Invisible Internet Common Data Structures

Revision 0.9, 28 August, 2003

http://www.InvisibleNet.net/ info@invisiblenet.net

jrandom@invisiblenet.net

Table of Contents

1.Document Overview

This document describes the common data structures used by the various Invisible Internet Project (I2P) protocols.

2.Data Types

Integer

Description:	Represents a nonnegative integer
Contents:	`1 or more bytes in network byte order representing an unsigned integer`
Notes:	

Date

Description:	The number of milliseconds since midnight on January 1, 1970 in the GMT timezone
Contents:	`8 byte Integer`
Notes:	If the number is 0, the date is undefined or null. (yes, this means you can't represent midnight on 1/1/1970)

String

Description:	Represents a UTF-8 encoded string
Contents:	`1 or more bytes where the first byte is the number of bytes (not characters!) in the string and the remaining 0-255 bytes are the non-null terminated UTF-8 encoded character array`
Notes:	

Boolean

Description:	A boolean value, supporting null/unknown representation
Contents:	`1 byte Integer`
Notes:	0 = false, 1 = true, 2 = unknown/null

Mapping

Description:	A mapping is a set of key / value pairs
Contents:	2 byte Integer defining the size of the mapping followed by that many bytes. In those bytes are UTF-8 encoded characters organized as *key=value;*. *Key* is a String unique within the mapping and cannot include the UTF-8 characters '=' or ';'. After that comes the literal UTF-8 character '=', followed by another String for a value. Finally comes the literal UTF-8 character ';'. This *key=value;* sequence is repeated until there are no more bytes (not characters!) left
Notes:	

PublicKey

Description:	This structure is used in ElGamal encryption, representing only the exponent, not the primes which are constant and defined in the appropriate spec.
Contents:	`256 byte Integer`
Notes:	

PrivateKey

Description:	This structure is used in ElGamal decryption, representing only the exponent, not the primes which are constant and defined in the appropriate spec.
Contents:	`256 byte Integer`
Notes:	

SessionKey

Description:	This structure is used for AES256 encryption and decryption
Contents:	`32 byte Integer`
Notes:	

SigningPublicKey

Description:	This structure is used for verifying DSA signatures
Contents:	`256 byte Integer`
Notes:	

SigningPrivateKey

Description:	This structure is used for creating DSA signatures
Contents:	`20 byte Integer`
Notes:	

Signature

Description:	This structure represents the DSA signature of some data
Contents:	`40 byte Integer`
Notes:	

Hash

Description:	Represents the SHA256 of some data
Contents:	`32 bytes`

Notes:	

Certificate

Description:	A certificate is a container for various receipts or proof of works used throughout the I2P network.
Contents:	`1 byte Integer specifying certificate type, followed by a 2 byte Integer specifying the size of the certificate payload, then that many bytes`
Notes:	Certificates of type 0 (null certificates) ignore the contents of the payload. Other certificates, such as type 1 (hashcash certificates) , the payload is specific to the algorithm in use. Hashcash certificates contain a 1 byte Integer specifying the number of bits (K) for the hash collision, followed by a non-trivial (non-identity) hash collision against the first K bits of the certified data, using the hashcash function defined at http://www.cypherspace.org/hashcash/hashcash.pdf. Other certificate types, such as real cash payment certificates or CA signed keys can be added later.

3.Common Structures

Destination

Description:	A Destination defines a particular end point to which messages can be directed for secure delivery.
Contents:	`PublicKey followed by a SigningPublicKey and then a Certificate entangled with the PublicKey`
Notes:	

Lease

Description:	Defines the authorization for a particular tunnel to receive messages targeting a `Destination`
Contents:	`RouterIdentity of the gateway router, then the TunnelId, and then a start Date and finally an end Date`
Notes:	

LeaseSet

Description:	Contains all of the currently authorized `Leases` for a particular `Destination`, the public key to which garlic routed messages can be encrypted, and then the public key which can be used to revoke this particular version of the structure. The `LeaseSet` is one of two structures stored in the network database (the other being `RouterInfo`), and is keyed under the SHA256 of the contained `Destination`.
Contents:	`Destination, followed by a PublicKey for encryption, then a SigningPublicKey which can be used to revoke this version of the LeaseSet, then a 1 byte Integer specifying how many Lease structures are in the set, then a 4 byte Integer defining the version of this structure, followed by the actual Lease structures and finally a Signature of the previous bytes signed by the Destination's SigningPrivateKey`
Notes:	

Payload

Description:	This structure is the content of a message being delivered from one `Destination` to another.
Contents:	`4 byte Integer specifying the size of the structure, followed by that many bytes`
Notes:	The contents of the Payload is encrypted to the `Destination`'s `PublicKey`, using ElGamal+AES256, as described below.

RouterIdentity

Description:	Defines the way to uniquely identify a particular router
Contents:	`PublicKey followed by SigningPublicKey and then a Certificate entangled with the PublicKey`
Notes:	

RouterInfo

Description:	Defines all of the data that a router wants to publish for the network to see. The `RouterInfo` is one of two structures stored in the network database (the other being `LeaseSet`), and is keyed under the SHA256 of the contained `RouterIdentity`.
Contents:	`RouterIdentity followed by a 4 byte Integer determining the version of the structure, then a 1 byte Integer specifying how many RouterAddress structures follow, then those actual structures. After that comes a 1 byte Integer specifying how many RouterSightings are included, followed by those sightings. Following that is a Mapping structure allowing the router to publish some metadata about itself, such as statistics, capabilities, and configuration options. Finally there is a Signature of the entire structure as created by the RouterIdentity's SigningPrivateKey.`
Notes:	

RouterAddress

Description:	This structure defines the means to contact a router through a transport protocol.
Contents:	`String defining the transport protocol this router address uses, followed by a 1 byte Integer defining the relative cost of using the address, where 0 is free and 255 is expensive. After that comes a Date after which the address should not be used, or if null, the address never expires. Finally there is a Mapping containing all of the protocol specific options necessary to establish the connection, such as IP address, port number, email address, URL, etc.`
Notes:	

RouterSighting

Description:	Proves that a particular router has agreed to source route messages destined for another router over a trusted connection.
Contents:	`Hash of the sighted router, followed by the Date of the sighting and the Date at which the sighting expires, and finally a Signature of the structure by the sighted router.`
Notes:	

TunnelId

Description:	Defines an identifier that is unique within a particular set of routers for a tunnel
Contents:	`4 byte Integer`
Notes:	

SessionTag

Description:	Used with various encryption techniques to identify what particular session key should be used to decrypt the data
Contents:	`32 bytes`

Notes:	Tags should be random and can only be used once in a time period.

4.I2CP Structures

SessionConfig

Description:	Defines the configuration options for a particular client session
Contents:	`Destination followed by a Mapping and finally a Signature by the Destination's SigningPrivateKey`
Notes:	

SessionId

Description:	Uniquely identifies a session on a particular router at a point in time
Contents:	`2 byte Integer`
Notes:	

MessageId

Description:	Uniquely identifies a message waiting on a particular router at a point in time.
Contents:	`4 byte Integer`
Notes:	

AuthenticationKey

Description:	This is calculated based on the passphrase necessary to access the administrative features of the router
Contents:	`Hash`
Notes:	The Hash is the SHA256 of the passphrase

BandwidthLimits

Description:	Contains the limits defined on how much bandwidth a router will use, with different limits associated with different classes of routers.
Contents:	`1 byte Integer specifying the number of classes defined, followed by that many bandwidth limit structures. Each of those structures starts with a String defining the user specified name for the class, followed by a 1 byte Integer specifying the number of routers in that class, then the Hash of each of their RouterIdentity structures, and finally a set of four 4 byte Integers defining the maximum average bytes per second downloaded, uploaded, and the peak bytes per second downloaded and uploaded, respectively. There must be exactly one limit structure that contains no routers which serves as the default class.`
Notes:	

AbuseReason

Description:	Contains a description of why the abuse is being reported

Contents:	`String`
Notes:	

AbuseSeverity

Description:	Specifies how severe the abuse was
Contents:	`1 byte Integer`
Notes:	0 is minimally abusive, 255 being extremely abusive

5.I2NP Structures

EndPointPublicKey

Description:	Data targeting the router at which a particular `Destination` is connected can be encrypted to this key
Contents:	`PublicKey`
Notes:	

EndPointPrivateKey

Description:	The private key associated with the `EndPointPublicKey`
Contents:	`PrivateKey`
Notes:	

TunnelVerificationsStructure

Description:	This structure is passed down a tunnel to make sure messages are not modified
Contents:	`SHA256 Hash of the data followed by the Signature from the TunnelSigningPrivateKey`
Notes:	

TunnelSigningPublicKey

Description:	The public key which `TunnelVerificationStructure` signatures are verified against
Contents:	`PublicKey`
Notes:	

TunnelSigningPrivateKey

Description:	The private key associated with the `TunnelSigningPublicKey`
Contents:	`PrivateKey`
Notes:	

TunnelConfigurationSessionKey

Description:	The session key used to issue update and delete instructions to routers already participating in a tunnel.
Contents:	`SessionKey`
Notes:	

TunnelSessionKey

Description:	The session key used to encrypt and decrypt `DeliveryInstructions` between the tunnel's gateway and it's end point
Contents:	`SessionKey`
Notes:	

DeliveryInstructions

Description:	Specifies how a message should be handled
Contents:	`1 byte set of flags describing the method of delivery, followed by the SHA256 Hash of the message, and finally the associated instructions`
Notes:	Each bit may add data to the instructions. The instructions are by default null, but the following bit flags can add data to that, following the order of anything added by the most significant bit to the least: The most significant bit (bit 0) in the flag represents whether the message is encrypted. If it is 0, no additional instructions are added. If it is 1, a `SessionKey` follows, with which the message is decrypted. The `Hash` references the decrypted message. The next two most significant bits (bits 1 and 2) in the flag represents how the message is delivered. If they are 00, the message is delivered locally and no additional instructions are added. If they are 01, the message is delivered to a destination and the `Hash` of the `Destination` is added. If they are 10, the message is delivered to a router and `Hash` the of the router's `RouterIdentity` is added. If they are 11, the message is delivered to a tunnel and the extra info contains the `Hash` of the router's `RouterIdentity` and then the `TunnelId`. The third most significant bit (bit 3) in the flag represents whether a delay is requested. If that flag is not set, no additional instructions are added. If it is set, then a 4 byte `Integer` is added, specifying the minimum number of seconds the sender is requesting the router delay sending the message for.

GarlicClove

Description:	A clove is a single piece of an unwrapped `GarlicMessage` requesting handling.
Contents:	`DeliveryInstructions then the 4 byte Integer specifying the size of the clove's payload, then that many bytes of data, then a 4 byte Integer specifying this clove's unique identifier, then a Date at which the clove should be dropped, a Certificate, and finally an optional SourceRouteBlock.`
Notes:	

SourceRouteBlock

Description:	This structure allows a message to be passed one hop in the network without exposing its real target.
Contents:	`SHA256 Hash of the RouterIdentity to which the data should be directed, followed by the a set of encrypted data.  This encrypted data contains the DeliveryInstructions, then a 4 byte message identifier, then a Certificate`

Notes:	The data is encrypted to the first step router's `RouterIdentity` public key using ElGamal+AES56 as described below.

6. Encryption

ElGamal + AES256

The data encrypted can take one of two forms. If the first 32 bytes are not equal to a known and unused `SessionTag`, as defined by a previous message, the first scenario takes place. Otherwise, the second takes place.

1. **Scenario 1: unknown `SessionTag`, or incorrect hash of session key**
   The first 514 bytes is decrypted with ElGamal against the appropriate `PublicKey` using the algorithm below. The first 32 bytes unencrypted make up the `SessionKey`. After that comes 32 bytes which, when hashed with SHA256 and the first 16 bytes of that `Hash` is retrieved, turns into the AES256 initialization vector (IV). The remaining bytes are random padding. All of the data following the initial 514 bytes is then decrypted with the AES `SessionKey` and the IV:
   - The content starts with a 2 byte `Integer` specifying the number of session tags that follow. After that comes that many 32 byte random `Integers` that act as session tags. After that comes an 4 byte `Integer` specifying the real size of the body of the payload to follow. Then comes the `Hash` of the the unencrypted body, for verification. Then there is a 1 byte `Integer`, which if it is set to 1, is followed by a new `SessionKey` that should be used on subsequent messages in this session (but not for this message). The remainder is the actual body of the message, padded with random bytes to match the size specified earlier in the decrypted data. The overall size of the data to be AES encrypted must be a multiple of 16 bytes, so padding should be placed accordingly.
2. **Scenario 2: known session key, known unused session tag**
   The 32 bytes acting as the session tag are hashed with SHA256 and the first 16 bytes of that `Hash` is retrieved and turned into the AES256 initialization vector (IV). The remainder of the data is considered to be encrypted with the `SessionKey` currently associated with the session (either as defined in the original scenario 1 message or rekeyed in a later message). The decrypted data is as follows:
   - The decrypted body starts with a `Hash` of the `SessionKey` in use. If this doesn't match the one in use, then scenario 2 is aborted and the entire structure is treated according to scenario 1. If it does match, then the processing continues, where the rest of the decrypted body is the same as defined in scenario 1 (e.g. After decryption it starts with a 2 byte `Integer`).

Session tags received should be kept for **1 hour** after being received, or until they are used (whichever comes first). Once a session tag is used or the time has passed, it should be discarded and further messages should not be checked for that tag. The code implementing the session tagging and verification may keep a list of tags that have been used with a particular session ID and make sure the other party doesn't try to reuse a session tag. Note: for extremely slow transports, extremely paranoid people (using many tunnel hops), or any other scenario where a message may take more than an hour to be received, scenario 1 should be used exclusively.

To the observer, there should be no way to tell whether the data begins with a `SessionTag` or is ElGamal encrypted. If this isn't the case, an observer can tell that some destination somewhere is receiving follow-on traffic. Since `SessionTag` change constantly, there is no way for an observer to know what session a tagged message belongs so (even if they were able to detect a scenario 1 vs scenario 2 message)

The private `SessionKeys` will be changed frequently, providing perfect forward secrecy with regard to the messages passed over the network (though if the `PrivateKey` of the `Destination` is compromised, all messages recorded will be broken, so endpoints should periodically change their `Destination`).

The algorithm used for AES encryption and decryption is AES256 with 16 byte blocks in CBC mode.

AES256

Straight AES256 encryption done without ElGamal begins with the SHA256 of the decrypted data, followed by a 4 byte `Integer` specifying the size of the decrypted data, and then the actual data and finally a set of random padding bytes making the total size a multiple of 16 bytes. AES256 operates with 16 byte blocks in CBC mode with a 16 byte IV.

ElGamal

ElGamal encryption is calculated with 2048bit `PublicKeys`, resulting in a 514 byte block using the ElGamal algorithm specified in the Handbook of Applied Cryptography[1]. The block starts with a 257 byte `Integer` representing γ, with the second 257 byte `Integer` representing δ. After calculating the decrypted value, there are zero or more leading bytes with the value of 0x00, followed by a single byte with any non-0x00 value, then a `Hash` of the payload, and then finally the payload itself. This leaves the largest number of bytes that can be placed in each ElGamal block by this engine at 223 bytes, though to assure the data, when treated as an `Integer`, is less than the ElGamal prime, no more than 222 bytes should be used.

Encryption Constants

These values are hex encoded in network byte order.

ElGamal prime:

This is the Oakley prime for 2048bit keys[2].

The value is: 2^2048 - 2^1984 - 1 + 2^64 * { [2^1918 pi] + 124476 }

or:

```
FFFFFFFF FFFFFFFF C90FDAA2 2168C234 C4C6628B 80DC1CD1
29024E08 8A67CC74 020BBEA6 3B139B22 514A0879 8E3404DD
EF9519B3 CD3A431B 302B0A6D F25F1437 4FE1356D 6D51C245
E485B576 625E7EC6 F44C42E9 A637ED6B 0BFF5CB6 F406B7ED
EE386BFB 5A899FA5 AE9F2411 7C4B1FE6 49286651 ECE45B3D
C2007CB8 A163BF05 98DA4836 1C55D39A 69163FA8 FD24CF5F
83655D23 DCA3AD96 1C62F356 208552BB 9ED52907 7096966D
670C354E 4ABC9804 F1746C08 CA18217C 32905E46 2E36CE3B
E39E772C 180E8603 9B2783A2 EC07A28F B5C55DF0 6F4C52C9
DE2BCBF6 95581718 3995497C EA956AE5 15D22618 98FA0510
```

[1] http://www.cacr.math.uwaterloo.ca/hac/about/chap8.pdf page 13, or page 294 in the 2nd edition

[2] http://www.ietf.org/proceedings/03mar/I-D/draft-ietf-ipsec-ike-modp-groups-05.txt

```
15728E5A 8AACAA68 FFFFFFFF FFFFFFFF
```

ElGamal generator:

```
2
```

DSA constants:

```
SEED = 86108236b8526e296e923a4015b4282845b572cc
Counter = 33
```

DSA prime:

```
9C05B2AA 960D9B97 B8931963 C9CC9E8C 3026E9B8 ED92FAD0
A69CC886 D5BF8015 FCADAE31 A0AD18FA B3F01B00 A358DE23
7655C496 4AFAA2B3 37E96AD3 16B9FB1C C564B5AE C5B69A9F
F6C3E454 8707FEF8 503D91DD 8602E867 E6D35D22 35C1869C
E2479C3B 9D5401DE 04E0727F B33D6511 285D4CF2 9538D9E3
B6051F5B 22CC1C93
```

DSA quotient:

```
A5DFC28F EF4CA1E2 86744CD8 EED9D29D 684046B7
```

DSA generator:

```
C1F4D27D 40093B42 9E962D72 23824E0B BC47E7C8 32A39236
FC683AF8 48895810 75FF9082 ED32353D 4374D730 1CDA1D23
C431F469 8599DDA0 2451824F F3697525 93647CC3 DDC197DE
985E43D1 36CDCFC6 BD5409CD 2F450821 142A5E6F 8EB1C3AB
5D0484B8 129FCF17 BCE4F7F3 3321C3CB 3DBB14A9 05E7B2B3
E93BE470 8CBCC82
```

7. References

- Invisible Internet Network Protocol
- Invisible Internet Client Protocol