Crypto module

> This document is a part of **MFlod** messenger documentation which an implementation of **Flod** overlay protocol. For a complete version of project documentation see [no link yet]. [TODO: add link]

This is a documentation for `Crypto` module of **MFlod** messenger. It describes the process of creation of **Flod** protocol message packet, security implications of a cryptographic design behind it and how all of this is implemented in **MFlod**.

Table of Contents

Message Packet Structure

The message packet is a structure that represents an encrypted message that is sent from a one user of **Flod** protocol overlay to an other. The size of the packet can be of variable length though some of its parts are always remain constant.

The general structure of the message packet is described below together with some security implications while the further sections describe how a general packet structure is expressed in terms of ASN.1 which is what actually used in a reference implementation.

General Structure

> The general structure documentation block is intended as a description of the message packet creation logic. It should not be used on its own to implement the message packet assembler. The block on [TODO: add link] ASN.1 structure is what to be used as an actual implementation reference.

In essence the message packet structure is very similar to a standard hybrid cryptosystem scheme. The illustration below is a high-level overview of **Flod** message packet:

```
--------------------------------------
|            |          |             |
| (2) HEADER | (1) HMAC | (0) CONTENT |
```

The indices from right to left are motivated by the order in which message packet is constructed. The basic description of the each block is as follows:

- `(0) CONTENT` - encrypted with AES-128-CBC timestamped message with prepended plaintext IV (initialization vector).
- `(1) HMAC` - hash-based message authentication code of a `(0)` block. The underlying hash function is SHA-1.
- `(2) HEADER` - encrypted with public key of a recipient with RSA-OAEP meta-information about a message along with keys generated for blocks `(0)` and `(1)`.

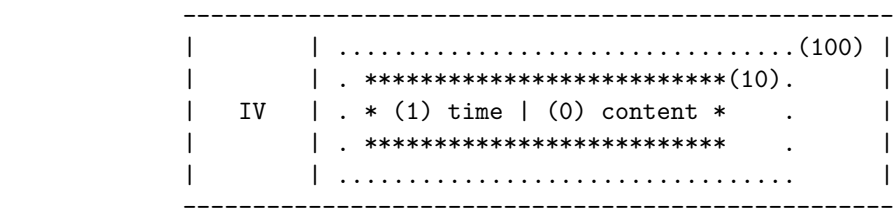
Below is a more detailed description of content of each block of the message packet.

(0) Content Block

The content block encapsulates the message being sent and time stamp of when it was composed. Concatenated together they are encrypted with a uniformly at random chosen encryption key (128 bit) of AES-128-CBC cipher.

The IV (128 bit) for AES-CBC encryption is also generated uniformly at random. The concatenation of the time stamp and the message is padded with PKCS#7 padding scheme.

The resulting `(0) CONTENT` block looks the following:



To create a valid content block the following steps to be performed (omitting ASN.1 aspects to simplify the initial description):

1. Get message to send from a user `(0)`
2. Get current UTC time in a format `YYMMDDhhmmssZ` (complies with ASN.1 UTCTime type) `(1)`
3. Concatenate `(1)` and `(0)` yielding `(1)|(0)`
4. Pad `(1)|(0)` according to PKCS#7 standard to get block `(10)`. At this stage block `(10)` should be strictly a multiple of AES block size (which is 128 bit).
5. Generate a random 128 bit bytestring which is an AES-128-CBC key for this message packet. The key generated is used **only 1 time** to encrypt

the current message. Each new message must be encrypted with randomly generated fresh AES key.
6. Generate a random 128 bit bytestring which is an initialization vector for this encryption procedure only. Each new message must be encrypted with randomly generated fresh IV.
7. Encrypt block `(10)` with AES-128 in CBC mode using the key and the IV generated during steps 5-6. The result is a block `(100)`.
8. Prepend block `(100)` with IV used for encryption (generated on step 6). After that the full `(0) CONTENT` block was constructed.

The reason why AES-128 was chosen over AES-256 is due to a poor design of a key schedule for a later flavor of AES (see this article).

(1) HMAC Block

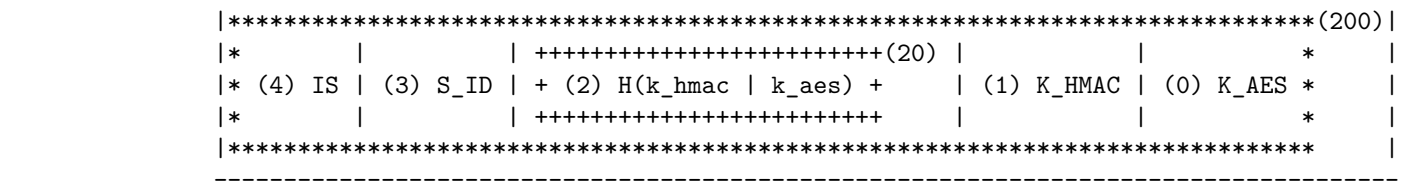
This block encapsulates a hash-based message authentication code for the block `(0) CONTENT`. The hash function of choice is SHA-1 and despite recent news of Google breaking it HMAC bases on SHA-1 is still secure (see [1], [2], [RFC2104, page 5]).

To produce a correct HMAC for a block `(0) CONTENT` the following steps are necessary:

1. Generate a random 160 bit bytestring which is a key to use for this HMAC calculation. Every new message must utilize a random and fresh HMAC key.
2. Calculate an SHA1-HMAC of `(0) CONTENT` block with a key from step 1. The result is the `(1) HMAC` block.

(2) Header Block

The header block is an encrypted storage for keys used in previous blocks as well as some extra meta-information that facilitates protocol operation. The structure is as follows:



These steps are necessary to produce a valid header block:

1. Concatenate keys that were used in `(1) HMAC` block and `(0) CONTENT` blocks. The first one is the key used to calculate an HMAC. The second

one is a key used in AES encryption. The local block `(1)` and `(0)` are ready.

2. Calculate a SHA-1 hash from the result of step 1. This yields a local block `(2)`.
3. Sign local block `(2)` produced in the previous step with RSA private key of a sender. The signature is a local block `(20)`. Prepend result of the step 1 with the signature.
4. Prepend the result of the previous step with a PGP keypair ID of a sender which was used to produce a signature in the previous step. If the sender's keypair is not a PGP keypar append all-zero dummy ID.
5. Prepend the result of the previous step with a 4-byte indentification string: *FLOD*. This values is used to determine a successful decription of a header block on a side of recipient.
6. Pad result of the previous step according to a OAEP padding standard
7. Encrypt the result of the previous step with RSA public key of the recipient. The result is a local block `(200)` and itself is a full `(2) HEADER` block.

Note that *steps 2-4 are optional* and can be omitted depending on a sender's decision. Though it would not allow the recipient to verify the identity of the sender in any way. But it still allows the recipient to decrypt a message.

The minimal required size of RSA keypair for both sender and recipient is 1024 bits. The recommended size of RSA keypair is at least 2048 bits.

In case of a usage of PGP RSA keypairs the actual public and secret key information must be extracted from PGP containers. The signature creation and encryption routines **cannot be preformed** by PGP software suit. The reason for it is a leakage of meta-information about a keypair owner in PGP software (ID, email etc).