Extensions for Financial Services (XFS) interface specification
End-to-End (E2E) for XFS/XFS4IoT
Programmer's Reference v1.0
Release Candidate

This CEN Workshop Agreement has been drafted and approved by a Workshop of representatives of interested parties, the constitution of which is indicated in the foreword of this Workshop Agreement.

The formal process followed by the Workshop in the development of this Workshop Agreement has been endorsed by the National Members of CEN but neither the National Members of CEN nor the CEN-CENELEC Management Centre can be held accountable for the technical content of this CEN Workshop Agreement or possible conflicts with standards or legislation.

This CEN Workshop Agreement can in no way be held as being an official standard developed by CEN and its Members.

This CEN Workshop Agreement is publicly available as a reference document from the CEN Members National Standard Bodies.
CEN members are the national standards bodies of Austria, Belgium, Bulgaria, Croatia, Cyprus, Czech Republic, Denmark, Estonia, Finland, France, Germany, Greece, Hungary, Iceland, Ireland, Italy, Latvia, Lithuania, Luxembourg, Malta, Netherlands, Norway, Poland, Portugal, Republic of North Macedonia, Romania, Serbia, Slovakia, Slovenia, Spain, Sweden, Switzerland, Turkey and United Kingdom.

Warning
This document is not a CEN Workshop Agreement. It is distributed for review and comment. It is subject to change without notice and may not be referred to as a CEN Workshop Agreement. Recipients should notify the committee of any relevant patent rights of which they are aware and to provide supporting documentation.

EUROPEAN COMMITTEE FOR STANDARDIZATION
COMITÉ EUROPÉEN DE NORMALISATION
EUROPÄISCHES KOMITEE FÜR NORMUNG

Management Centre: rue de Stassart, 36 B-1050 Brussels

Table of Contents

Revision History

Revision History:

Initial release	December, 2021	End-to-End (E2E) for XFS/XFS4IoT Specification.

1 References

1. ANS X9 TR-34 2019, Interoperable Method for Distribution of Symmetric Keys using Asymmetric Techniques: Part 1 – Using Factoring-Based Public Key Cryptography Unilateral Key Transport
2. ANSI - X9.143, Retail Financial Services Interoperable Secure Key Block Specification

2 Introduction

2.1 XFS End-to-End (E2E) Overview

The XFS End-to-End (E2E) commands and events are defined to allow host to device authentication of vulnerable commands that must be protected from malicious attack. ‘E2E’ is the abbreviation for ‘End-to-End’ and refers to the bidirectional transmission of secure data between a device and a host.

It should be noted that XFS supports a Windows C based environment, whereas XFS4IoT is designed for multiple Operating Systems and programming environments. However, the E2E data flow and token format description in this document are the same for both architectures.

This document provides a general description of how E2E is used in XFS, for a description of exact syntax the relevant XFS/XFS4IoT documentation should be used. Note that for the purposes of this document the term ‘XFS’ refers to both XFS 3.x and XFS4IoT based architectures.

2.1.1 XFS E2E General description

A key priority for XFS is to improve security of the entire environment where it is used. This means securing not only the interface between the service and the device, or the interface between the client and the service, but providing security all the way from one end of an operation to the other.

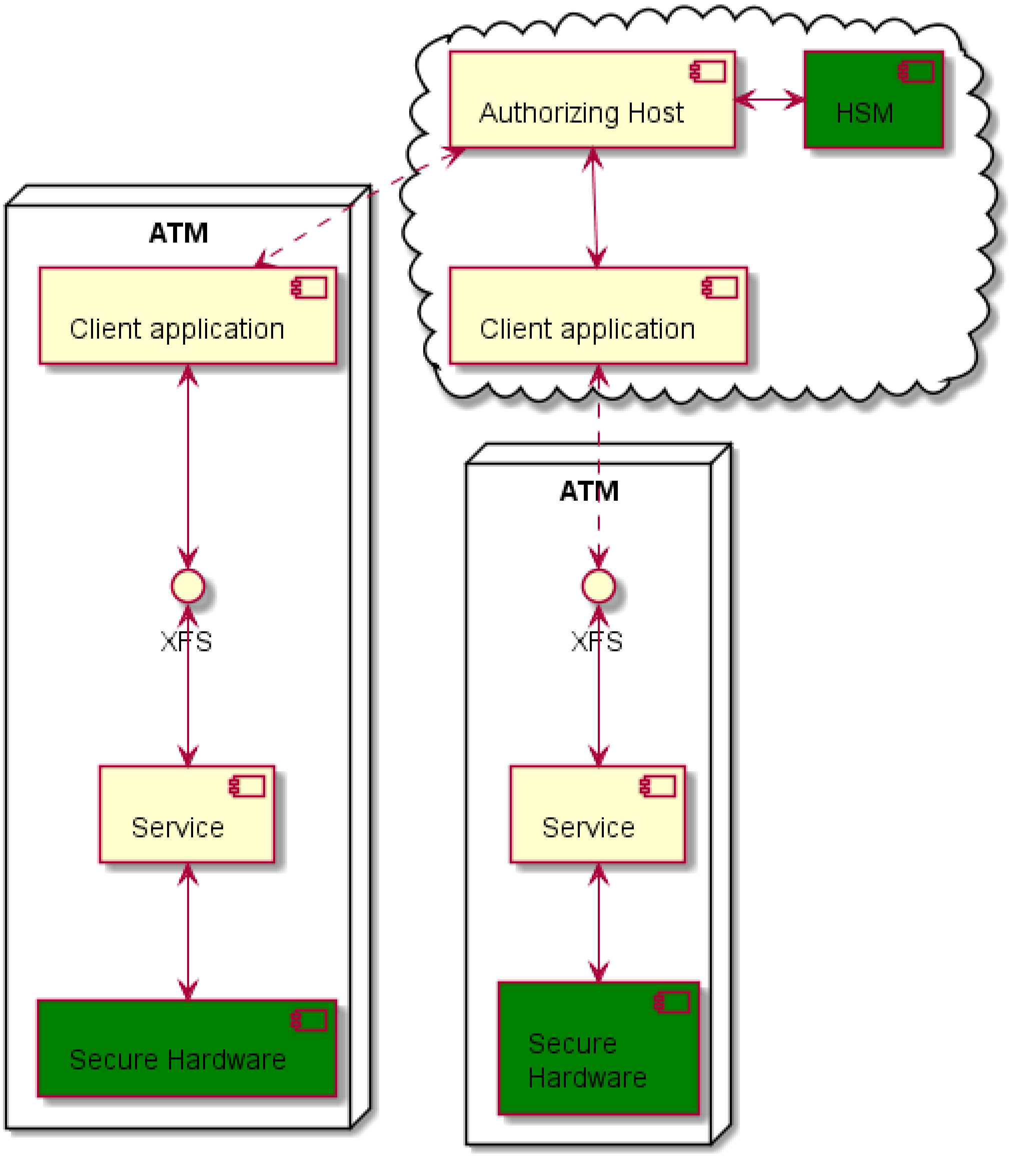
For example, during a cash dispense operation the transaction will first be authorized by an authorizing host which represents the owner of the cash in the device. That host will communicate through various other systems to the client application, the client application will communicate with the service and the service will finally communicate with the device. Any part of that process is vulnerable to an attack which could lead to the wrong amount of cash being dispensed. XFS E2E has been designed to block attacks at any point between the authorizing host and the dispenser hardware.

Both data 'integrity' like this, and 'confidentiality' for things like card data, are important. E2E focuses on integrity since confidentiality is covered by other mechanisms, such as TLS encryption of network messages.

3 General E2E sequence

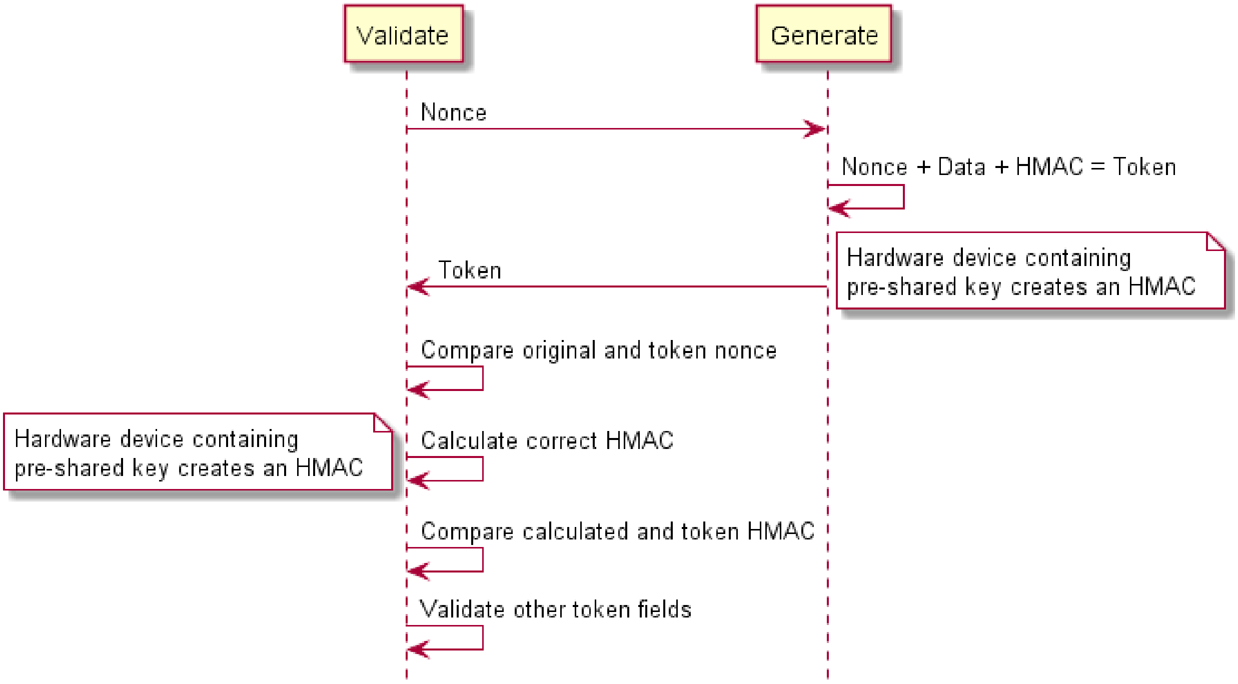
E2E security involves communicating facts between two end points and validating that those facts have not been changed or tampered with. Typically, this would be between a Hardware Security Module (HSM) in a data-center, and a Hardware Security Element (HSE) built into a device such as a cash dispenser. The method of communication between these two ends does not matter since any tampering with the data will be detected.

The following diagram shows a typical system with the secure hardware end points in green. It shows two possible architectures, one with the client application running locally on the ATM, and the other with application running in the cloud:



The general sequence for an E2E security operation involves one system generating a token, and another system validating that the token has not been changed. This sequence looks roughly the same in both directions; for example, an Authorizing Host can create a 'Dispense Token' and the Dispenser will validate that token before dispensing cash. In the opposite direction, the dispenser can create a 'Present Status' token to protect information about the presented cash and the client can validate that the token is valid and has not been tampered with.

In both directions the sequence looks like this:


Validate
Generate
Nonce
Nonce + Data + HMAC = Token
Hardware device containing pre-shared key creates an HMAC
Token
Compare original and token nonce
Hardware device containing pre-shared key creates an HMAC
Calculate correct HMAC
Compare calculated and token HMAC
Validate other token fields

4 E2E Tokens

XFS implements E2E security using strong cryptography, such as HMAC values. Secret keys are shared between endpoints, ideally by using TR34 and X9.143 (Ref 1). The use of public key RKL ensures that only the correct endpoints have access to the E2E transactions.

Note that X9.143 is referenced for the loading of symmetric keys. This is because TR31 is a non-binding ‘technical document’, whereas X9.143 is a published standard.

The data relevant to the E2E transactions is separated out from the normal clear text properties passed in the messages and is included in different "token" properties in the messages. Each token property is defined in the relevant interface documentation. For example, there is a "dispense token" for cash dispense actions and a "present status" token to protect the information about the last presented cash operation. These are defined in the relevant section of this document.

Tokens are encoded as a simple string which contains all of the information needed for that token, and also the information needed to keep it secure such as the HMAC value. Keeping all the information together in this way ensures that a single HMAC can protect all of the data, and there is no possibility that part of the data could be changed.

Keeping the token as a simple string means that it is easy to handle for low-power hardware. For example, the token may need to be checked by embedded firmware, even if the service is running on a front-end machine. To be fully E2E it must be checked on the hardware, so the format is kept simple.

The general format of a token is a string with a set of key=value pairs all comma separated and UTF8 encoded. The first pair will give the nonce. The last value will be a SHA256 HMAC, The UTF8 encoding is important so that the HMAC is consistent.

There will be multiple other key=value pairs, separated by commas. NONCE is always the first key, and HMAC256 is always the last key. All other keys can appear in any order. Key names are always upper case.

Tokens will not contain any extra whitespace.

To avoid possible parsing errors, binary data in tokens including the HMAC is encoded as simple hex values, rather than the normal BASE64 encoding. This is because BASE64 uses the "=" character which could be confused for the key=value separator. Hex encoded data will be all upper case.

For similar reasons, the characters "=" and "," must never be used in any value data, including custom values. If custom key/value pairs are used, then care must be taken that the value never contains those two characters.

The key=value pairs define what the token is used for. For example, the CDM service class in 3.x or Cash Dispenser interface in XFS4IoT uses "DISPENSE1" and "PRESENTSTATUS" keys in different Tokens. The different key names ensure that tokens with different uses can not be reused by an attacker. "value" is the actual value of the data being protected and will be different for each operation.

The set of standard key names and format for each value is defined in the relevant specification for each token type, including which keys are required and which are optional.

It is also permitted to include custom key values in a token. For example, a hardware dependent error code might be included. Unknown keys will be included in the HMAC calculation, but otherwise ignored. There must not be a dependency on custom keys. Care should be taken to avoid name clashes between keys, maybe by using a vendor's name in the key name. For example, if the Acme Corporation wants to include "ERRORCODE" as a custom key name then they should call it something like "ACMEERRORCODE" and not "ERRORCODE".

The total token length will be limited to 1024 bytes to avoid the risk of buffer overflows. Any token longer than this will be treated as invalid data. The limit is in bytes not characters since UTF8 characters may include multiple bytes.

4.1 Token Keys

E2E security tokens are made up of key=value pairs. There are various key names that are common to all types of tokens, plus key names which are only valid for specific token types. The following key names are valid for all tokens.

- **NONCE** : The token nonce, expressed as a HEX encoded value which was initially exchanged between the end points and is used to ensure that every token is different and to avoid replay attacks. The nonce will always come at the start of the token (so that there is not too much constant leading data - this is important for security).
  The nonce value must meet the cryptographic requirements for a nonce. It must be non-repeating - that is, the same nonce value must never be used twice. Code creating a nonce value should be carefully reviewed to make sure this is true.

  There are two simply ways of guaranteeing this:

  - An incrementing integer can be used if it always increments between every token. This must be true across restarts and power fails. The counter must not reset and repeat the same values. Note that it doesn't matter that the value can be predicted. Also note that the length of an integer nonce doesn't matter as long as it never repeats. This is different to a random nonce (below).

  - A strong random number can be used if the random number never repeats. This can be easy if there is a hardware random number generator available. If a pseudo-random number is used, then the seed value needs to be carefully picked so that the seed never repeats (which would cause the random number to repeat.) Needing to track unique seed values might mean it's easier to simply use a persistent integer counter. Also, the chance of two random numbers matching needs to be no more likely than the chance of two hash values matching, which means that if a random number is used it must be 128 bits, to match the SHA256 length.

  Note that X9.143 (Ref 1) is referenced for the loading of symmetric keys, rather than TR31 (Ref 1). This is because TR31 is a non-binding ‘technical report’, whereas X9.143 is a published standard.

  The value of the current nonce must be cached so that it can be checked against tokens, for example by the firmware for incoming tokens. The nonce should not be persistent though - the current value will be lost after a power cycle. After a power failure the nonce value will be cleared and any operations that include a token will fail with the error code indicating that there is no token nonce. Note that this is possible if the service is running on a different machine to the protected device - for example, if the service is running on a PC connected via USB to a dispenser then the dispenser could lose power, but a client could still have a valid connection to the service. In this case the client could use a token with an old nonce but will receive an error code indicating that the nonce is invalid.

- **TOKENFORMAT** : The version number of the token format. Currently this will always be "1"

- **TOKENLENGTH** : The total number of bytes in the token, including the HMAC value, in decimal. This value will be exactly four digits and include leading zeroes as required. Since this value has to include the length of itself, making it fixed length makes it easier to calculate. Note that this is bytes and not characters, since UTF8 characters may contain multiple bytes.

- **HMACSHA256** : The HEX encoded HMAC of all the preceding data up to and including the last equals after the HMACSHA256 key. The HMAC is always the last value - this makes it easy to calculate the HMAC since it can be calculated over all other data, converted to hex, then appended to the string. The HMAC will use SHA256 as the algorithm. The hex encoded data will be all upper case.

4.2 Token Examples

The general token format looks something like this:

NONCE=<noncevalue>,TOKENFORMAT=1,TOKENLENGTH=<Length>,<KEY>=<value>,HMACSHA256=<HMAC>

For example, a dispense token might be:

NONCE=254611E63B2531576314E86527338D61,TOKENFORMAT=1,TOKENLENGTH=0164,DISPENSE1=50.00EUR,HMACSHA256=CB735612FD6141213C2827FB5A6A4F4846D7A7347B15434916FEA6AC16F3D2F2

The value used to calculate the HMAC is

NONCE=254611E63B2531576314E86527338D61,TOKENFORMAT=1,TOKENLENGTH=0164,DISPENSE1=50.00EUR,HMACSHA256=

A HMAC for this data, with SHA256 and a key of 112233445566778899AABBCCDDEEFF112233445566778899AABBCCDDEEFF, is:

CB735612FD6141213C2827FB5A6A4F4846D7A7347B15434916FEA6AC16F3D2F2

An example for including this in a XFS4IoT CashDispenser.Dispense command message would be as follows for JSON:

```
{
  "header": {
    "type": "command",
    "name": " CashDispenser.Dispenser",
    "requestId": 456
  },
  "payload": {
    ...
    "dispenseToken":
"NONCE=254611E63B2531576314E86527338D61,TOKENFORMAT=1,TOKENLENGTH=0164,
DISPENSE1=50.00EUR,HMACSHA256=CB735612FD6141213C2827FB5A6A4F4846D7A7347B154
34916FEA6AC16F3D2F2"
    ...
  }
}
```

Similarly, the Present Status might include a PresentStatus token:

**NONCE=1414,TOKENFORMAT=1,TOKENLENGTH=0268,DISPENSEID=CB735612FD6141213C2827FB5A6A4F4846D7A7347B15434916FEA6AC16F3D2F2,
DISPENSED1=50.00EUR,PRESENTED1=YES,PRESENTEDAMOUNT1=50.00EUR,RETRACTED1=NO,
HMACSHA256=55D123E9EE64F0CC3D1CD4F953348B441E521BBACCD6998C6F51D645D71E6C83**

For example, in XFS4IoT this would be included in the CashDispenser.GetPresentStatus completion message as:

```
{
 "header": {
  "requestId": 765,
  "type": "completion",
  "name": "CashDispenser.GetPresentStatus"
 },
 "payload": {
  ...

  "denomination": {
   "currencies": [
        {
          "currencyID": "EUR",
          "amount": 50
        }
      ],
    ...

    },
    "presentState": "presented",
    "token" : "NONCE=1414,TOKENFORMAT=1,TOKENLENGTH=0268,
DISPENSEID=CB735612FD6141213C2827FB5A6A4F4846D7A7347B15434916FEA6AC16F3D2F2,
DISPENSED1=50.00EUR,PRESENTED1=YES,
PRESENTEDAMOUNT1=50.00EUR,RETRACTED1=NO,HMACSHA256=55D123E9EE64F0CC3
D1CD4F953348B441E521BBACCD6998C6F51D645D71E6C83"
  }
}
```

Note: Token strings never have new lines or any white-space. New lines are included in examples only to make them more readable.

In this example €50.00 was moved from the cassettes, possibly to a stacker, and €50.00 was then presented to the customer. The notes were not retracted so it should be assumed that the customer could have taken the notes.

5 E2E Encryption Key Management

To ensure strong security, secret symmetric encryption keys are shared between different endpoints.

The keys used for E2E encryption have fixed names. They are *XFSAuthenticateHost* and *XFSAuthenticateDevice*, where *XFSAuthenticateHost* is used by the host to create an HMAC, and *XFSAuthenticateDevice* is used by the device to create an HMAC. Both keys must be shared between both endpoints so that HMAC values can be both calculated and checked.

Ideally sharing the keys will be done using public key encryption, as defined in TR34. However, in some cases it may be necessary to pre-load keys into hardware in some other secure way, for example in legacy hardware that can not support TR34 (Ref 1). The security of the whole system is only as good as the security of these keys, so it is vital that this is done in as secure a way as possible.

The key details are defined by the X9.143 specification which defines both key data and details such the intended use. This includes the algorithm that the key can be used with. Since X9.143 defines the algorithm there is no need for algorithms to be negotiated in any other way - the two end points will effectively agree the algorithm to use by loading the relevant keys. The KeyblockHeader (including the optional blocks) for each of the keys should be included. For examples of the KeyBlockHeader see section 8 of this document.

Encryption keys must be long enough to ensure security when used with a particular algorithm. For example, where SHA256 is used for the HMAC value, the key must be a minimum of 128 bits and up to 256 bits long to match the algorithm.

For simplicity, communication in each direction will be handled separately. So, for example, there will be a separate key for tokens passed in each direction. Also, there will be a separate nonce in each direction.

For XFS4IoT, the details of the key management are covered by the shared Key Management interface. Any service that implements E2E security will implement this interface as part of its interface. In XFS 3.x, key management is handled using the key management interfaces in the PIN device class.

6 Unique Messages and Replay Attacks

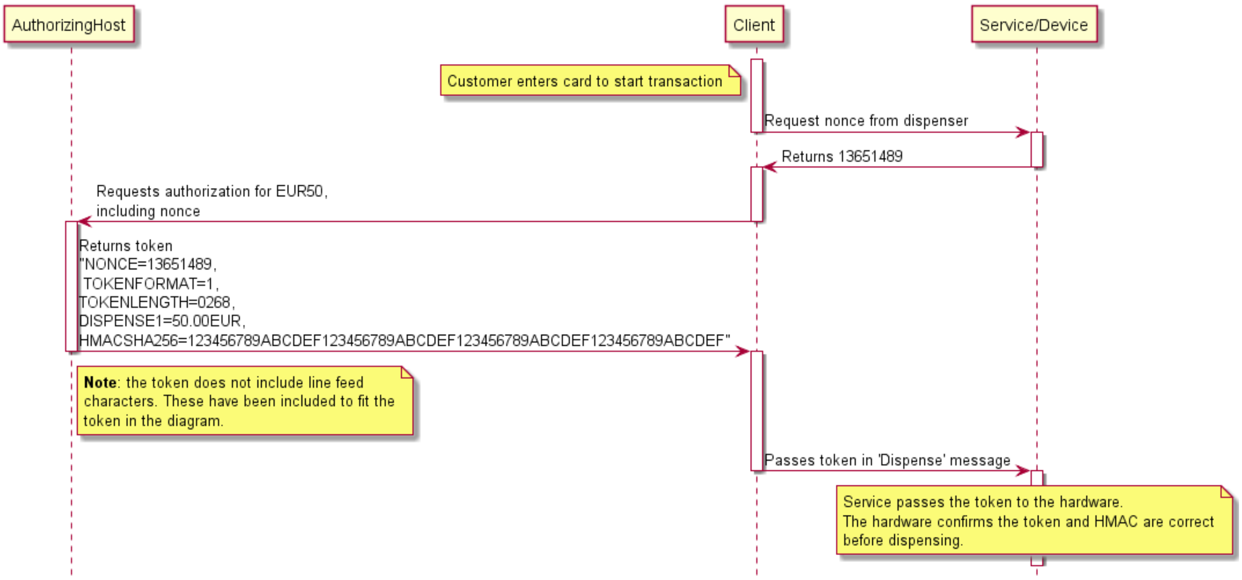
To avoid 'replay attacks', where an attacker reuses an old message to replace a new one, it is important that all individual tokens are unique, and the same data is not used multiple times. For example, it is common to have a dispense token for "10EUR" so that value (and its HMAC) will be the same for many transactions and could be reused by an attacker. To avoid this a "Nonce" value is included in each token. The nonce will be different for each transaction. This guarantees uniqueness.

There will be a different nonce for tokens passed in each direction. That is, there will be a command nonce and response nonce. Each nonce must be generated and checked at the same end of the communication, so the command nonce must be generated and checked by the device. The response nonce must be generated and checked by the client. (Ideally in the host HSM.)

To fully avoid replay attacks the nonce must be agreed between the endpoints before it is used. An extra command on the interface is called by the client to fetch a new command nonce. This nonce is then remembered by both end points and included in each command token. Similarly, a response nonce must be generated by the client/host, passed to the service, and included in all response tokens.

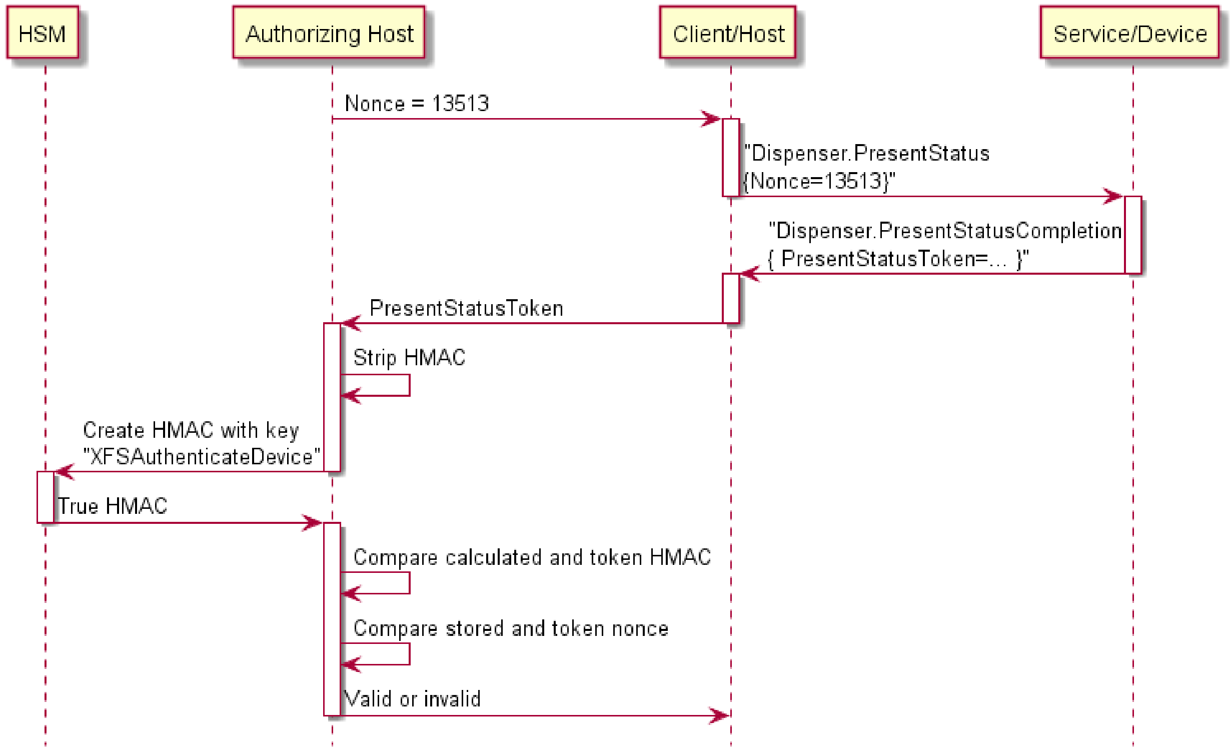
6.1 Example: A classic dispense operation

The following example shows how E2E security is used to protect a cash dispense operation during a classic ATM transaction.


AuthorizingHost
Client
Service/Device
Customer enters card to start transaction
Request nonce from dispenser
Returns 13651489
Requests authorization for EUR50,
including nonce
Returns token
"NONCE=13651489,
TOKENFORMAT=1,
TOKENLENGTH=0268,
DISPENSE1=50.00EUR,
HMACSHA256=123456789ABCDEF123456789ABCDEF123456789ABCDEF123456789ABCDEF"
Note: the token does not include line feed
characters. These have been included to fit the
token in the diagram.
Passes token in 'Dispense' message
Service passes the token to the hardware.
The hardware confirms the token and HMAC are correct
before dispensing.

Once a dispense has been performed it is important to accurately report the result. If an attacker can change the reported result, then they might fake an error to make it look like cash was not presented to the customer and tricking the banks into reversing the transaction - this is known as transaction reversal fraud.

To protect against this, the WFS_INF_CDM_PRESENT_STATUS (XFS 3.x) or CashDispenser.GetPresentStatus (XFS4IoT) commands return a token that can not be tampered with. Note that this token goes in the opposite direction, from the device to the host. This means that the nonce is now coming from the host rather than from the device. The nonce can be included in the command to get the present status without needing to call an extra command.

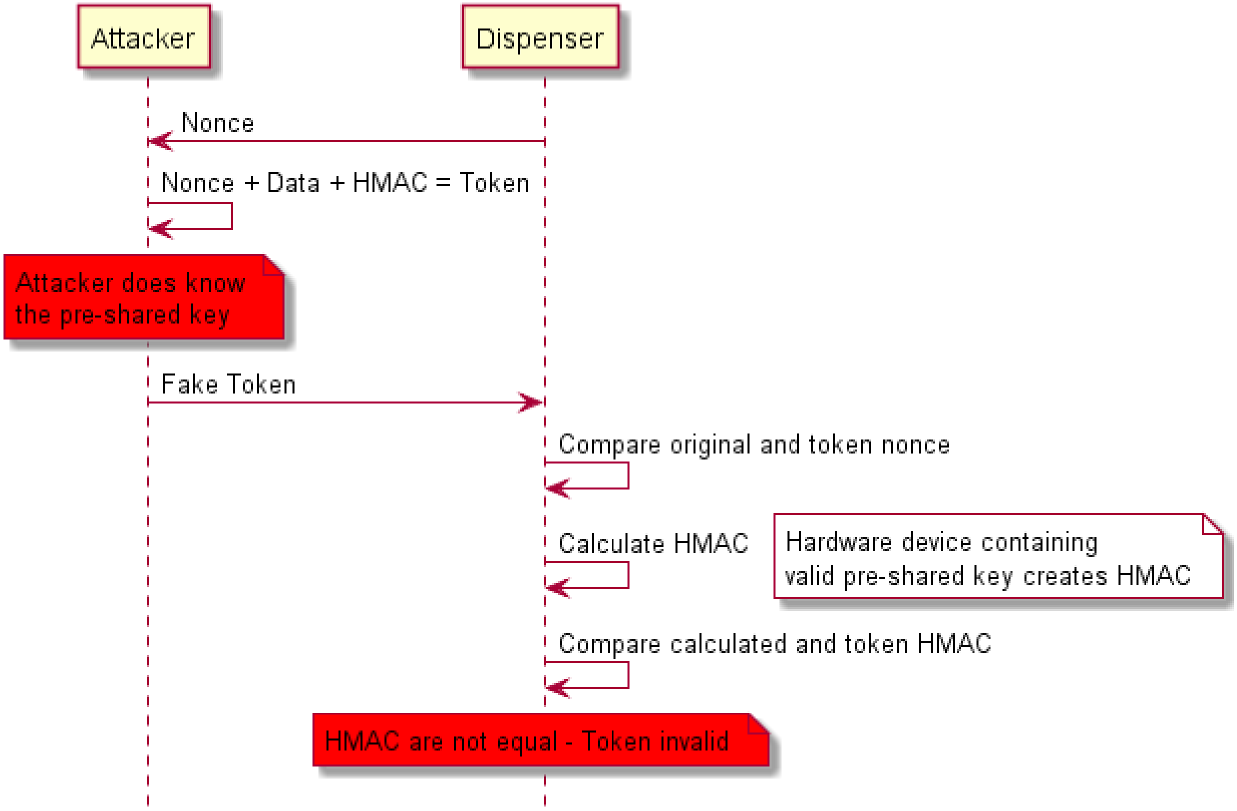


6.2 Example: Types of attacks that are blocked

Various types of attacks are blocked by E2E security. Some examples are given here.

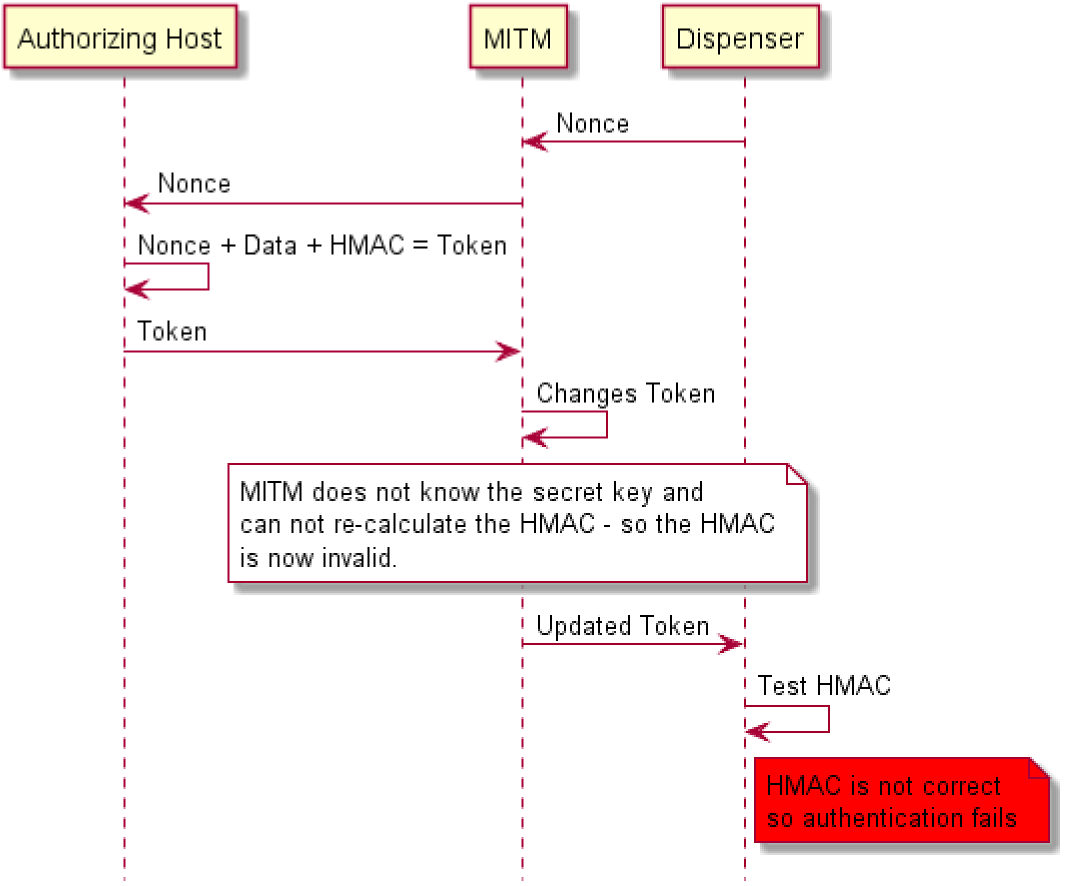
6.2.1 Black Box Attack

A fake client attempts to issue a dispense command without authorization. This is commonly known as a "black box" attack:


Attacker
Dispenser
Nonce
Nonce + Data + HMAC = Token
Attacker does know
the pre-shared key
Fake Token
Compare original and token nonce
Calculate HMAC
Hardware device containing
valid pre-shared key creates HMAC
Compare calculated and token HMAC
HMAC are not equal - Token invalid

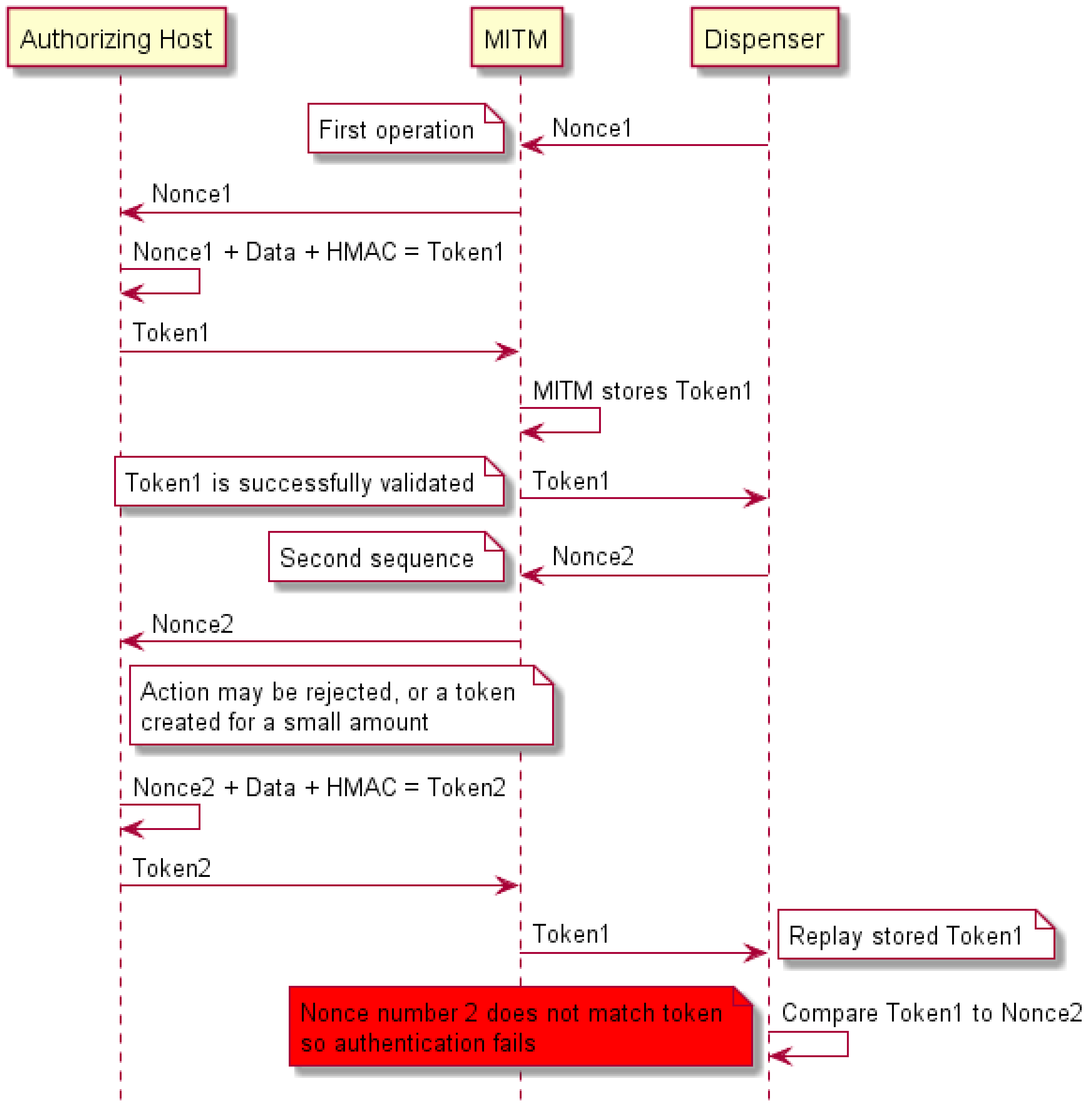
6.2.2 Man in the Middle attack

An attacker with control over communications tries to change details, for example to increase the amount of cash being dispensed. This is known as a "Man in the Middle" attack:



6.2.3 Replay Attack

An attacker with control over communication stores a token and attempts to use it a second time. This is known as a "replay attack":


Authorizing Host
MITM
Dispenser
First operation
Nonce1
Nonce1
Nonce1 + Data + HMAC = Token1
Token1
MITM stores Token1
Token1 is successfully validated
Token1
Second sequence
Nonce2
Nonce2
Action may be rejected, or a token created for a small amount
Nonce2 + Data + HMAC = Token2
Token2
Token1
Replay stored Token1
Nonce number 2 does not match token so authentication fails
Compare Token1 to Nonce2

7 E2E Token Formats

This section defines the E2E Token formats used in XFS. This section defines the keys use for token formats used in XFS. Keys in this section are in addition to the NONCE, TOKENFORMAT, TOKENLENGTH and other common keywords described in section 4.

7.1 Dispense Token Format

The following describes the fields required to validate the dispense and guarantee that the details have not been tampered with. This token will follow the standard token format, defined in this documentation, and will contain the following keys:

DISPENSE<*n*>=<*AMOUNT*><*CURRENCY*>”

The format for these keys is as follows:

Key	Description
DISPENSE<n>	This is a logical key starting with 1 and incrementing by 1.
AMOUNT	The amount as a number string, including the fractional part will be defined by the ISO currency. The maximum value to be dispensed. The decimal character will be ".".
CURRENCY	This field will be defined using the ISO currency code. The currency code will be upper case.

As an example, "123.45EUR" will be €123 and 45 cents. The DISPENSE key may appear multiple times with a number suffix. For example, DISPENSE1, DISPENSE2, DISPENSE3. The number will start at 1 and increment. Each key can only be given once. Each key must have a value in a different currency. For example:

DISPENSE1=100.00EUR,DISPENSE2=200.00USD

The actual amount dispensed will be given by the denomination. The value in the token MUST be greater or equal to the amount in the denomination parameter. If the Token has a lower value or the Token is invalid for any reason, then the command will fail with an invalid data error code.

7.2 Present Status Token Format

The following describes the fields required to validate the present status and guarantee that the details have not been tampered with. This token will follow the standard token format, defined in this documentation, and will contain the following keys. Note that all these keywords are mandatory, in order to give the maximum amount of information. The format for these keys is as follows:

Key	Description
DISPENSEID	HEX encoded HMACSHA256 value from the token for the last dispense operation. This is included so that the present status can be linked to a previously authorized dispense. If this doesn't match the expected value then the receiver of the token may assume that the transaction is suspect, and that the cash may have been accessible to the customer. If there was no dispense token this key will not be included - This may also mark the dispense as suspect.
DISPENSED<n>	The total value of a single currency that was removed from cassettes and possible stacked inside the machine ready to present. This does not include any notes moved to the reject cassette. This will be a number string that may contain a fractional part. The decimal character will be ".". The value, including the fractional part, will be defined by the ISO currency. The number will be followed by the ISO currency code. The currency code will be upper

	case. For example, "123.45EUR" will be €123 and 45 cents.
PRESENTED<n>	May be "YES" or "NO" (upper case.) This will be YES if the notes could at any time have been accessible outside the machine and may have been tampered with. If the notes were never accessible outside the machine and can not have been tampered with then this value will be NO.
PRESENTEDAMOUNT<n>	The total value of a single currency that was presented outside of the machine. The format is the same as for DISPENSED1.
RETRACTED<n>	May be "YES" or "NO". If notes were accessible outside of the machine and were then retracted back into the device, then this value will be YES. If notes were never presented, or all notes that were presented were not retracted, then this will be NO.
RETRACTEDAMOUNT<n>	If notes are counted during a retract this will be the amount retracted. If the notes aren't counted, or the value of the notes retracted is not reliably known for any reason, this will be "?". For example: "RETRACTEDAMOUNT1=123.45EUR" €123 and 45 cents was retracted and counted. "RETRACTEDAMOUNT1=?" Notes may have been retracted, but the value of the notes can't be guaranteed.

The following is an example of how these keys and values may look in the formatted token:

"NONCE=1414,TOKENFORMAT=1,TOKENLENGTH=0268**,DISPENSEID=CB735612FD6141213C2827FB5A6A4F4846D7A7347B15434916FEA6AC16F3D2F2,**
DISPENSED1=50.00EUR,PRESENTED1=YES,PRESENTEDAMOUNT1=50.00EUR,RETRACTED1=NO,
HMACSHA256=55D123E9EE64F0CC3D1CD4F953348B441E521BBACCD6998C6F51D645D71E6C83"

Each numbered key may appear multiple times with a different number suffix. For example, DISPENSED1, DISPENSED2, DISPENSED3. The number will start at 1 and increment by 1. Each key can only be given once. Each key must have a value in a different currency. For example, DISPENSED1=100.00EUR,DISPENSED2=200.00USD

Note: In most cases, once currency has been presented and is accessible to a customer then the value of that currency shouldn't be relied on since the customer might take notes, replace notes with counterfeit etc. The value of any notes retracted back into the machine isn't reliable so RETRACTEDAMOUNT1 will be "?".

However, some devices may be able to test notes that are retracted are valid. This is possible with cash accepting hardware. The RETRACTEDAMOUNT value will only give an actual value if the notes have been checked by the hardware.

Note: Values in the PresentStatus token are the actual values. This is different to the token in the Dispense command where currency "up to" the token value may be dispensed. This doesn't apply to the PresentStatus token.

7.3 Multiple Dispense/Present Operations

Note that in the case where there is a physical limit on the number of notes that can be presented to the customer in one operation, multiple Dispense/Present commands are required. This means that multiple Dispense/Present commands with the same token must be permitted under E2E security. The requirements for this use case are as follows:

1. The nonce, and therefore the token, can only be implicitly cleared by the service when the total authorized value has been dispensed and presented.
2. If less than the total authorized amount is required to be dispensed and presented, then the nonce must be explicitly cleared by the application when the sequence of dispense/present commands have completed.
3. The Present Status token can only be returned once all of the Dispense/Present commands have completed.

4. If the command to get the Present Status is called with a response nonce before all of the requested notes have been presented, then it will return a sequence error. The command to get the Present Status can still be called without a nonce to get information on the last present without a token.
5. Once the sequence of dispense/present calls have completed, the nonce can be cleared either implicitly by the service (because the total authorized value has been dispensed) or explicitly by calling the command to clear the nonce (where the value to dispense and present is less than the authorized value). It is then valid to get the Present Status with a response nonce to get the total amount presented.
6. At the end of a sequence of multiple dispense/present operations with the same token, the command to get the Present Status will only report the details of the last present operation, but the token will cover the whole sequence related to the dispense token. This means the token and the Present Status data may be different.

8 HMAC Key Block Examples

This section describes some HMAC Key Block Examples.

An X9.143 Key block for HMAC Key is used for Host Based Authentication.
An X9.143 key block consists of a header, confidential data and an authentication value.

This Example is a 128bit key for HMAC using hash algorithm SHA256.

1. Header field (Version, length, usage, algorithm, mode, key version, exportability, no. of optional blocks, context*, reserved, 1st optional block ID (HMAC), optional block length, optional block data, , 2nd optional block ID (padding), optional block length, optional block data)
    a. TDEA wrapped key for Verification (XFS key name = XFSAuthenticateHost)

    B 0160 M7 H V 00 N 02 2 0 HM 06 21 PB 0A 000000

    b. TDEA wrapped key for Generation (XFS key name = XFSAuthenticateDevice)

    B 0160 M7 H G 00 N 02 2 0 HM 06 21 PB 0A 000000

    c. AES wrapped key for Verification (XFS key name = XFSAuthenticateHost)

    D 0160 M7 H V 00 N 02 2 0 HM 06 21 PB 0A 000000

    d. AES wrapped key for Generation (XFS key name = XFSAuthenticateDevice)

    D 0160 M7 H G 00 N 02 2 0 HM 06 21 PB 0A 000000

2. Confidential data (example of 128bit HMAC, allowable lengths are 128-256)

    0080 (length of key, 2 bytes)
    Key (128bit HMAC, 16 bytes)
    Key obfuscation pad (128 bits, 16 bytes) required only if key is < 256 bits
    Cipher pad (14 bytes)

    Total key data field = 48 bytes, 96 chars.
    Data field is enciphered by the KPEK to create the confidential data

3. Authentication Value

    MAC (32 bytes)
    Authentication value is computed over the header and confidential data

*****note context is defined in X9.143, TR31 has this as a reserved byte. Value of 0 is compliant with both TR31 and X9.143 and has a meaning of ‘no specific context’. Value of 2 means key is to be transported.

Appendix A. Diagram Source

Attached http://plantuml.com source for sequence diagrams. These can be loaded into various editors (e.g. VSCode, Atom) with an appropriate PlantUML extension or plugin installed, or online (e.g. www.planttext.com):.

UML Source.zip