice" along with an A record. The NICK record should be interpreted as: The zone defined by "example" wants to be referred to as "john". This distinction is likely useful for other records published as supplemental.

8. Internationalization and Character Encoding

All names in GNS are encoded in UTF-8 [RFC3629]. Labels **MUST** be canonicalized using Normalization Form C (NFC) [Unicode-UAX15]. This does not include any DNS names found in DNS records, such as CNAME record data, which is internationalized through the IDNA specifications [RFC5890].

9. Security and Privacy Considerations

9.1. Availability

In order to ensure availability of records beyond their absolute expiration times, implementations **MAY** allow to locally define relative expiration time values of records. Records can then be published recurringly with updated absolute expiration times by the implementation.

Implementations **MAY** allow users to manage private records in their zones that are not published in the storage. Private records are considered just like regular records when resolving labels in local zones, but their data is completely unavailable to non-local users.

9.2. Agility

The security of cryptographic systems depends on both the strength of the cryptographic algorithms chosen and the strength of the keys used with those algorithms. The security also depends on the engineering of the protocol used by the system to ensure that there are no non-cryptographic ways to bypass the security of the overall system. This is why developers of applications managing GNS zones **SHOULD** select a default ztype considered secure at the time of releasing the software. For applications targeting end users that are not expected to understand cryptography, the application developer **MUST NOT** leave the ztype selection of new zones to end users.

This document concerns itself with the selection of cryptographic algorithms used in GNS. The algorithms identified in this document are not known to be broken (in the cryptographic sense) at the current time, and cryptographic research so far leads us to believe that they are likely to remain secure into the foreseeable future. However, this is not necessarily forever, and it is expected that new revisions of this document will be issued from time to time to reflect the current best practices in this area.

In terms of crypto-agility, whenever the need for an updated cryptographic scheme arises to, for example, replace ECDSA over Ed25519 for PKEY records it can simply be introduced through a new record type. Zone administrators can then replace the delegation record type for future records. The old record type remains and zones can iteratively migrate to the updated zone keys. To ensure that implementations correctly generate an error message when encountering a ztype that they do not support, current and future delegation records must always have the CRITICAL flag set.

9.3. Cryptography

The following considerations provide background on the design choices of the ztypes specified in this document. When specifying new ztypes as per Section 4, the same considerations apply.

GNS PKEY zone keys use ECDSA over Ed25519. This is an unconventional choice, as ECDSA is usually used with other curves. However, standardized ECDSA curves are problematic for a range of reasons described in the Curve25519 and EdDSA papers [ed25519]. Using EdDSA directly is also not possible, as a hash function is used on the private key which destroys the linearity that the key blinding in GNS depends upon. We are not aware of anyone suggesting that using Ed25519 instead of another common curve of similar size would lower the security of ECDSA. GNS uses 256-bit curves because that way the

encoded (public) keys fit into a single DNS label, which is good for usability.

In order to ensure ciphertext indistinguishability, care must be taken with respect to the initialization vector in the counter block. In our design, the IV always includes the expiration time of the record block. When applications store records with relative expiration times, monotonicity is implicitly ensured because each time a block is published into the storage, its IV is unique as the expiration time is calculated dynamically and increases monotonically with the system time. Still, an implementation **MUST** ensure that when relative expiration times are decreased, the expiration time of the next record block **MUST** be after the last published block. For records where an absolute expiration time is used, the implementation **MUST** ensure that the expiration time is always increased when the record data changes. For example, the expiration time on the wire could be increased by a single microsecond even if the user did not request a change. In case of deletion of all resource records under a label, the implementation **MUST** keep track of the last absolute expiration time of the last published resource block. Implementations **MAY** define and use a special record type as a tombstone that preserves the last absolute expiration time, but then **MUST** take care to not publish a block with this record. When new records are added under this label later, the implementation **MUST** ensure that the expiration times are after the last published block. Finally, in order to ensure monotonically increasing expiration times the implementation **MUST** keep a local record of the last time obtained from the system clock, so as to construct a monotonic clock in case the system clock jumps backwards.

9.4. Abuse Mitigation

GNS names are UTF-8 strings. Consequently, GNS faces similar issues with respect to name spoofing as DNS does for internationalized domain names. In DNS, attackers can register similar sounding or looking names (see above) in order to execute phishing attacks. GNS zone administrators must take into account this attack vector and incorporate rules in order to mitigate it.

Further, DNS can be used to combat illegal content on the internet by having the respective domains seized by authorities. However, the same mechanisms can also be abused in order to impose state censorship. Avoiding that possibility is one of the motivations behind GNS. In GNS, TLDs are not enumerable. By design, the start zone of the resolver is defined locally and hence such a seizure is difficult and ineffective in GNS.

9.5. Zone Management

In GNS, zone administrators need to manage and protect their zone keys. Once a zone key is lost, it cannot be recovered or revoked. Revocation messages can be pre-calculated if revocation is required in case a zone key is lost. Zone administrators, and for GNS this includes end-users, are required to responsibly and diligently protect their cryptographic keys. GNS supports signing records in advance ("offline") in order to support processes which aim to protect private keys such as air gaps.

Similarly, users are required to manage their local start zone configuration. In order to ensure integrity and availability or names, users must ensure that their local start zone information is not compromised or outdated. It can be expected that the processing of zone revocations and an initial start zone is provided with a GNS implementation ("drop shipping"). Shipping an initial start zone configuration effectively establishes a root zone. Extension and customization of the zone is at the full discretion of the user.

While implementations following this specification will be interoperable, if two implementations connect to different storages they are mutually unreachable. This can lead to a state where a record exists in the global namespace for a particular name, but the implementation is not communicating with the storage and is hence unable to resolve it. This situation is similar to a split-horizon DNS configuration. Which storages are implemented usually depends on the application it is built for. The storage used will most likely depend on the specific application context using GNS resolution. For example, one application is the resolution of hidden services within the Tor network, which would suggest using Tor routers for storage. Implementations of "aggregated" storages are conceivable, but are expected to be the exception.

9.6. DHTs as Storage

This document does not specify the properties of the underlying storage which is required by any GNS implementation. It is important to note that the properties of the underlying storage are directly inherited by the GNS implementation. This includes both security as well as other non-functional properties such as scalability and performance. Implementers should take great care when selecting or implementing a DHT for use as storage in a GNS implementation. DHTs with reasonable security and performance properties exist [R5N]. It should also be taken into consideration that GNS implementations which build upon different DHT overlays are unlikely to be interoperable with each other.

9.7. Revocations

Zone administrators are advised to pre-generate zone revocations and to securely store the revocation information in case the zone key is lost, compromised or replaced in the future. Pre-calculated revocations can cease to be valid due to expirations or protocol changes such as epoch adjustments. Consequently, implementers and users must take precautions in order to manage revocations accordingly.

Revocation payloads do not include a 'new' key for key replacement. Inclusion of such a key would have two major disadvantages:

1. If a revocation is published after a private key was compromised, allowing key replacement would be dangerous: if an adversary took over the private key, the adversary could then broadcast a revocation with a key replacement. For the replacement, the compromised owner would have no chance to issue even a revocation. Thus, allowing a revocation message to replace a private key makes dealing with key compromise situations worse.

2. Sometimes, key revocations are used with the objective of changing cryptosystems. Migration to another cryptosystem by replacing keys via a revocation message would only be secure as long as both cryptosystems are still secure against forgery. Such a planned, non-emergency migration to another cryptosystem should be done by running zones for both cipher systems in parallel for a while. The migration would conclude by revoking the legacy zone key only once it is deemed no longer secure, and hopefully after most users have migrated to the replacement.

9.8. Zone Privacy

GNS does not support authenticated denial of existence of names within a zone. Record blocks are published in encrypted form using keys derived from the zone key and record label. Zone administrators should carefully consider if the label and zone key is public or if those should be used and considered as a shared secret. Unlike zone keys, labels can also be guessed by an attacker in the network observing queries and responses. Given a known and targeted zone key, the use of well known or easily guessable labels effectively results in general disclosure of the records to the public. If the labels and hence the records should be kept secret except to those knowing a secret label and the zone in which to look, the label must be chosen accordingly. It is recommended to then use a label with sufficient entropy as to prevent guessing attacks.

It should be noted that this attack on labels only applies if the zone key is somehow disclosed to the adversary. GNS itself does not disclose it during a lookup or when resource records are published as the zone keys are blinded beforehand. However, zone keys do become public during revocation.

9.9. Zone Governance

While DNS is distributed, in practice it relies on centralized, trusted registrars to provide globally unique names. As the awareness of the central role DNS plays on the Internet rises, various institutions are using their power (including legal means) to engage in attacks on the DNS, thus threatening the global availability and integrity of information on the Internet. While a wider discussion of this issue is out of scope for this document, analyses and investigations can be found in recent academic research works including [SecureNS].

GNS is designed to provide a secure, privacy-enhancing alternative to the DNS name resolution protocol, especially when censorship or manipulation is encountered. In particular, it directly addresses concerns in DNS with respect to query privacy. However, depending on the governance of the root zone, any deployment will likely suffer from the issues of a "Single Hierarchy with a Centrally Controlled Root" and "Distribution and Management of Root Servers" as raised in [RFC8324]. In DNS, those issues are a direct result from the centralized root zone governance at the Internet Corporation for Assigned Names and Numbers (ICANN) which allows it to provide globally unique names.

In GNS, start zones give users local authority over their preferred root zone governance. It enables users to replace or enhance a trusted root zone configuration provided by a third party (e.g. the implementer or a multi-stakeholder governance body like ICANN) with secure delegation of authority using local petnames while operating under a very strong adversary model. In combination with zTLDs, this provides users of GNS with a global, secure and memorable mapping without a trusted authority.

Any GNS implementation **MAY** provide a default governance model in the form of an initial start zone mapping.

9.10. Namespace Ambiguity

Technically, the GNS protocol can be used to resolve names in the namespace of the global DNS. However, this would require the respective governance bodies and stakeholders (e.g. IETF and ICANN) to standardize the use of GNS for this particular use case.

However, this capability implies that GNS names may be indistinguishable from DNS names in their respective common display format [RFC8499] or other special-use domain names [RFC6761] if a local start zone configuration maps suffixes from the global DNS to GNS zones. For applications, it is then ambiguous which name system should be used in order to resolve a given name. This poses a risk when trying to resolve a name through DNS when it is actually a GNS name. In such a case, the GNS name is likely to be leaked as part of the DNS resolution.

In order to prevent disclosure of queried GNS names it is **RECOMMENDED** that GNS-aware applications try to resolve a given name in GNS before any other method taking into account potential suffix-to-zone mappings and zTLDs. Suffix-to-zone mappings are expected to be configured by the user or local administrator and as such the resolution in GNS is in line with user expectations even if the name could also be resolved through DNS. If no suffix-to-zone mapping for the name exists and no zTLD is found, resolution **MAY** continue with other methods such as DNS. If a suffix-to-zone mapping for the name exists or the name ends with a zTLD, it **MUST** be resolved using GNS and resolution **MUST NOT** continue by any other means independent of the GNS resolution result.

Mechanisms such as the Name Service Switch (NSS) of Unix-like operating systems are an example of how such a resolution process can be implemented and used. It allows system administrators to configure host name resolution precedence and is integrated with the system resolver implementation.

The user or system administrator **MAY** configure one or more unique suffixes for all suffix-to-zone mappings. If this suffix is a special-use domain name for GNS or an unreserved DNS TLD, this prevents namespace ambiguity through local configuration.

10. GANA Considerations

GANA [GANA] manages the "GNU Name System Record Types" registry. Each entry has the following format:

* Name: The name of the record type (case-insensitive ASCII string, restricted to alphanumeric characters. For zone delegation records, the assigned number represents the ztype value of the zone.

* Number: 32-bit, above 65535

* Comment: Optionally, a brief English text describing the purpose of the record type (in UTF-8)

*Contact: Optionally, the contact information of a person to contact for further information.

*References: Optionally, references describing the record type (such as an RFC)

The registration policy for this registry is "First Come First Served". This policy is modeled on that described in [RFC8126], and describes the actions taken by GANA:

Adding new records is possible after expert review, using a first-come-first-served policy for unique name allocation. Experts are responsible to ensure that the chosen "Name" is appropriate for the record type. The registry will assign a unique number for the entry.

The current contact(s) for expert review are reachable at gns-registry@gnunet.org.

Any request **MUST** contain a unique name and a point of contact. The contact information **MAY** be added to the registry given the consent of the requester. The request **MAY** optionally also contain relevant references as well as a descriptive comment as defined above.

GANA has assigned numbers for the record types defined in this specification in the "GNU Name System Record Types" registry as listed in Figure 24.

Number	Name	Contact	References	Comment
65536	PKEY	N/A	[This.I-D]	GNS zone delegation (PKEY)
65537	NICK	N/A	[This.I-D]	GNS zone nickname
65538	LEHO	N/A	[This.I-D]	GNS legacy hostname
65540	GNS2DNS	N/A	[This.I-D]	Delegation to DNS
65541	BOX	N/A	[This.I-D]	Boxed records
65551	REDIRECT	N/A	[This.I-D]	Redirection record.
65556	EDKEY	N/A	[This.I-D]	GNS zone delegation (EDKEY)

Figure 24: The GANA Resource Record Registry.

GANA has assigned signature purposes in its "GNUnet Signature Purpose" registry as listed in Figure 25.

Purpose	Name	References	Comment
3	GNS_REVOCATION	[This.I-D]	GNS zone key revocation
15	GNS_RECORD_SIGN	[This.I-D]	GNS record set signature

Figure 25: Requested Changes in the GANA GNUnet Signature Purpose Registry.

11. IANA Considerations

This document makes no requests for IANA action. This section may be removed on publication as an RFC.

12. Implementation and Deployment Status

There are two implementations conforming to this specification written in C and Go, respectively. The C implementation as part of GNUnet [GNUnetGNS] represents the original and reference implementation. The Go implementation [GoGNS] demonstrates how two implementations of GNS are interoperable if they are built on top of the same underlying DHT storage.

Currently, the GNUnet peer-to-peer network [GNUnet] is an active deployment of GNS on top of its [R5N] DHT. The [GoGNS] implementation uses this deployment by building on top of the GNUnet DHT services available on any GNUnet peer. It shows how GNS implementations can attach to this existing deployment and participate in name resolution as well as zone publication.

The self-sovereign identity system re:claimID [reclaim] is using GNS in order to selectively share identity attributes and attestations with third parties.

The Ascension tool [Ascension] facilitates the migration of DNS zones to GNS zones by translating information retrieved from a DNS zone transfer into a GNS zone.

13. Acknowledgements

The authors thank all reviewers for their comments. In particular, we thank D. J. Bernstein, S. Bortzmeyer, A. Farrel, E. Lear and R. Salz for their insightful and detailed technical reviews. We thank J. Yao and J. Klensin for the internationalization reviews. We thank NLnet and NGI DISCOVERY for funding work on the GNU Name System.

14. Normative References

[RFC1034] Mockapetris, P V., "Domain names - concepts and facilities", STD 13, RFC 1034, DOI 10.17487/RFC1034, November 1987, <https://www.rfc-editor.org/info/rfc1034>.

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

[RFC2782] Gulbrandsen, A., Vixie, P., and L. Esibov, "A DNS RR for specifying the location of services (DNS SRV)", RFC 2782, DOI 10.17487/RFC2782, February 2000, <https://www.rfc-editor.org/info/rfc2782>.

[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>.

[RFC3629] Yergeau, F., "UTF-8, a transformation format of ISO 10646", STD 63, RFC 3629, DOI 10.17487/RFC3629, November 2003, <https://www.rfc-editor.org/info/rfc3629>.

[RFC3686] Housley, R., "Using Advanced Encryption Standard (AES) Counter Mode With IPsec Encapsulating Security Payload (ESP)", RFC 3686, DOI 10.17487/RFC3686, January 2004, <https://www.rfc-editor.org/info/rfc3686>.

[RFC3826] Blumenthal, U., Maino, F., and K. McCloghrie, "The Advanced Encryption Standard (AES) Cipher Algorithm in the SNMP User-based Security Model", RFC 3826, DOI 10.17487/RFC3826, June 2004, <https://www.rfc-editor.org/info/rfc3826>.

[RFC5237] Arkko, J. and S. Bradner, "IANA Allocation Guidelines for the Protocol Field", BCP 37, RFC 5237, DOI 10.17487/RFC5237, February 2008, <https://www.rfc-editor.org/info/rfc5237>.

[RFC5869] Krawczyk, H. and P. Eronen, "HMAC-based Extract-and-Expand Key Derivation Function (HKDF)", RFC 5869, DOI 10.17487/RFC5869, May 2010, <https://www.rfc-editor.org/info/rfc5869>.

[RFC5890] Klensin, J., "Internationalized Domain Names for Applications (IDNA): Definitions and Document Framework", RFC 5890, DOI 10.17487/RFC5890, August 2010, <https://www.rfc-editor.org/info/rfc5890>.

[RFC5895] Resnick, P. and P. Hoffman, "Mapping Characters for Internationalized Domain Names in Applications (IDNA) 2008", RFC 5895, DOI 10.17487/RFC5895, September 2010, <https://www.rfc-editor.org/info/rfc5895>.

[RFC6234] Eastlake 3rd, D. and T. Hansen, "US Secure Hash Algorithms (SHA and SHA-based HMAC and HKDF)", RFC 6234, DOI 10.17487/RFC6234, May 2011, <https://www.rfc-editor.org/info/rfc6234>.

[RFC6895] Eastlake 3rd, D., "Domain Name System (DNS) IANA Considerations", BCP 42, RFC 6895, DOI 10.17487/RFC6895, April 2013, <https://www.rfc-editor.org/info/rfc6895>.

[RFC6979] Pornin, T., "Deterministic Usage of the Digital Signature Algorithm (DSA) and Elliptic Curve Digital Signature Algorithm (ECDSA)", RFC 6979, DOI 10.17487/RFC6979, August 2013, <https://www.rfc-editor.org/info/rfc6979>.

[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>.

[RFC8032] Josefsson, S. and I. Liusvaara, "Edwards-Curve Digital Signature Algorithm (EdDSA)", RFC 8032, DOI 10.17487/RFC8032, January 2017, <https://www.rfc-editor.org/info/rfc8032>.

[RFC8126] Cotton, M., Leiba, B., and T. Narten, "Guidelines for Writing an IANA Considerations Section in RFCs", BCP 26, RFC 8126, DOI 10.17487/RFC8126, June 2017, <https://www.rfc-editor.org/info/rfc8126>.

[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>.

[RFC8499] Hoffman, P., Sullivan, A., and K. Fujiwara, "DNS Terminology", BCP 219, RFC 8499, DOI 10.17487/RFC8499, January 2019, <https://www.rfc-editor.org/info/rfc8499>.

[RFC9106] Biryukov, A., Dinu, D., Khovratovich, D., and S. Josefsson, "Argon2 Memory-Hard Function for Password Hashing and Proof-of-Work Applications", RFC 9106, DOI 10.17487/RFC9106, September 2021, <https://www.rfc-editor.org/info/rfc9106>.

[GANA] GNUnet e.V., "GNUnet Assigned Numbers Authority (GANA)", April 2020, <https://gana.gnunet.org/>.

[MODES] Dworkin, M., "Recommendation for Block Cipher Modes of Operation: Methods and Techniques", December 2001, <https://doi.org/10.6028/NIST.SP.800-38A>.

[CrockfordB32] Douglas, D., "Base32", March 2019, <https://www.crockford.com/base32.html>.

[XSalsa20] Bernstein, D., "Extending the Salsa20 nonce", 2011, <https://cr.yp.to/snuffle/xsalsa-20110204.pdf>.

[Unicode-UAX15]
The Unicode Consortium, "Unicode Standard Annex #15: Unicode Normalization Forms, Revision 31", September 2009, <http://www.unicode.org/reports/tr15/tr15-31.html>.

[Unicode-UTS46] The Unicode Consortium, "Unicode Technical Standard #46: Unicode IDNA Compatibility Processing, Revision 27", August 2021, <https://www.unicode.org/reports/tr46>.

15. Informative References

[RFC1928] Leech, M., Ganis, M., Lee, Y., Kuris, R., Koblas, D., and L. Jones, "SOCKS Protocol Version 5", RFC 1928, DOI 10.17487/RFC1928, March 1996, <https://www.rfc-editor.org/info/rfc1928>.

[RFC4033] Arends, R., Austein, R., Larson, M., Massey, D., and S. Rose, "DNS Security Introduction and Requirements", RFC 4033, DOI 10.17487/RFC4033, March 2005, <https://www.rfc-editor.org/info/rfc4033>.

[RFC6066] Eastlake 3rd, D., "Transport Layer Security (TLS) Extensions: Extension Definitions", RFC 6066, DOI 10.17487/RFC6066, January 2011, <https://www.rfc-editor.org/info/rfc6066>.

[RFC7363] Maenpaa, J. and G. Camarillo, "Self-Tuning Distributed Hash Table (DHT) for REsource LOcation And Discovery (RELOAD)", RFC 7363, DOI 10.17487/RFC7363, September 2014, <https://www.rfc-editor.org/info/rfc7363>.

[RFC8324] Klensin, J., "DNS Privacy, Authorization, Special Uses, Encoding, Characters, Matching, and Root Structure: Time for Another Look?", RFC 8324, DOI 10.17487/RFC8324, February 2018, <https://www.rfc-editor.org/info/rfc8324>.

[RFC8806] Kumari, W. and P. Hoffman, "Running a Root Server Local to a Resolver", RFC 8806, DOI 10.17487/RFC8806, June 2020, <https://www.rfc-editor.org/info/rfc8806>.

[RFC6761] Cheshire, S. and M. Krochmal, "Special-Use Domain Names", RFC 6761, DOI 10.17487/RFC6761, February 2013, <https://www.rfc-editor.org/info/rfc6761>.

[Tor224] Goulet, D., Kadianakis, G., and N. Mathewson, "Next-Generation Hidden Services in Tor", November 2013,

<https://gitweb.torproject.org/torspec.git/tree/proposals/224-rend-spec-ng.txt#n2135>.

[SDSI] Rivest, R. and B. Lampson, "SDSI - A Simple Distributed Security Infrastructure", April 1996, <http://people.csail.mit.edu/rivest/Sdsi10.ps>.

[Kademlia] Maymounkov, P. and D. Mazieres, "Kademlia: A peer-to-peer information system based on the xor metric.", 2002, <http://css.csail.mit.edu/6.824/2014/papers/kademlia.pdf>.

[ed25519] Bernstein, D., Duif, N., Lange, T., Schwabe, P., and B. Yang, "High-Speed High-Security Signatures", 2011, <https://ed25519.cr.yp.to/ed25519-20110926.pdf>.

[GNS] Wachs, M., Schanzenbach, M., and C. Grothoff, "A Censorship-Resistant, Privacy-Enhancing and Fully Decentralized Name System", 2014, <https://sci-hub.st/10.1007/978-3-319-12280-9_9>.

[R5N] Evans, N. S. and C. Grothoff, "R5N: Randomized recursive routing for restricted-route networks", 2011, <https://sci-hub.st/10.1109/ICNSS.2011.6060022>.

[SecureNS] Grothoff, C., Wachs, M., Ermert, M., and J. Appelbaum, "Towards secure name resolution on the Internet", 2018, <https://sci-hub.st/https://doi.org/10.1016/j.cose.2018.01.018>.

[GNUnetGNS] GNUnet e.V., "The GNUnet GNS Implementation", <https://git.gnunet.org/gnunet.git/tree/src/gns>.

[Ascension] GNUnet e.V., "The Ascension Implementation", <https://git.gnunet.org/ascension.git>.

[GNUnet] GNUnet e.V., "The GNUnet Project", <https://gnunet.org>.

[reclaim] GNUnet e.V., "The GNUnet Project", <https://reclaim.gnunet.org>.

[GoGNS] Fix, B., "The Go GNS Implementation", <https://github.com/bfix/gnunet-go/tree/master/src/gnunet/service/gns>.

[nsswitch] GNU Project, "System Databases and Name Service Switch", <https://www.gnu.org/software/libc/manual/html_node/Name-Service-Switch.html>.

Appendix A. Usage and Migration

This section outlines a number of specific use cases which may help readers of the technical specification to understand the protocol better. The considerations below are not meant to be normative for the GNS protocol in any way. Instead, they are provided in order to give context and to provide some background on what the intended use of the protocol is by its designers. Further, this section contains pointers to migration paths.

A.1. Zone Dissemination

In order to become a zone owner, it is sufficient to generate a zone key and a corresponding secret key using a GNS implementation. At this point, the zone owner can manage GNS resource records in a local zone database. The resource records can then be published by a GNS implementation as defined in Section 6. For other users to resolve the resource records, respective zone information must be disseminated first. The zone owner may decide to make the zone key and labels known to a selected set of users only or to make this information available to the general public.

Sharing zone information directly with specific users not only allows to potentially preserve zone and record privacy, but also allows the zone owner and the user to establish strong trust relationships. For example, a bank may send a customer letter with a QR code which contains the GNS zone of the bank. This allows the user to scan the QR code and establish a strong link to the zone of the bank and with it, for example, the IP address of the online banking web site.

Most Internet services likely want to make their zones available to the general public as efficiently as possible. First, it is reasonable to assume that zones which are commanding high levels of reputation and trust are likely included in the default suffix-to-zone mappings of implementations. Hence dissemination of a zone through delegation under such zones can be a viable path in order to disseminate a zone publicly. For example, it is conceivable that organizations such as ICANN or country-code top-level domain registrars also manage GNS zones and offer registration or delegation services.

Following best practices in particularly those relating to security and abuse mitigation are methods which allow zone owners and aspiring registrars to gain a good reputation and eventually trust. This includes, of course, diligent protection of private zone key material. Formalizing such best practices is out of scope of this specification and should be addressed in a separate document and take Section 9 into account.

A.2. Start Zone Configuration

A user is expected to install a GNS implementation if it is not already provided through other means such as the operating system or the browser. It is likely that the implementation ships with a default start zone configuration. This means that the user is able to resolve GNS names ending on a zTLD or ending on any suffix-to-name mapping that is part of the default start zone configuration. At this point the user may delete or otherwise modify the implementation's default configuration:

Deletion of suffix-to-zone mappings may become necessary of the zone owner referenced by the mapping has lost the trust of the user. For example, this could be due to lax registration policies resulting in phishing activities. Modification and addition of new mappings are means to heal the namespace perforation which would occur in the case of a deletion or to simply establish a strong direct trust relationship. However, this requires the user's knowledge of the respective zone keys. This information must be retrieved out of band, as illustrated in Appendix A.1: A bank may send the user a letter with a QR code which contains the GNS zone of the bank. The user scans the QR code and adds a new suffix-to-name mapping using a chosen local name for his bank. Other examples include scanning zone information off the device of a friend, from a storefront, or an advertisement. The level of trust in the respective zone is contextual and likely varies from user to user. Trust in a zone provided through a letter from a bank which may also include a credit card is certainly different from a zone found on a random advertisement in the streets. However, this trust is immediately tangible to the user and can be reflected in the local naming as well.

User clients should facilitate the modification of the start zone configuration, for example by providing a QR code reader or other import mechanisms. Implementations are ideally implemented according to best practices and addressing applicable points from Section 9. Formalizing such best practices is out of scope of this specification.

A.3. Globally Unique Names and the Web

HTTP virtual hosting and TLS Server Name Indication are common use cases on the Web. HTTP clients supply a DNS name in the HTTP "Host"-header or as part of the TLS handshake, respectively. This allows the HTTP server to serve the indicated virtual host with a matching TLS certificate. The global uniqueness of DNS names are a prerequisite of those use cases.

Not all GNS names are globally unique. But, any resource record in GNS can be represented as a concatenation of of a GNS label and the zTLD of the zone. While not human-readable, this globally unique GNS name can be leveraged in order to facilitate the same use cases. Consider the GNS name "www.example.gns" entered in a GNS-aware HTTP client. At first, "www.example.gns" is resolved using GNS yielding a record set. Then, the HTTP client determines the virtual host as follows:

If there is a LEHO record (Section 5.3.1) containing "www.example.com" in the record set, then the HTTP client uses this as the value of the "Host"-header field of the HTTP request:

```
GET / HTTP/1.1
Host: www.example.com
```

If there is no LEHO record in the record set, then the HTTP client tries to find the zone of the record and translates the GNS name into a globally unique zTLD-representation before using it in the "Host"-header field of the HTTP request:

```
GET / HTTP/1.1
Host: www.000G0037FH3QTBCK15Y8BCCNRVWPV17ZC7TSGB1C9ZG2TPGHZVFV1GMG3W
```

In order to determine the canonical representation of the record with a zTLD, at most two queries are required: First, it must be checked whether "www.example.gns" itself points to a zone delegation record which would imply that the record set which was originally resolved is published under the apex label. If it does, the unique GNS name is simply the zTLD representation of the delegated zone:

```
GET / HTTP/1.1
Host: 000G0037FH3QTBCK15Y8BCCNRVWPV17ZC7TSGB1C9ZG2TPGHZVFV1GMG3W
```

If it does not, the unique GNS name is the concatenation of the label "www" and the zTLD representation of the zone as given in the example above. In any case, this representation is globally unique. As such, it can be configured by the HTTP server administrator as a virtual host name and respective certificates may be issued.

If the HTTP client is a browser, the use of a unique GNS name for virtual hosting or TLS SNI does not necessarily have to be shown to the user. For example, the name in the URL bar may remain as "www.example.gnu" even if the used unique name differs.

A.4. Migration Paths

DNS resolution is built into a variety of existing software components. Most significantly operating systems and HTTP clients.

This section illustrates possible migration paths for both in order to enable "legacy" applications to resolve GNS names.

One way to efficiently facilitate the resolution of GNS names are GNS-enabled DNS server implementations. Local DNS queries are thereby either rerouted or explicitly configured to be resolved by a "DNS-to-GNS" server that runs locally. This DNS server tries to interpret any incoming query for a name as a GNS resolution request. If no start zone can be found for the name and it does not end in a zTLD, the server tries to resolve the name in DNS. Otherwise, the name is resolved in GNS. In the latter case, the resulting record set is converted to a DNS answer packet and is returned accordingly. An implementation of a DNS-to-GNS server can be found in [GNUnet].

A similar approach is to use operating systems extensions such as the name service switch [nsswitch]. It allows the system administrator to configure plugins which are used for hostname resolution. A GNS name service switch plugin can be used in a similar fashion as the "DNS-to-GNS" server. An implementation of a glibc-compatible nsswitch plugin for GNS can be found in [GNUnet].

The methods above are usually also effective for HTTP client software. However, HTTP clients are commonly used in combination with TLS. TLS certificate validation and in particular server name indication (SNI) requires additional logic in HTTP clients when GNS names are in play (Appendix A.3). In order to transparently enable this functionality for migration purposes, a local GNS-aware SOCKS5 proxy [RFC1928] can be configured to resolve domain names. The SOCKS5 proxy, similar to the DNS-to-GNS server, is capable of resolving both GNS and DNS names. In the event of a TLS connection request with a GNS name, the SOCKS5 proxy can act as a man-in-the-middle, terminating the TLS connection and establishing a secure connection against the requested host. In order to establish a secure connection, the proxy may use LEHO and TLSA records stored in the record set under the GNS name. The proxy must provide a locally trusted certificate for the GNS name to the HTTP client which usually requires the generation and configuration of a local trust anchor in the browser. An implementation of this SOCKS5 proxy can be found in [GNUnet].

Appendix B. Example flows

B.1. AAAA Example Resolution

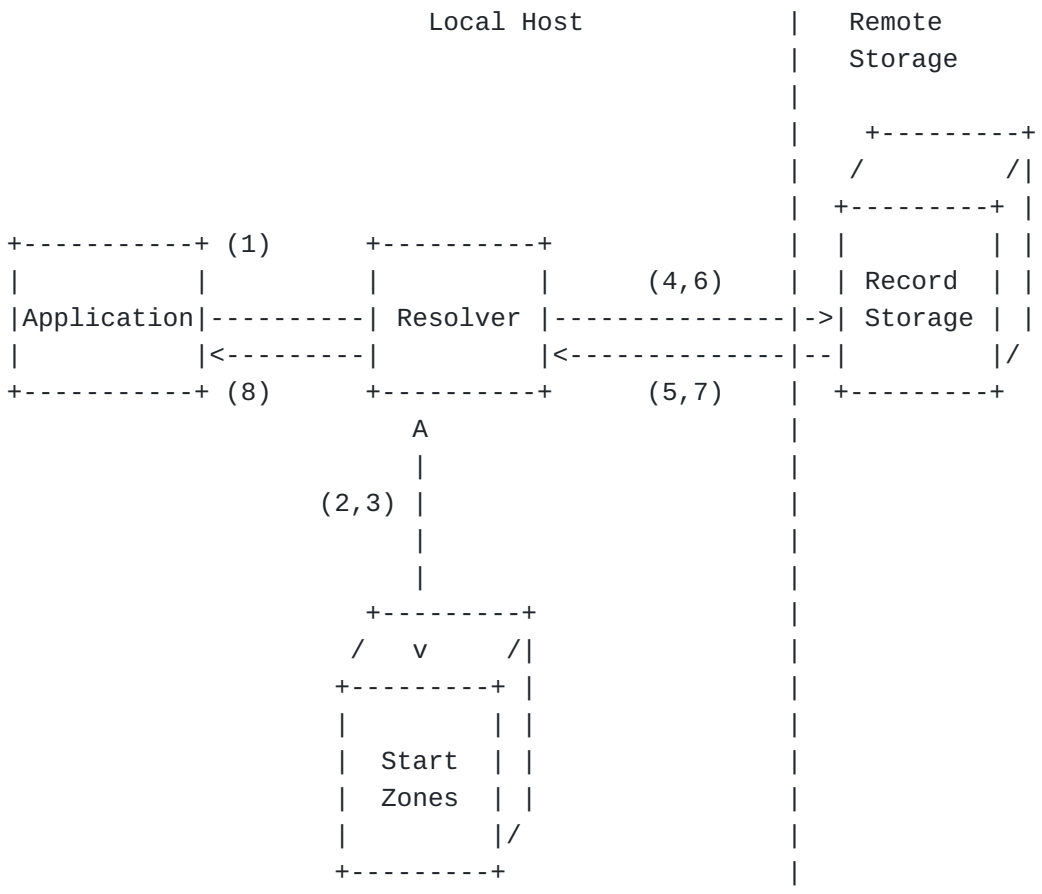

Figure 26: Example resolution of an IPv6 address.

1. Lookup AAAA record for name: www.example.gns.

2. Determine start zone for www.example.gns.

3. Start zone: zk0 - Remainder: www.example.

4. Calculate q0=SHA512(ZKDF(zk0, "example")) and initiate GET(q0).

5. Retrieve and decrypt RRBLOCK consisting of a single PKEY record containing zk1.

6. Calculate q1=SHA512(ZKDF(zk1, "www")) and initiate GET(q1).

7. Retrieve RRBLOCK consisting of a single AAAA record containing the IPv6 address 2001:db8::1.

8. Return record set to application

B.2. REDIRECT Example Resolution

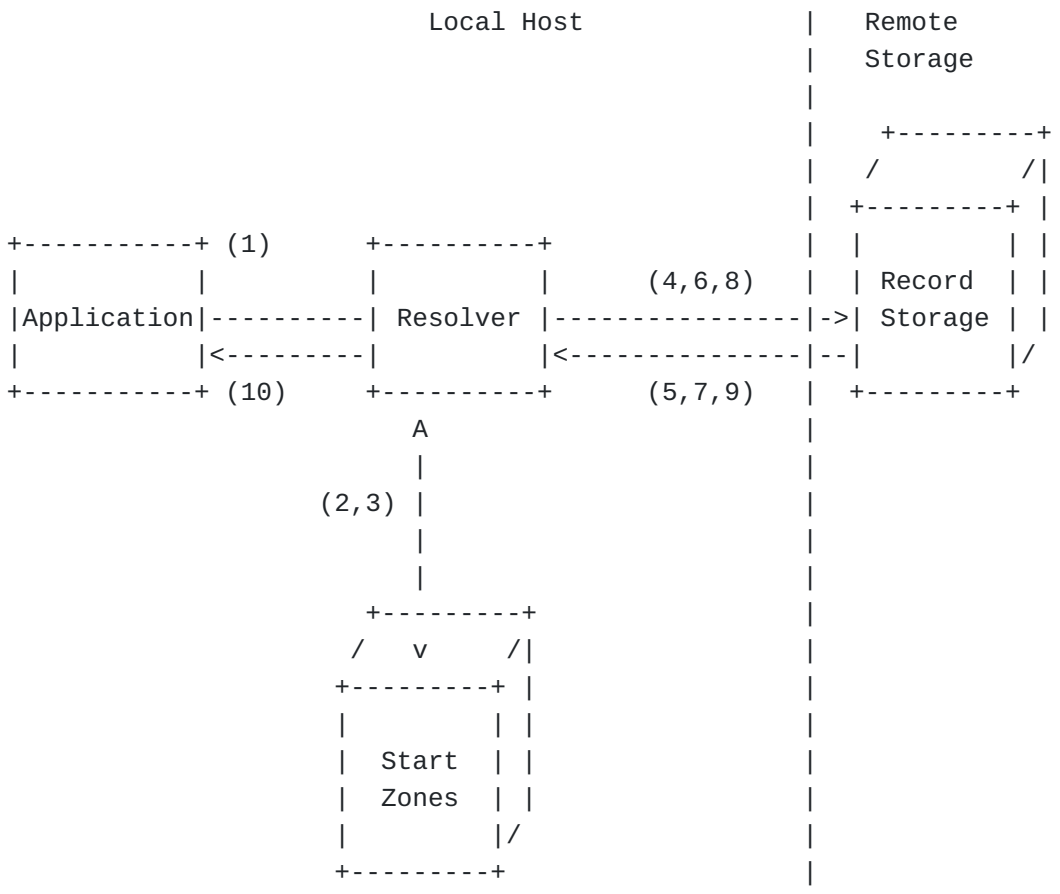

Figure 27: Example resolution of an IPv6 address with redirect.

1. Lookup AAAA record for name: www.example.tld.

2. Determine start zone for www.example.tld.

3. Start zone: zk0 - Remainder: www.example.

4. Calculate q0=SHA512(ZKDF(zk0, "example")) and initiate GET(q0).

5. Retrieve and decrypt RRBLOCK consisting of a single REDIRECT record containing zk1.

6. Calculate q1=SHA512(ZKDF(zk1, "www")) and initiate GET(q1).

7. Retrieve and decrypt RRBLOCK consisting of a single REDIRECT record containing www2.+.

8. Calculate q2=SHA512(ZKDF(zk1, "www2")) and initiate GET(q2).

9. Retrieve and decrypt RRBLOCK consisting of a single AAAA record containing the IPv6 address 2001:db8::1.

10. Return record set to application.

B.3. GNS2DNS Example Resolution

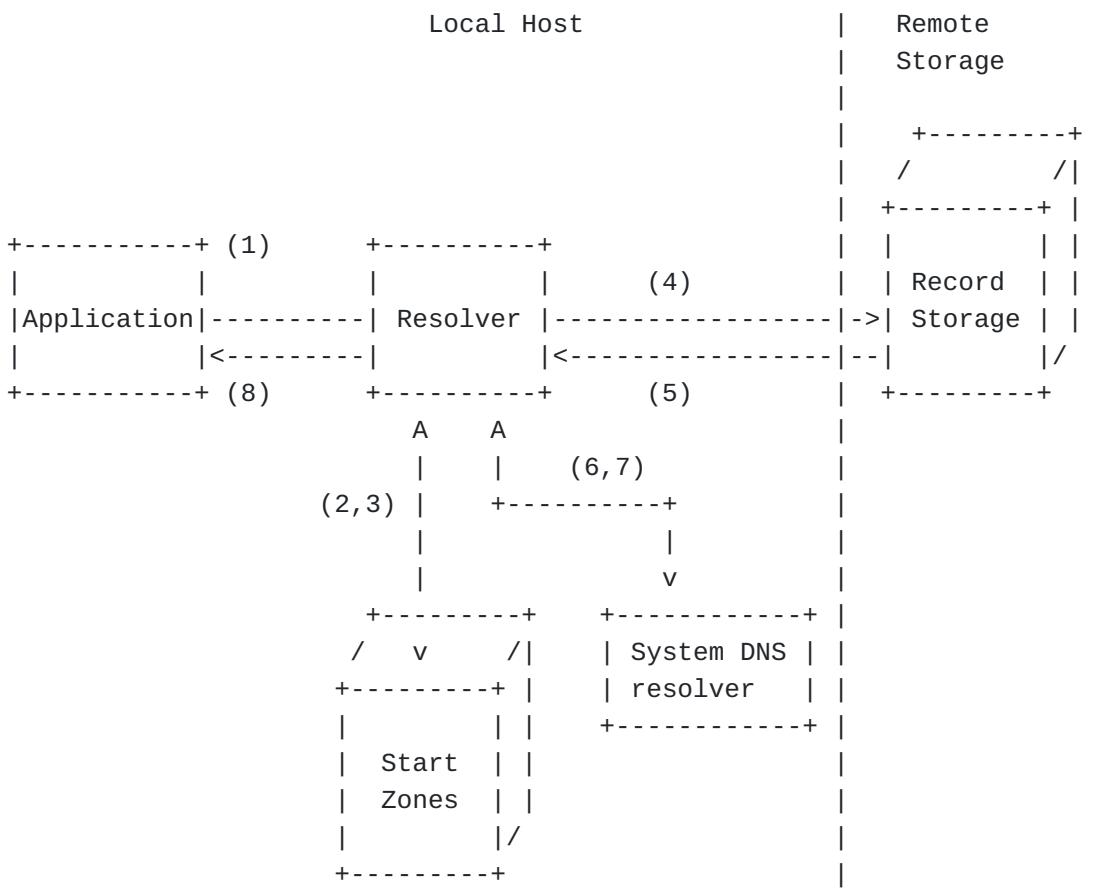

Figure 28: Example resolution of an IPv6 address with DNS handover.

1. Lookup AAAA record for name: www.example.gnu

2. Determine start zone for www.example.gnu.

3. Start zone: zk0 - Remainder: www.example.

4. Calculate q0=SHA512(ZKDF(zk0, "example")) and initiate GET(q0).

5. Retrieve and decrypt RRBLOCK consisting of a single GNS2DNS record containing the name example.com and the DNS server IPv4 address 192.0.2.1.

6. Use system resolver to lookup an AAAA record for the DNS name www.example.com.

7. Retrieve a DNS reply consisting of a single AAAA record containing the IPv6 address 2001:db8::1.

8. Return record set to application.

Appendix C. Base32GNS

This table defines the encode symbol and decode symbol for a given symbol value. It can be used to implement the encoding by reading it as: A character "A" or "a" is decoded to a 5 bit value 10 when decoding. A 5 bit block with a value of 18 is encoded to the character "J" when encoding. If the bit length of the byte string to encode is not a multiple of 5 it is padded to the next multiple with zeroes. In order to further increase tolerance for failures in character recognition, the letter "U" **MUST** be decoded to the same value as the letter "V" in Base32GNS.

Symbol Value	Decode Symbol	Encode Symbol
0	0 O o	0
1	1 I i L l	1
2	2	2
3	3	3
4	4	4
5	5	5
6	6	6
7	7	7
8	8	8
9	9	9
10	A a	A
11	B b	B
12	C c	C
13	D d	D
14	E e	E
15	F f	F
16	G g	G
17	H h	H
18	J j	J
19	K k	K
20	M m	M
21	N n	N
22	P p	P
23	Q q	Q
24	R r	R
25	S s	S
26	T t	T
27	V v U u	V
28	W w	W
29	X x	X
30	Y y	Y
31	Z z	Z

Figure 29: The Base32GNS Alphabet Including the Additional U Encode Symbol.

Appendix D. Test Vectors

The following are test vectors for the Base32GNS encoding used for zTLDs. The strings are encoded without the zero terminator.

```
Base32GNS-Encode:
  Input string: "Hello World"
  Output string: "91JPRV3F41BPYWKCCG"

  Input bytes: 474e55204e616d652053797374656d
  Output string: "8X75A82EC5PPA82KF5SQ8SBD"

Base32GNS-Decode:
  Input string: "91JPRV3F41BPYWKCCG"
  Output string: "Hello World"

  Input string: "91JPRU3F41BPYWKCCG"
  Output string: "Hello World"
```

The following test vectors can be used by implementations to test for conformance with this specification. The test vectors include record sets with a variety of record types and flags for both PKEY and EDKEY zones. Unless indicated otherwise, the test vectors are provided as hex byte values. This includes labels as some test vectors contain UTF-8 multibyte characters to demonstrate internationalized labels.

Zone private key (d, big-endian):
50d7b652a4efeadf
f37396909785e595
2171a02178c8e7d4
50fa907925fafd98

Zone identifier (ztype|zkey):
00010000677c477d
2d93097c85b195c6
f96d84ff61f5982c
2c4fe02d5a11fedf
b0c2901f

zTLD:
000G0037FH3QTBCK15Y8BCCNRVWPV17ZC7TSGB1C9ZG2TPGHZVFV1GMG3W

Label:
7465737464656c65
676174696f6e

Number of records (integer): 1

Record #0 := (
EXPIRATION:
0008c06fb9281580

DATA_SIZE:
0020

TYPE:
00010000

FLAGS: 0001

DATA:
21e3b30ff93bc6d3
5ac8c6e0e13afdff
794cb7b44bbbc748
d259d0a0284dbe84

)

RDATA:
0008c06fb9281580
0020000100010000
21e3b30ff93bc6d3
5ac8c6e0e13afdff
794cb7b44bbbc748
d259d0a0284dbe84

Encryption NONCE|EXPIRATION|BLOCK COUNTER:
e90a00610008c06f
b928158000000001

Encryption key (K):
864e7138eae7fd91
a30136899c132b23
acebdb2cef43cb19
f6bf55b67db9b3b3

Storage key (q):
4adc67c5ecee9f76
986abd71c2224a3d
ce2e917026c9a09d
fd44cef3d20f55a2
7332725a6c8afbbb
b0f7ec9af1cc4264
1299406b04fd9b5b
5791f86c4b08d5f4

BDATA:
41dc7b5f2176ba59
1998afb9e3c82579
5050afc4b53d68e4
1ed921da89de51e7
da35a295b59c2b8a
aea4399148d50cff

RRBLOCK:
000000a000010000
182bb636eda79f79
5711bc2708adbb24
2a60446ad3c30803
121d03d348b7ceb6
01beab944aff7ccc
51bffb212779c341
87660c625d1ceb59
d5a0a9a2dfe4072d
0f08cd2ab1e9ed63
d3898ff732521b57
317a6c4950e1984d
74df015f9eb72c4a
0008c06fb9281580
41dc7b5f2176ba59
1998afb9e3c82579
5050afc4b53d68e4
1ed921da89de51e7
da35a295b59c2b8a

```
aea4399148d50cff

Zone private key (d, big-endian):
50d7b652a4efeadf
f37396909785e595
2171a02178c8e7d4
50fa907925fafd98

Zone identifier (ztype|zkey):
00010000677c477d
2d93097c85b195c6
f96d84ff61f5982c
2c4fe02d5a11fedf
b0c2901f

zTLD:
000G0037FH3QTBCK15Y8BCCNRVWPV17ZC7TSGB1C9ZG2TPGHZVFV1GMG3W

Label:
e5a4a9e4b88be784
a1e695b5

Number of records (integer): 3

Record #0 := (
EXPIRATION:
0008c06fb9281580

DATA_SIZE:
0010

TYPE:
0000001c

FLAGS: 0000

DATA:
0000000000000000
00000000deadbeef

)

Record #1 := (
EXPIRATION:
00b00f81b7449b40

DATA_SIZE:
0006

TYPE:
```

```
00010001

FLAGS: 8000

DATA:
e6849be7a7b0

)

Record #2 := (
EXPIRATION:
000000016b597108

DATA_SIZE:
000b

TYPE:
00000010

FLAGS: 4004

DATA:
48656c6c6f20576f
726c64

)

RDATA:
0008c06fb9281580
001000000000001c
0000000000000000
00000000deadbeef
00b00f81b7449b40
0006800000010001
e6849be7a7b00000
00016b597108000b
4004000000104865
6c6c6f20576f726c
6400000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000

Encryption NONCE|EXPIRATION|BLOCK COUNTER:
ee9633c10005db3b
cdbd617c00000001

Encryption key (K):
```

fb3ab5de23bddae1
997aaf7b92c2d271
51408b77af7a41ac
79057c4df5383d01

Storage key (q):
aff0ad6a44097368
429ac476dfa1f34b
ee4c36e7476d07aa
6463ff20915b1005
c0991def91fc3e10
909f8702c0be4043
6778c711f2ca47d5
5cf0b54d235da977

BDATA:
f8c5e4badf1649d4
04da64df7d9d285f
4072a5f7a2547d56
74227e9b188eb2bb
6b34532f61e08ffb
d5bdea3741e60967
b687f8d8c44c8f6f
120a0f980f393b21
60407be128a74a51
51d6370be56a86ea
e32fdc217596b13f
6fea3fcfea0f4deb
881a25458f505a8f
cfca62d6da56073f
497698613475a1ad
14b7877f9455b0ec

RRBLOCK:
000000f000010000
a51296df757ee275
ca118d4f07fa7aae
5508bcf512aa4112
1429d4a0de9d057e
05c095040b10c7f8
187aa5da12287d1c
2910ff04d6f50af1
fa95382e9f007f75
098f620d1ff7c971
28f40d7458a2d3c7
f048ca3820064bdd
ee9413e9548ec994
0005db3bcdbd617c
f8c5e4badf1649d4

04da64df7d9d285f
4072a5f7a2547d56
74227e9b188eb2bb
6b34532f61e08ffb
d5bdea3741e60967
b687f8d8c44c8f6f
120a0f980f393b21
60407be128a74a51
51d6370be56a86ea
e32fdc217596b13f
6fea3fcfea0f4deb
881a25458f505a8f
cfca62d6da56073f
497698613475a1ad
14b7877f9455b0ec

Zone private key (d):
5af7020ee1916032
8832352bbc6a68a8
d71a7cbe1b929969
a7c66d415a0d8f65

Zone identifier (ztype|zkey):
000100143cf4b924
032022f0dc505814
53b85d93b047b63d
446c5845cb48445d
db96688f

zTLD:
000G051WYJWJ80S04BRDRM2R2H9VGQCKP13VCFA4DHC4BJT88HEXQ5K8HW

Label:
7465737464656c65
676174696f6e

Number of records (integer): 1

Record #0 := (
EXPIRATION:
0008c06fb9281580

DATA_SIZE:
0020

TYPE:
00010000

FLAGS: 0001

DATA:
21e3b30ff93bc6d3
5ac8c6e0e13afdff
794cb7b44bbbc748
d259d0a0284dbe84

)

RDATA:
0008c06fb9281580
0020000100010000
21e3b30ff93bc6d3
5ac8c6e0e13afdff
794cb7b44bbbc748
d259d0a0284dbe84

Encryption NONCE|EXPIRATION:
98132ea86859d35c
88bfd317fa991bcb
0008c06fb9281580

Encryption key (K):
85c429a9567aa633
411a9691e9094c45
281672be586034aa
e4a2a2cc716159e2

Storage key (q):
abaabac0e1249459
75988395aac0241e
5559c41c4074e255
7b9fe6d154b614fb
cdd47fc7f51d786d
c2e0b1ece76037c0
a1578c384ec61d44
5636a94e880329e9

BDATA:
9cc455a129331943
5993cb3d67179ec0
6ea8d8894e904a0c
35e91c5c2ff2ed93
9cc2f8301231f44e
592a4ac87e4998b9
4625c64af51686a2
b36a2b2892d44f2d

RRBLOCK:
000000b000010014

9bf233198c6d53bb
dbac495cabd91049
a684af3f4051baca
b0dcf21c8cf27a1a
44d240d07902f490
b7c43ef00758abce
8851c18c70ac6df9
7a88f79211cf875f
784885ca3e349ec4
ca892b9ff084c535
8965b8e74a231595
2d4c8c06521c2f0c
0008c06fb9281580
9cc455a129331943
5993cb3d67179ec0
6ea8d8894e904a0c
35e91c5c2ff2ed93
9cc2f8301231f44e
592a4ac87e4998b9
4625c64af51686a2
b36a2b2892d44f2d

Zone private key (d):
5af7020ee1916032
8832352bbc6a68a8
d71a7cbe1b929969
a7c66d415a0d8f65

Zone identifier (ztype|zkey):
000100143cf4b924
032022f0dc505814
53b85d93b047b63d
446c5845cb48445d
db96688f

zTLD:
000G051WYJWJ80S04BRDRM2R2H9VGQCKP13VCFA4DHC4BJT88HEXQ5K8HW

Label:
e5a4a9e4b88be784
a1e695b5

Number of records (integer): 3

Record #0 := (
EXPIRATION:
0008c06fb9281580

DATA_SIZE:

```
0010

TYPE:
0000001c

FLAGS: 0000

DATA:
0000000000000000
00000000deadbeef

)

Record #1 := (
EXPIRATION:
00b00f81b7449b40

DATA_SIZE:
0006

TYPE:
00010001

FLAGS: 8000

DATA:
e6849be7a7b0

)

Record #2 := (
EXPIRATION:
000000016b597108

DATA_SIZE:
000b

TYPE:
00000010

FLAGS: 4004

DATA:
48656c6c6f20576f
726c64

)

RDATA:
0008c06fb9281580
```

001000000000001c
0000000000000000
00000000deadbeef
00b00f81b7449b40
0006800000010001
e6849be7a7b00000
00016b597108000b
4004000000104865
6c6c6f20576f726c
6400000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000

Encryption NONCE|EXPIRATION:
bb0d3f0fbd224277
50da5d691216e6c9
0005db3bcdbd7769

Encryption key (K):
3df805bd6687aa14
209628c244b11191
88c3925637a41e5d
76496c2945dc377b

Storage key (q):
baf82177eec081e0
74a7da47ffc64877
58fb0df01a6c7fbb
52fc8a31bef029af
74aa0dc15ab8e2fa
7a54b4f5f637f615
8fa7f03c3fcebe78
d3f9d640aac0d1ed

BDATA:
6f79a9fd28bc5e38
2fc931ed22931797
326fdd698129fc47
8a639e902b411088
0a45037c667ff769
5f09c4a7f4f3471a
b2365bf3af79e953
697f1e35f93bd1ad
876971ce70527a3b
82c098d23fffd4a4
0057b694bec43416

4fb83c12b1f4570f
69a28f3bc3b7d838
b2619f6b8e1723ba
78c4b7ce19ef3f39
0405b63f7ce00216
1bdd7f5e9b3622bc
1af2d4ca84fd5fc5

RRBLOCK:
0000010000010014
74f90068f1676953
52a8a6c2eb984898
c53acca0980470c6
c81264cbdd78ad11
13b6b78358a88de7
3c5d22f73f1ad588
ee6f07d13410a2f5
15a074872608ec02
ef9020fdeb4266bf
1177c7e57e786059
97032a3f71f7216c
894e073ac77f2a0d
0005db3bcdbd7769
6f79a9fd28bc5e38
2fc931ed22931797
326fdd698129fc47
8a639e902b411088
0a45037c667ff769
5f09c4a7f4f3471a
b2365bf3af79e953
697f1e35f93bd1ad
876971ce70527a3b
82c098d23fffd4a4
0057b694bec43416
4fb83c12b1f4570f
69a28f3bc3b7d838
b2619f6b8e1723ba
78c4b7ce19ef3f39
0405b63f7ce00216
1bdd7f5e9b3622bc
1af2d4ca84fd5fc5

The following is an example revocation for a zone:

Zone private key (d, big-endian scalar):
6fea32c05af58bfa
979553d188605fd5
7d8bf9cc263b78d5
f7478c07b998ed70

Zone identifier (ztype|zkey):
000100002ca223e8
79ecc4bbdeb5da17
319281d63b2e3b69
55f1c3775c804a98
d5f8ddaa

Encoded zone identifier (zTLD):
000G001CM8HYGYFCRJXXXDET2WRS50EP7CQ3PTANY71QEQ409ACDBY6XN8

Difficulty (5 base difficulty + 2 epochs): 7

Signed message:
0000003400000003
0005d66da3598127
000100002ca223e8
79ecc4bbdeb5da17
319281d63b2e3b69
55f1c3775c804a98
d5f8ddaa

Proof:
0005d66da3598127
0000395d1827c000
3ab877d07570f2b8
3ab877d07570f332
3ab877d07570f4f5
3ab877d07570f50f
3ab877d07570f537
3ab877d07570f599
3ab877d07570f5cd
3ab877d07570f5d9
3ab877d07570f66a
3ab877d07570f69b
3ab877d07570f72f
3ab877d07570f7c3
3ab877d07570f843
3ab877d07570f8d8
3ab877d07570f91b
3ab877d07570f93a
3ab877d07570f944
3ab877d07570f98a

3ab877d07570f9a7
3ab877d07570f9b0
3ab877d07570f9df
3ab877d07570fa05
3ab877d07570fa3e
3ab877d07570fa63
3ab877d07570fa84
3ab877d07570fa8f
3ab877d07570fa91
3ab877d07570fad6
3ab877d07570fb0a
3ab877d07570fc0f
3ab877d07570fc43
3ab877d07570fca5
000100002ca223e8
79ecc4bbdeb5da17
319281d63b2e3b69
55f1c3775c804a98
d5f8ddaa053b0259
700039187d1da461
3531502bc4a4eecc
c69900d24f8aac54
30f28fc509270133
1f178e290fe06e82
ce2498ce7b23a340
58e3d6a2f247e92b
c9d7b9ab

Authors' Addresses

Martin Schanzenbach
Fraunhofer AISEC
Lichtenbergstrasse 11
85748 Garching
Germany

Email: martin.schanzenbach@aisec.fraunhofer.de

Christian Grothoff
Berner Fachhochschule
Hoeheweg 80
CH-2501 Biel/Bienne
Switzerland

Email: grothoff@gnunet.org

Bernd Fix
GNUnet e.V.
Boltzmannstrasse 3
85748 Garching
Germany

Email: fix@gnunet.org