


M3DA Security Extension

 This is a draft document

M3DA Security Extension

Document history

Date	Version	Author	Comments
Aug 11th 2009	01/A	Cuero Bugot	Document Creation
Jul 4th 2011	01/B	Gilles Cannenterre	Fix issues on public/protected headers fields Restrict hash function Define padding for CDC mode
Aug 26th 2011	1.0	Cuero Bugot	Refactor the document into confluence Add Nested envelope for protected headers

Reference documents

M3DA	M3DA Protocol	M3DA Protocol Specification
PAD	PKCS5 padding	RFC2898, RFC1423
HMAC	keyed-Hash Message Authentication Code	RFC2104
MD5	Message-Digest algorithm 5	RFC1321
SHA1	US Secure Hash Algorithm 1	RFC3174
HTTP	Hypertext Transfer Protocol - HTTP/1.1	RFC2616

Table of Content

- [M3DA Security Extension](#)
 - [Document history](#)
 - [Reference documents](#)
 - [Table of Content](#)

- [1. Introduction](#)
 - [1.1. Definitions](#)
 - [1.2. Notations](#)
- [2. Packages](#)
 - [2.1. Authentication](#)
 - [2.2. Challenge](#)
 - [2.3. One way package](#)
- [3. Envelope header and footer fields](#)
 - [3.1. Nonce](#)
 - [3.2. Status](#)
 - [3.3. Challenge](#)
 - [3.4. Authentication](#)
 - [3.5. Confidentiality](#)
- [4. Standard processes](#)
 - [4.1. Credentials](#)
 - [4.2. Nonce management](#)
 - [4.3. Authentication](#)
 - [4.4. Encryption](#)
 - [4.4.1. Cipher key computation](#)
 - [4.4.2. Initial vector](#)
 - [4.4.2.1. Padding](#)
- [5. Sequence diagram](#)
 - [5.1. Simple authenticated session](#)
 - [5.2. Authenticated session with server sending some data](#)
 - [5.3. Server initiated challenge session](#)
 - [5.4. Client initiated challenge session](#)
 - [5.5. Failed challenge session](#)
- [6. Denial of service protection](#)
 - [6.1. Authentication before accepting a lot of data](#)
 - [6.2. Ignore successive challenge messages](#)

1. Introduction

This document details the security mechanism that enable authentication, message integrity and ciphering of exchanges between server and devices.

This is an extension of the M3DA protocol specification document [#M3DA](#). The M3DA protocol has two distinct communication layers. One is the Envelope which can be assimilated to the transport layer, the other one is the M3DA::Message that is actually the application layer.

All the security-related add-ons of this specification are located into the transport layer i.e. the Envelope. More precisely this specification adds some specific fields in the Envelope headers and footers and some specific work-flows for setting up secured connections.

The authentication scheme must be symmetric. This means that if one peer is requesting authentication, it will also be authenticated by the other peer. This is mandatory so that there will not be nonce de-synchronization by a third party.

1.1. Definitions

- **Authentication:** Authentication is the process of ascertaining the validity of the remote peer identity (either

the Device or the Server).

- **Confidentiality:** Confidentiality is the ability to keep contents secret from all but the two peers exchanging a message. It does not limit the visibility of the message (being able to eavesdrop), but it does prevent the interpretation of the data being transmitted. Effectively confidentiality prevents the contents of a message from being understood by anybody but the intended sender and intended recipient.
- **Credentials:** Credentials are elements that are required to prove authenticity. Typically a username and a password.
- **Device:** See [#M3DA](#).
- **Integrity** Integrity is the ability for a message to maintain its content or at a minimum, have the ability to detect modification or corruption of its content.
- **Envelope:** See [#M3DA](#).
- **Nonce:** A Nonce is a public information (basically a string) that is shared between the two peers, randomly updated at each message and that is used to generate the HMAC key.
- **One way package:** See [#M3DA](#).
- **Package:** See [#M3DA](#).
- **Session:** See [#M3DA](#).

1.2. Notations

Mathematical formula operators:

- \circ is the concatenation operator
- \oplus is the bitwise exclusive OR operator

2. Packages

2.1. Authentication

During normal sessions, both request and reply are authenticated. The authentication works both ways: the server authenticates the device, and the device authenticates the server. When there is an authentication problem, a challenge envelope is issued.

In order to authenticate an envelope the receiver must compute the *mac* key of the received envelope and verify that it matches the *mac* key provided in the envelope footers.

2.2. Challenge


When one peer receives an envelope that is not authenticated, it must reply with status code 407 (authentication required) if there is no authentication field in the envelope header, or with status code 401 (Unauthorized) if the authentication fields are not correct. In addition the peer must add an envelope header field challenge that specify which authentication type it requires.

A challenge envelope does not contain authentication credentials, so to prevent the other peer to ask for an authentication of the challenge. A challenge is issued only one time. If the authentication fails after a challenge issue, the peer should close the session instead of sending a new challenge.

A challenge envelope is actually a reply: it contains the *status* field. This means that a session cannot be started with a challenge envelope.

2.3. One way package

One way packages are defined in the M3DA Standard specification [M3DA](#). It is important to point out the fact that in one way communication a peer cannot know if the message was authenticated successfully or even if the message was actually received. Because of this limitation a nonce de-synchronization is highly probable in that scheme.

 when a one way package is received, the nonce is only updated if the package is authenticated.

3. Envelope header and footer fields

In addition of the standard fields defined in the [M3DA](#) document, new fields are added into the envelope headers and footers to enable the security layer. Those fields are used at different stage of the connection process.

The Security layer requires that some fields are protected. Protected fields are part of the authenticated envelope, and when encryption is enabled, the protected fields are encrypted as well.

In order to provide this protection mechanism, M3DA authorize to have nested envelopes, up to two layers. The second envelope (the one that is encapsulated) is then protected, and thus its headers and footers are protected fields, as much as its payload.

Public header field	Description
<i>id</i> ★	Sender ID. See M3DA . When security layer is enabled this field is mandatory for both ways of communication: device to server, and server to device. When the device sends data, this field is set to the DeviceID, when the server sends data, this field is set to the ServerID.
<i>challenge</i>	Indicate that the peer challenges the other peer in order to accomplish a correct authentication. The challenge must comply with the given challenge method.
Public footer field	Description
<i>mac</i>	This is the message authentication code. This is used by the authenticator for ascertaining the identity of the remote peer with the given mac value.
Protected header field	Description
<i>nonce</i>	This specify the nonce that must be used for the next authenticated envelope.
<i>status</i> ★	Informs of the status of the package. See M3DA . The status value maps on standard HTTP status values. This field must be present when sending a response envelope.

★ this fields are defined in [M3DA](#) document. This specification may add more constraints on those fields but still complying with M3DA specification.

3.1. Nonce

The nonce mechanism is added in this security enhancement in order to prevent “replay” attacks. The nonce is used in the HMAC process so that the produced hash always depends on the nonce. The value of the nonce is changed for each envelope. The value of the next nonce to be used is given during the current envelope header field *nonce*.

When initiating a new package, the next nonce should be derived as a random value that should be hardly predictable.

There are two strategies for the initial nonce synchronization. Either the nonce value is provisioned for each server-device pair, or the nonce is initialized to a random value, and the first connection will trigger a challenge mechanism. If the latter solution is used, the first connection cannot be a one way package (as SMS) because nonce will be out of sync.

When the mac computation does not match (happens when the nonce is not in sync between the two peers due to initialization or loss of packets) a challenge is sent to the other peer. It allows to re-synchronize the nonce.

The length (in bytes) of the nonce should be at least equal to the length of the output of the hash function, i.e. 16 bytes for MD5 and 20 bytes for SHA1.

The nonce is actually a shared variable between the two peers. The nonce need to be stored into a persistent memory storage so it will survive to reboot sequences.

The nonce is always updated and stored when a peer receives an authenticated envelope. When the peer receives a challenge envelope (which is not authenticated by definition), the nonce is updated temporarily (not persistently stored) until the other peer is authenticated.

A peer updates and stores the received nonce only when the envelope is successfully authenticated. This prevents third parties to intentionally desynchronize the nonce variables. There is one exception when receiving a challenge envelope where the peer must use the nonce provided in the envelope. However the generated nonce in the reply is not stored until the authentication of the other peer is successful.

Nonce management is specified in the section [#Nonce management](#).

3.2. Status

The *status* field in the envelope header holds the status code of the previous request. The values of the status are derived from the one specified into the HTTP specification [HTTP](#).

The following status are added by this specification:

Status Code	Description
401	Unauthorized (usually happens when the credentials are not correct)
407	Authentication required (usually happens when no credentials were provided)
450	The payload data need to be encrypted (usually happens when a peer try to send data without encryption and the other require the data to be encrypted)

3.3. Challenge

The *challenge* field is used to challenge the other peer to prove its credentials. The *challenge* fields must be one of the defined authentication scheme. See [#Authentication](#).

The *challenge* field can only be present into a challenge envelope: an envelope that has a *status* field and no payload.

In a challenge envelope the *status* and *nonce* fields are in the public headers/footers.

3.4. Authentication

The authentication method is configured out of band. Both peers must have the same authentication method so that a successful handshake can happen.

The way to synchronize the authentication method is not part of this specification.

The available authentication methods are defined in the following table.

Authentication method	Description
<i>hmac-md5</i>	keyed-Hash Message Authentication Code using MD5 hash function
<i>hmac-sha1</i>	keyed-Hash Message Authentication Code using SHA1 hash function

The MAC value is computed on the the binary string composed of the serialized protected envelope (the encapsulated envelope that contains its headers, payload and footers). This means that the protected header/footer fields are protected against any alteration (integrity).

On the contrary the public header/footer fields are not used for the MAC computation and should not be used to transport sensitive (non authenticable) information, apart from the device/server Id and the MAC value that are integral part of the authentication mechanism.

3.5. Confidentiality

The authentication mechanism does not ensure data confidentiality. An encryption layer is added when confidentiality is needed. Confidentiality layer requires authentication layer to work, in particular it uses the same nonce value to compute the cipher key.

The symmetric ciphering algorithm is configured out of band. Both peers must have the same ciphering algorithm so that a successful communication can happen.

The available ciphering algorithms are defined in the following table.

Cipher algorithm	Description
<i>aes-cbc-128</i>	AES Cipher-block chaining with a 128-bit key and PKCS5 padding

<i>aes-cbc-256</i>	AES Cipher-block chaining with a 256-bit key and PKCS5 padding
<i>aes-ctr-128</i>	AES Counter mode with a 128-bit key (no padding required), big endian counter
<i>aes-ctr-256</i>	AES Counter mode with a 256-bit key (no padding required), big endian counter

The encryption is applied on the the binary string composed of the serialized protected envelope (the encapsulated envelope that contains its headers, payload and footers). This means that the protected header/footer fields are encrypted and not transmitted in the clear, so they can convey sensitive information.

On the contrary the public header/footer fields are transmitted without any encryption and must not convey any secret information.

4. Standard processes

4.1. Credentials

For each device-server couple a single password is defined. The password is defined within ASCII 7bits character set, excluding ',' (coma) and ';' (semi colon) characters.

This password is derived into a key, K, which does not add security but only provide a convenient way to exchange this credential information. Later on, only the key K is used, so the password need not to be stored.

K is computed in the following way:

$$K = H_{MD5}(\text{password})$$

Both the device and the server are uniquely identified with a device id and a server id.

The unique key is then derived into device and server specific keys in the following way:

$$K_D = H_{MD5}(\text{deviceid} \circ K)$$

$$K_S = H_{MD5}(\text{serverid} \circ K)$$

where:

- H_{MD5} is the MD5 hash function

The secret hash keys K , K_S and K_D are as sensitive as the original password, and must be handled as such.

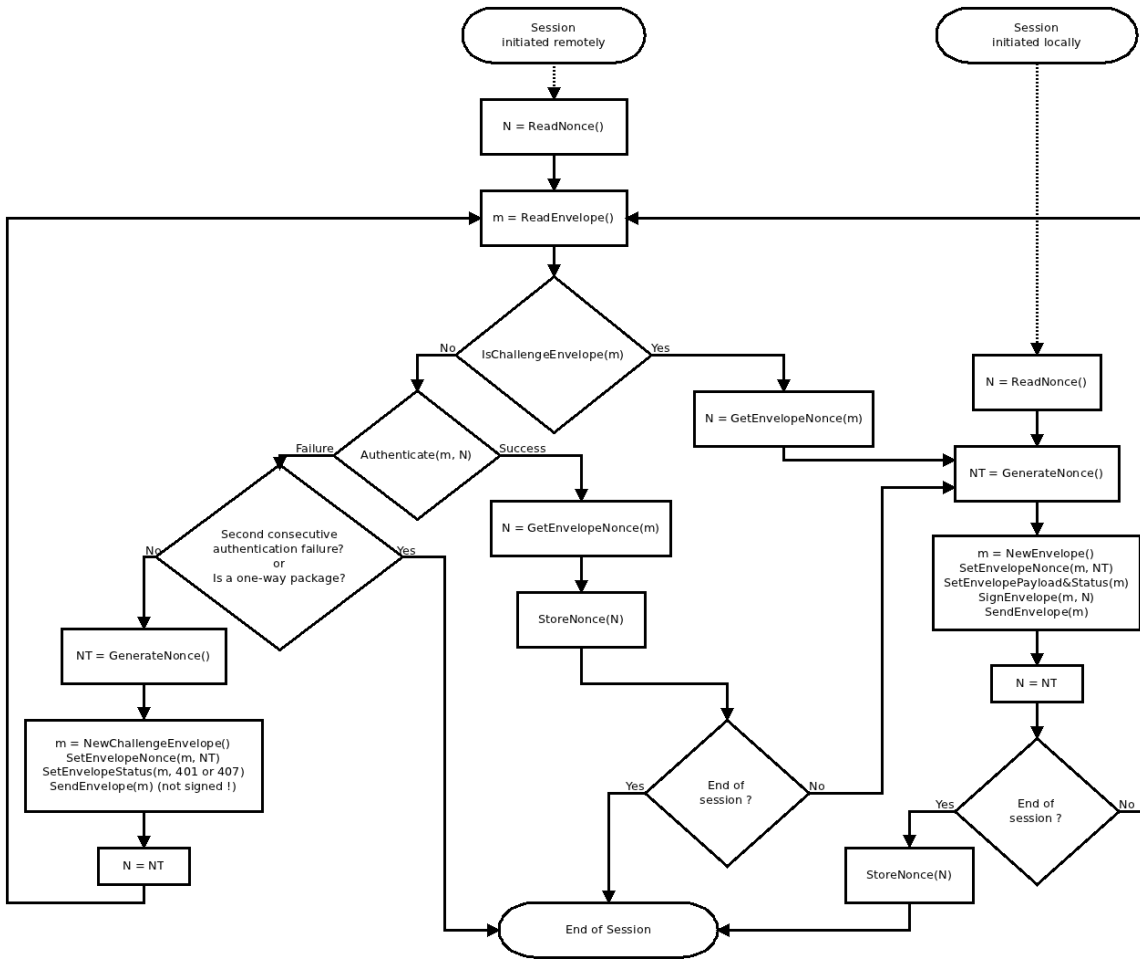
4.2. Nonce management

The nonce management flowchart specify how the nonce should be handled during a session.

The following functions are used:

- **ReadNonce()** reads and returns the nonce that is currently stored into the persistent storage
- **StoreNonce(N)** stores the nonce N into persistent storage
- **NewEnvelope()** creates and returns a new envelope
- **NewChallengeEnvelope()** creates and returns a new challenge envelope (an envelope that holds the challenge header field)
- **ReadEnvelope()** reads and returns an envelope that is incoming from the other peer
- **SendEnvelope()** sends an envelope to the other peer
- **IsChallengeEnvelope(m)** returns Yes if m is a challenge envelope, No otherwise
- **SignEnvelope(m, N)** signs the envelope m using the peer credentials and the given nonce
- **Authenticate(m, N)** authenticates the envelope m with the peer credentials and the given nonce
- **GetEnvelopeNonce(m)** return the nonce that is written in the nonce field of the envelope header
- **SetEnvelopeNonce(m, N)** sets the envelope m nonce header field to the given value N
- **SetEnvelopeStatus(m, s)** sets the envelope m header field status to the given value s
- **SetMessagePayload&Status(m)** sets the payload and/or status of an envelope m with the correct values depending on the current package
- **GenerateNonce()** returns a generated a nonce as recommended into this document
- **Second consecutive authentication failure?** The answer is Yes if this envelope was not authenticated and the previous envelope was not authenticated as well.
- **Is a one-way package?** The answer is Yes if the received envelope is a one-way package.
- **End of session?** The answer is Yes if the previous sent/received envelope was not a request (has a status field and no payload) and not a challenge, No otherwise. In particular: when sending a challenge this cannot be an end of session envelope (the other peer will want to try the challenge and re-send its data).

The functions defined above may fail on some use cases (a connection is closed or some other error happens during the process). In case of error in any above function the process is ending in the "End of session" state. The way the error is reported (log, event, etc.) is outside this specification. If the peer wants to communicate again, it has to restart a new session.



4.3. Authentication

Authentication uses HMAC. The HMAC process is defined in the document: [#HMAC](#). The HMAC process requires a hashing algorithm. In this specification either MD5 [#MD5](#) or SHA1 [#SHA1](#) can be used.

The HMAC is computed in the following way:

$$\text{HMAC}(K_X, m) = H((K_X \text{ opad}) \circ H((K_X \text{ ipad}) \circ m))$$

and

$K_X = K_S$ or K_D depending of who generate the message, respectively the server or the device.
 $m = \text{protectedenvelope} \circ \text{nonce}$
 $\text{opad} = 0x5c5c5c \dots 5c$
 $\text{ipad} = 0x363636 \dots 36$

where

- H is a hash function (MD5 or SHA1), the hashing function is configured to either *hmac-md5* or *hmac-sha1*.
- *protectedenvelope* is the serialized byte stream of the protected envelope (the one that is encapsulated in a Bysant string).
- *nonce* is the current nonce. It was updated during the previous authenticated message. (it is not the value of the nonce field (next nonce) of the currently sent/receive envelope).
- *opad* and *ipad* are 16 bytes long.

The HMAC value (result of $\text{HMAC}(K_X, m)$) is transmitted into the `mac` public footer field of the Envelope. The length (in bytes) of the HMAC value is equal to the hash function output length.

The `mac` field must be present when sending a message that contains some data or a status code equal to 200. The `mac` field must not be present for challenge messages.

4.4. Encryption

4.4.1. Cipher key computation

The secret cipher key is computed as follow:

$$\text{CK} = \langle \text{HMAC}_{\text{MD5}}(K, \text{nonce}) \rangle_{\text{ckl}}$$

where:

- HMAC_{MD5} is the HMAC MD5 hash function as defined above.
- `ckl` is the cipher key length (depends on the cipher algorithm and parameters)
- $\langle k \rangle_n$ is a binary trunk function. It takes the first `n` bits of the value `k`.
- `K` is the password hash as defined in the [credentials section](#).
- `nonce` is the current nonce.

From the above key computation `CK`, it is clear that the hash function HMAC_{MD5} must produce a hash that has a length superior or equal to the length `ckl`.

When this is not the case (for instance AES cipher key is 256 bits wide) then another hash is concatenated so to extend the key length. The following formula is used:

$$\text{CK} = \langle \text{HMAC}_{\text{MD5}}(K, \text{nonce}) \circ \text{HMAC}_{\text{MD5}}(K, \text{nonce} \circ \text{nonce}) \rangle_{\text{ckl}}$$

4.4.2. Initial vector

The initial vector is set equal to the hash of the current nonce.

$$\text{IV} = \text{H}_{\text{MD5}}(\text{nonce})$$

where:

- H_{MD5} is the MD5 hash function.
- `nonce` is the current nonce.

4.4.2.1. Padding


When using an encryption mode that requires that the message size that is multiple of the encryption block size, then the plain text stream is padded in PKCS5 padding style (RFC2898, RFC1423).

In PKCS5 padding the input is padded with a padding string between 1 and 16 bytes to make the total length an exact multiple of 16 bytes. The value of each byte of the padding string is set to the number of bytes added due to the padding (for instance, when the message needs a padding of 7 bytes, 7 bytes of value 0x7).

CBC encryption mode requires padding.

CTR encryption mode does not require padding.

5. Sequence diagram

 this section of the specification is informative

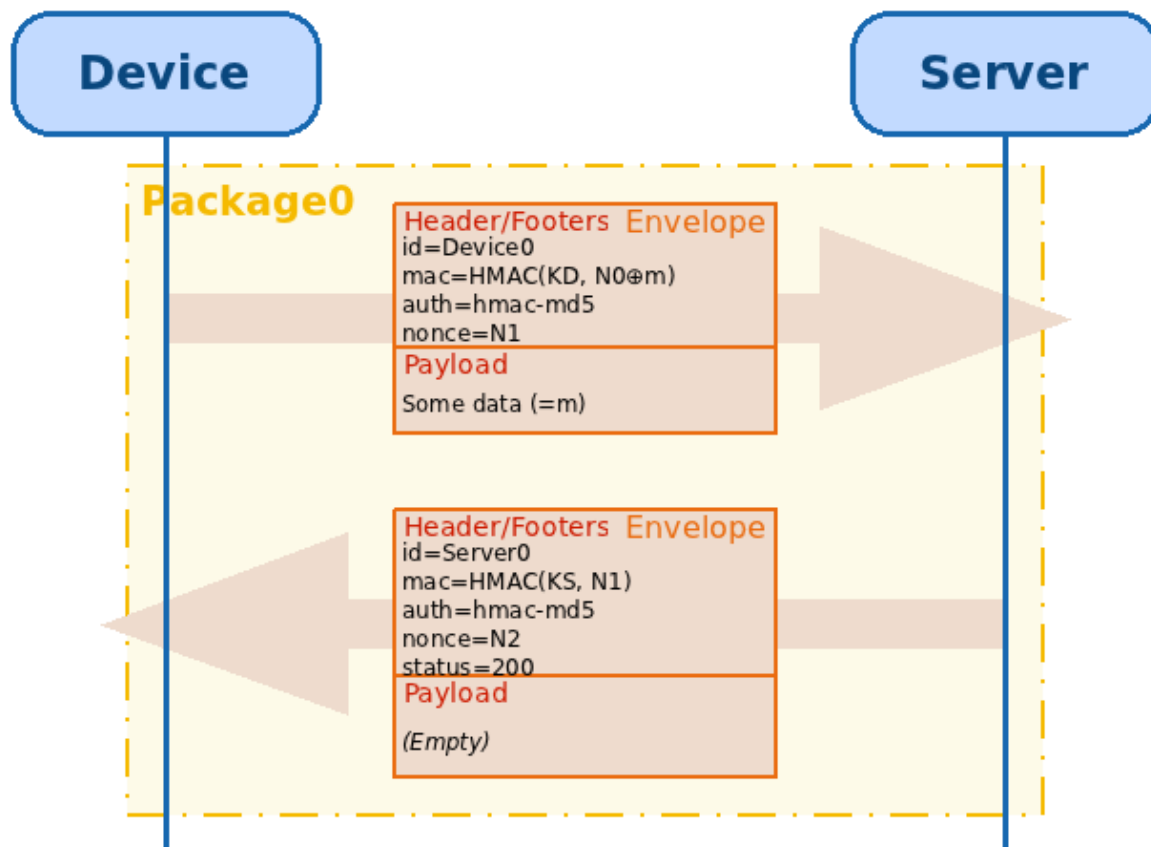
In the following diagrams:

- HMAC represents the above described HMAC function,
- KD and KS represents respectively the Device and Server secret key hash: $H_{MD5}(\text{username}:\text{password})$,
- NX (where X is a number) represents nonce values,
- m is the payload contained in the current envelope,
- When present, N0 is a nonce that was received into the previous authenticated message, or, if this is the first communication, N0 was provisioned before.
- For simplification reasons, nested envelopes are not represented here, instead all headers and footers appears at the same level.

5.1. Simple authenticated session

This is a simple authenticated session.

The Device sends an envelope containing the credentials and some data. The Server answers OK (status field) because the credentials are OK. The nonce is updated for each envelope because the peers are authenticated correctly.



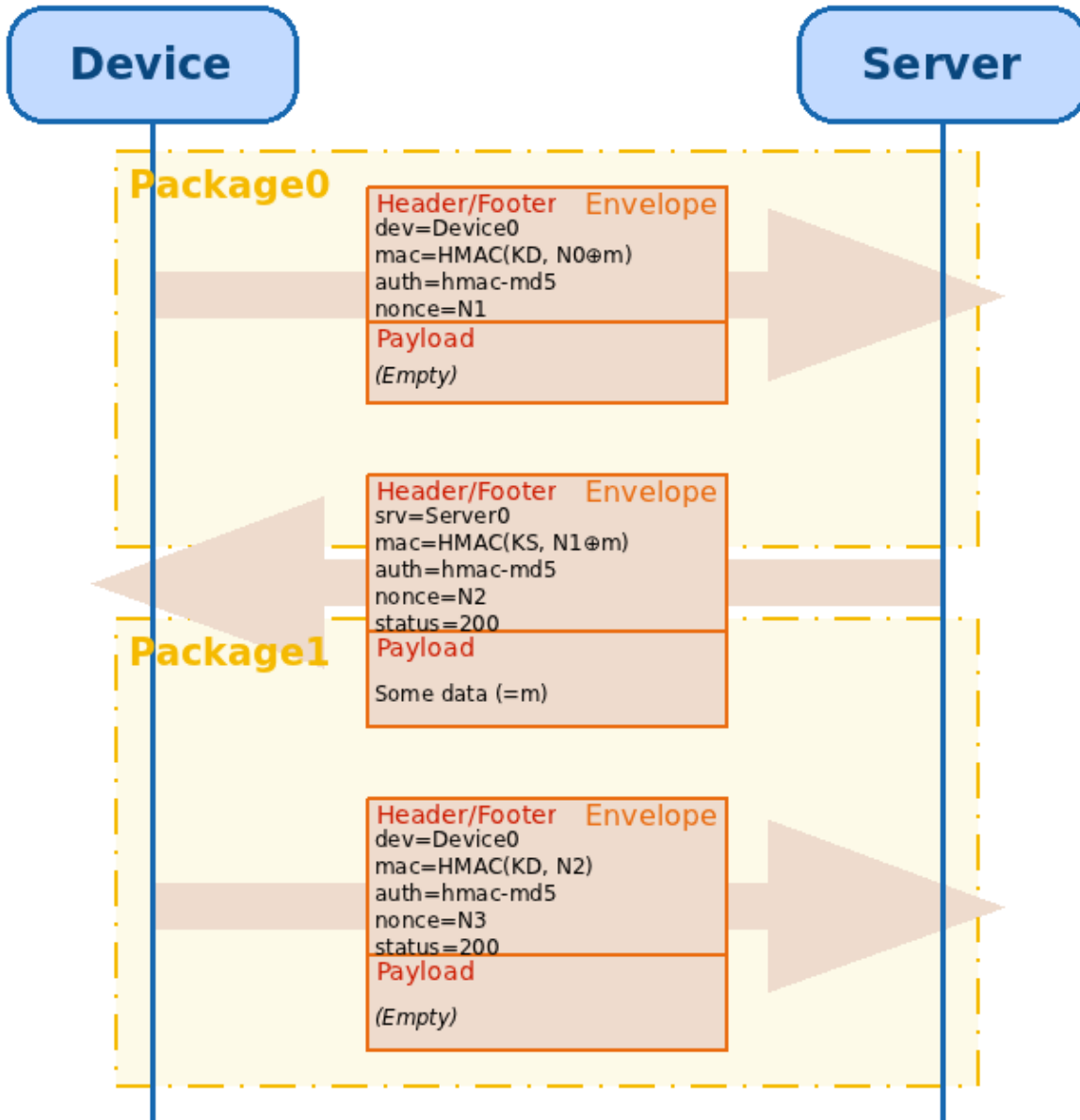
5.2. Authenticated session with server sending some data

This diagram represents two packages spanned on three envelopes.

First the Device connects to the Server, with no payload in the envelope in order to identify itself and receive an envelope from the Server.

The Server responds with an envelope containing the status field stating that the authentication is OK. Also the Server initiates a new package, adding some payload into the envelope.

Finally the Device replies with an envelope and the status field indicates that the authentication is OK.



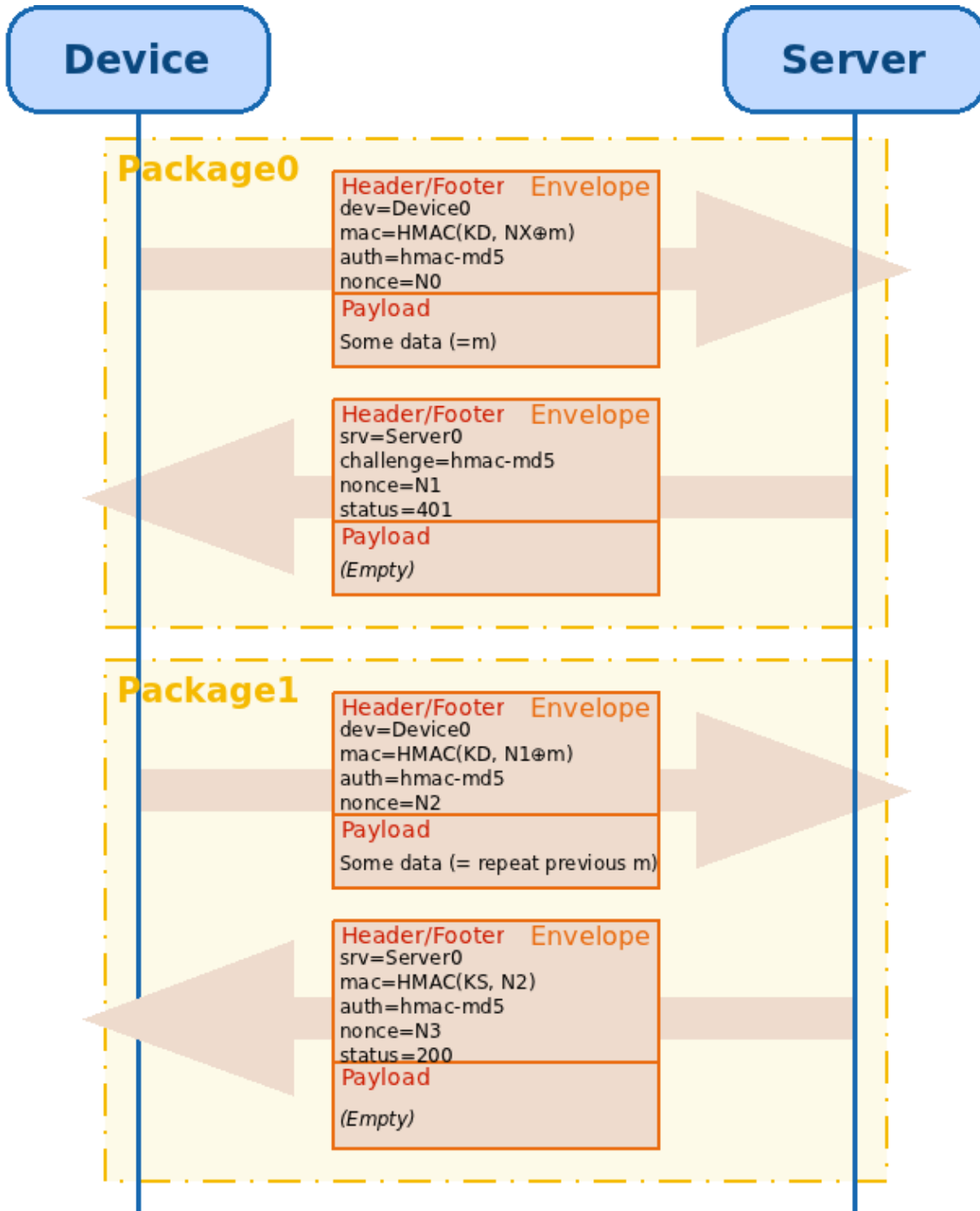
5.3. Server initiated challenge session

This diagram represents a server initiated challenge session. First the Device tries to send some data.

The Server replies "Not Authorized" (status=401) probably because the nonces are not synchronized (device used nonce NX which is not in sync with the server). The Server adds the field challenge in its reply to ask the Device to authenticate again. It also gives the next nonce to use for authentication.

The Device re-sends its data with the new authentication parameters. The Server replies OK. The nonces are updated.

In this work-flow the device sends twice the data. This can be a problem where bandwidth matters. In the case of big data payload, and / or when the nonce synchronization is not guaranteed, it is recommended to first make a no-data authenticated package (as in the previous sequence diagram. This will enable nonce re-synchronization and authentication validation with a minimal bandwidth cost.



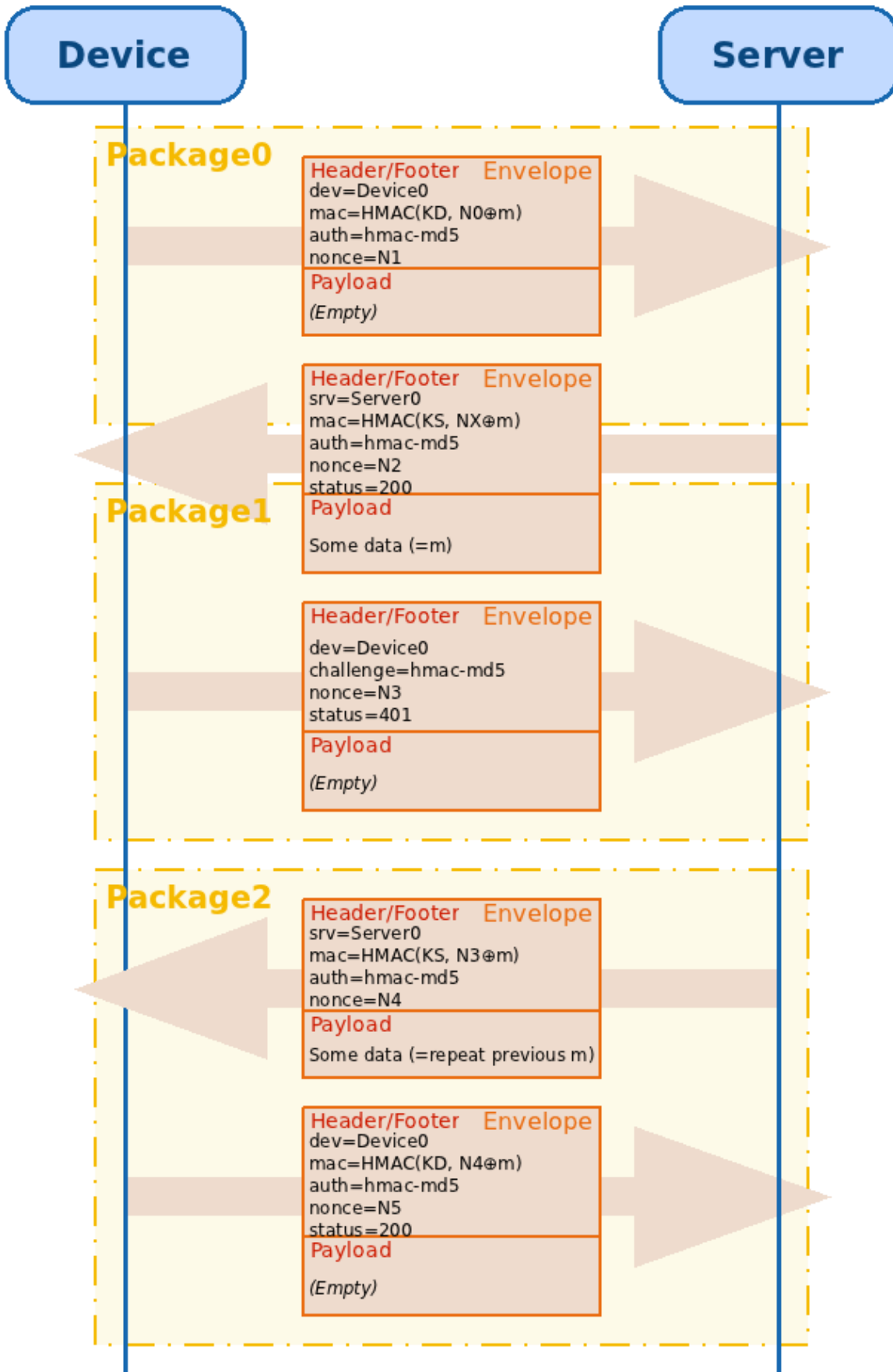
5.4. Client initiated challenge session

This diagram represents a client initiated challenge session. First the Device connects to the Server with an empty payload in order to trigger a Server to Device exchange.

The Server acknowledges the package (status = 200) and start a new package sending some data to the Device. However it uses a wrong nonce (NX) that is not in sync with the Device nonce.

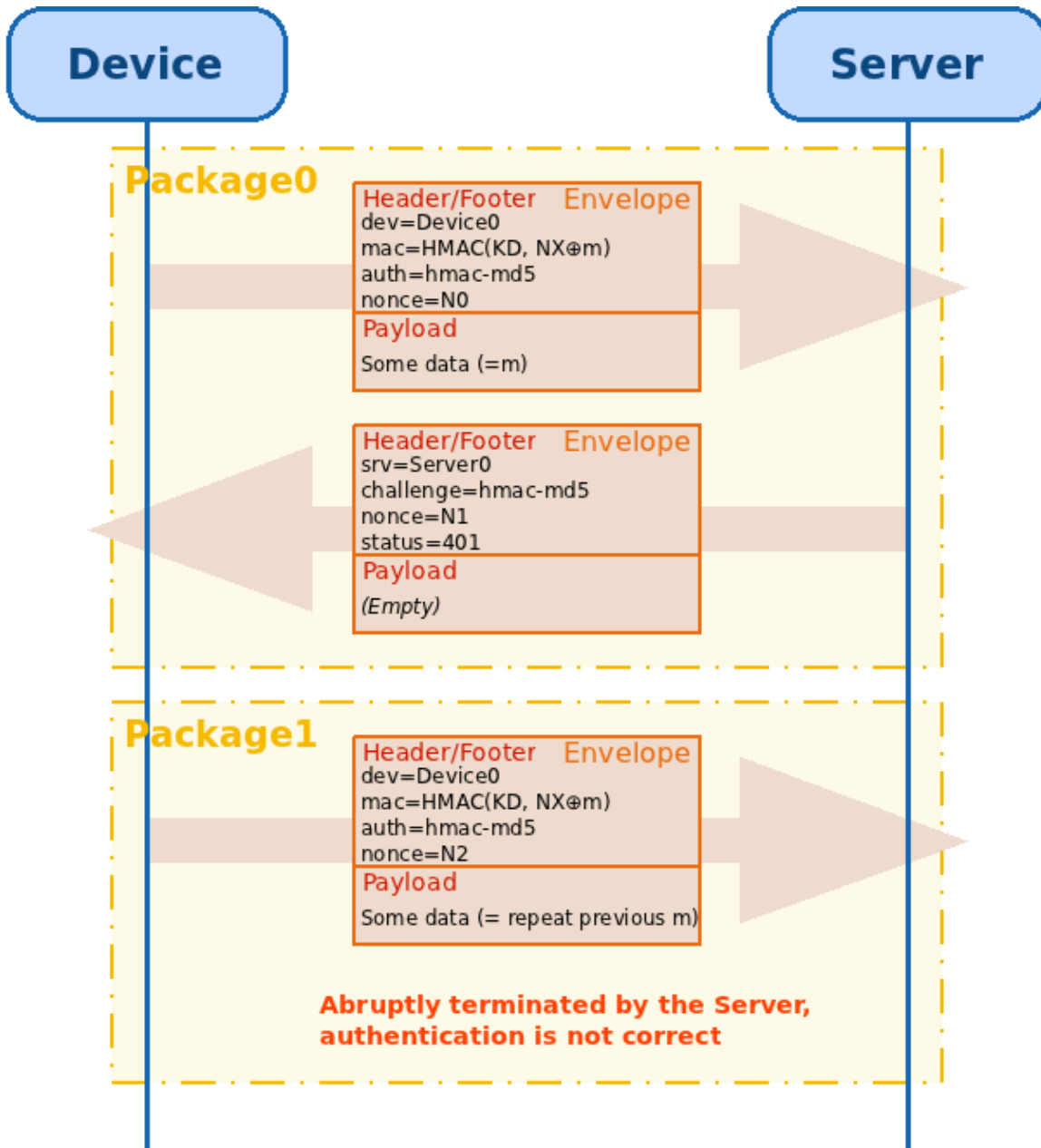
The device replies "Not Authorized" (status=401) because the nonces is not synchronized. The Device adds the field challenge in its reply to ask the Server to authenticate again. It also gives the next nonce to use for authentication. The Server re-sends its data with the new authentication parameters. The Device replies OK. The nonces are updated.

This work flow is shown here as an example but it will happen very seldom or never in the real life. Indeed the Device just send the nonce in the initial request, the Server should not loose the nonce to use in the same session.



5.5. Failed challenge session

This diagram represents challenge and authentication failure session. First the Device tries to send some data. The Server replies "Not Authorized" (status=401) probably because the nonces are not synchronized (device used nonce NX which is not in sync with the server). The Server adds the field challenge in its reply to ask the Device to authenticate again. It also gives the next nonce to use for authentication. The Device re-sends its data with new authentication parameters but they are still not correct. The Server closes the connection and the session is finished. Because no authentication was successful and the challenge failed, the nonces are not updated.



6. Denial of service protection

6.1. Authentication before accepting a lot of data

The hmac computation can take some time on big messages.

The attack scenario could be: an attacker could try to send a lot of huge well formed envelope. The authenticating

peer would need to compute the mac key in order to authenticate the envelope.

In order to protect from this form of attack, a peer must reject an envelope that has a payload size bigger than `MAX_FIRST_PAYLOAD_SIZE`, if the other peer was not authenticated during the last `MAX_OPEN_TIME` seconds. When a lot of data must be exchanged, an initial empty package would open the channel for a limited period of time. `MAX_FIRST_PAYLOAD_SIZE` (in bytes) and `MAX_OPEN_TIME` (in seconds) should be adapted according to the performances of the system.

6.2. Ignore successive challenge messages

When the authentication fails, the peer initiate a challenge.

The attack scenario could be: an attacker make a lot of connections in order to force the authenticating peer to issue a lot of challenge message.

In order to protect from this form of attack, when a peer issue a challenge envelope, if the other peer reply with a non authenticated envelope, then the session should be immediately closed and no further challenge envelope sent.



A session cannot start with a challenge envelope. A peer should not reply a challenge envelope starting a new session.