# APPENDIX SECOM Test Project SAAB Report

## Participant Testbed description

The goal is to implement the interface as described in the SECOM specification. Due to firewalls some interoperability may not be possible.

The implementation is based on DotNet Core 3.1 using Microsoft Cryptography library for encryption and signatures. The idea is to use a different implementation than openSSL and bouncyCastle to verify interoperability.

To verify the implementation internally, data is also verified on Linux using openSSL.

## Participant Test report

Describe outcome within each Test Case. Outcome is both observations during creation of the testbed and observations on test runs using the testbed.

### Observations

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Number | Observation | Consequence/Proposal | Found when … | Reference in SECOM Document |
| STT-001 | Conversion of signature to HEX may give output with several rows (LF/CR in the resulting HEX string) | Unnecessary new line characters will be included in the resulting JSON message. | Following Test instructions |  |
| STT-002 | Data may need padding if shorter than signing key | So far this small datasets have not been tested, is there a usecase for this? | Reading Secom document | Page 85,  7.4.2 7.4.2 Encryption algorithm |
| STT-003 | When exchanging data per email the data should be changed into Base64 format. This can be done by adding the flag -base64 to the enc command. To skip line breaks also add the -A flag. | Always provide base64 data without line breaks in json.  Can be done by providing the –A parameter:  Example:  openssl enc -aes-256-cbc -base64 -A -in data.rtz -out data.rtz.enc64A -pass file:secretRandomKey.bin -salt | Running SECOM test cases in openSSL |  |
| STT-004 | The openSSL commands for encryption provided in the test cases use passphrases and not keys and init vectors. The Version of openSSL used can affect this | Using different version of openSSL and passphrase results in different key and passphrase generated, making it very hard to use when testing interoperability. Further, the passphrase option used as described results in a warning in openSSL v1.1.1 and forward. It is suggested to use the -pbkdf2 option instead. Alternatively the key and iv could be set directly. |  | All references to openSSL commands with passphrase option in the WG17 SECOM document,  -  -  -  -  All the passphrase commands in en encrypted testcases documents:  - Case 3 |
| STT-005 | Examples show Base64 data being signed and not the original data | The original data should be signed as this is what should be protected from end to end.  (If only signing the payload) | Testing encryption in testproject according instructions |  |
| STT-006 | The signature in the Ack has the format “S100\_DigitalSignatureValue” | Suggest to use same format for Acknowledgement signature as for upload signature | Reading document and working on What to sign task | Page 37,  Table 5 13 |

### Test Case 1 Data protection (signing) of unclassified data

#### Results, observations, discussions

Exchange of a single signed non encrypted dataset (rtz) is working fine between SMA/SAAB

Some observations:

Using Base64 increases the size of the payload and is something to be aware of when sending larger data.

Exchange of public key can be done using certificate or an unsigned public key. Both have been tested and work to verify the data signature.

## Passphrase or key/iv: (STT-004)

Depending on the version of openSSL used, the outcome will differ.

To compare three different version of openSSL were used:

* openSSL 1.0.2k 26 Jan 207 (The version provided with the testTool)
* openSSL 1.1.0l Sept 2019 (Version bundled with Debian9)
* openSSL 1.1.1g Apr 21 2020 (latest recommended at time of writing)

Example 1:

Using the passphrase “password” as text (using a file with password is no different)

Test tool: (version 1.0.2k  26 Jan 2017)

openssl enc -aes-256-cbc -S 9AFC4D99FF13226E -in Data.rtz -out Data.rtz.enc -pass pass:password  -P

Output:

salt=9AFC4D99FF13226E

key=DF72F51BA575BA71A3D8FFA433233BC989E58FDEACE75BB2490E684943367F63

iv =ADB572B68058554A78DE6CC232BF86A7

Linux openSSL (version 1.1.0l Sep 2019)

openssl enc -aes-256-cbc -S 9AFC4D99FF13226E -in Data.rtz -out Data.rtz.enc -pass pass:password  -P

Output:

salt=9AFC4D99FF13226E

key=751B52077803FC64530C3D2641B9F09243474CA405967300729C7286C0441BD4

iv =654C3CD6B14ADBE53044C041022F36C8

- Latest version openSSL ( version 1.1.1g  21 Apr 2020)

openssl enc -aes-256-cbc -S 9AFC4D99FF13226E -in Data.rtz -out Data.rtz.enc -pass pass:password  -P

Output:

\*\*\* WARNING : deprecated key derivation used.

Using -iter or -pbkdf2 would be better.

salt=9AFC4D99FF13226E

key=751B52077803FC64530C3D2641B9F09243474CA405967300729C7286C0441BD4

iv =654C3CD6B14ADBE53044C041022F36C8

This shows that the testTool and the newer versions differ in how they derive key and iv from the provided passphrase. In the newest version there is also a warning about using the default pass option as it is now recommended to use the pbkdf2 algorithm to derive keys from a passphrase.

Pbkdf2 is available in dotnet as well, and so far seems to result in identical output compared to openSSL.

However, this then need to define the exact same number of iterations in hashing to work. (openSSL defaults to 10000) Also, pbkdf is not available in older versions of openSSL

Example 2: Comparison using pbkdf2

-- openSSL 1.1.1 version:

openssl enc -aes-256-cbc -S 9AFC4D99FF13226E -in Data.rtz -out Data.rtz.enc -pass file:key.hex -pbkdf2 -iter 10000  -P

salt=9AFC4D99FF13226E

key=236C20A8DFF3019C8304517C103850DB8C5FE76348281F91E18B5DA4688D1498

iv =D2167D37FD54888C0CE9742CD9CF7AD7

-- dotnet implementation output: (10000 iterations)

Salt:9AFC4D99FF13226E

Key:236C20A8DFF3019C8304517C103850DB8C5FE76348281F91E18B5DA4688D1498

Iv: D2167D37FD54888C0CE9742CD9CF7AD7

Summary: Identical salt, key and iv in both implementations.

Suggested change:

Either change so that encryption is done by providing key and init vector. This will then work for older openSSL implementations, but require transfer of the init vector. The S-100 way of handling the init vector is possible but is not standard, and require special handling when encrypting/decrypting.

Or, change to use the pbkdf2 algorithm, but then one also need to pass along the number of iterations to be used. Or to agree upon the number of iterations up front.

So I wonder if we should continue to use a passphrase to create key and iv, assuming the default iteration works?

Or, as an alternative would be to provide a key + iv as the input when encrypting/decrypting:

Encrypt

openssl enc -aes-256-cbc  -in Data.rtz -out Data.rtz.enc -K {key} -iv {init vector} -S {salt}

Decrypt

openssl enc -aes-256-cbc -d -in Data.rtz.enc -out Data.rtz.decrypted -K {key} -iv {init vector}

If doing it this way, we will need to transfer the iv along the data.

Another alternative would be to define that key and iv is created using pbkdf2, but then we either need to agree upon the number of iterations, or pass that along.

Suggested change:

When exchanging data per email the data should be changed into Base64 format. This can be done by adding the flag -base64 to the enc command. To skip line breaks also add the -A flag.

Example command:

openssl enc -aes-256-cbc -base64 -A -in data.rtz -out data.rtz.enc64A -pass file:secretRandomKey.bin -salt

## What to sign

Kept for historic reasons, is now implemented in SECOM.

~~The signature in the message is currently optional. The task for what to sign say that signatures should only be done when necessary, so a definition of necessary is needed.~~

~~Is lack of TLS an example of when signatures are needed?~~

~~Or is it also when there is a vendor Api involved in the transaction? (Non-direct end-to-end communication)~~

~~In either way optional cases lead to confusion so it would be better to have the signature mandatory. The alternative is to remove it completely and replaced it with an unsigned hash. Doing so however would reduce the protection end to end when sending unencrypted data so it is not a good alternative.~~

~~The suggestion is therefore to make the signature mandatory.~~

### Message structure for signing

Kept for historic reasons, is now implemented in SECOM.

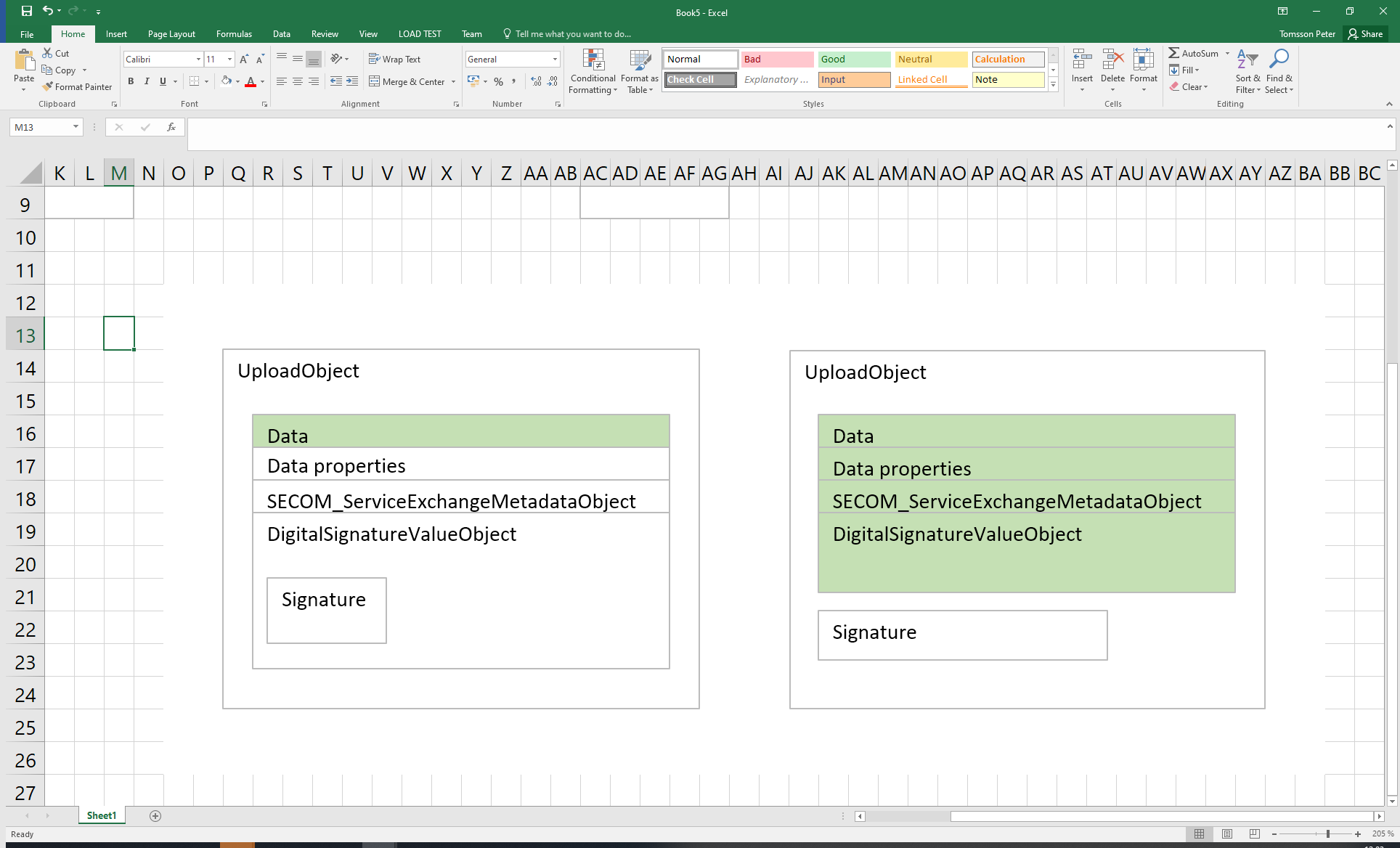
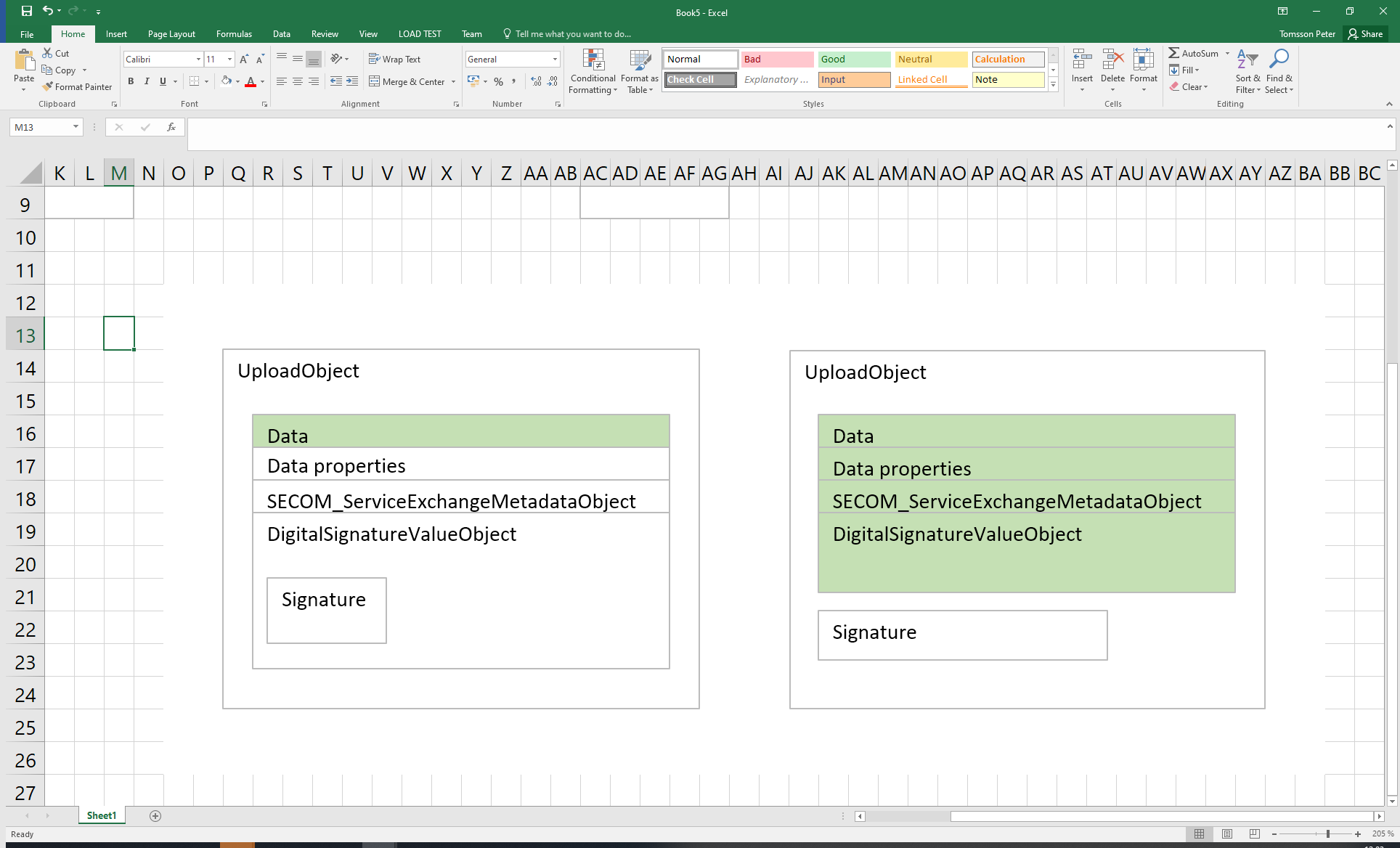
The message structure in the current document is not well structured for signing.

1. Only the data is currently signed:
   1. This makes everything else in the message open for tampering if using an insecure channel, or communication that is not end to end in one TLS tunnel.
   2. This also means that the details of the message and the signature requirements is not part of the signature. In practice this means that one could exchange the complete data and signature in a message and the message would still have a valid signature. This is not secure.
2. The signature is baked into the DigitalSignatureValueObject.
   1. This means that the DigitalSignatureValueObject cannot be signed, since inserting the signature later would affect the signature. Moving the actual signature outside would allow for the DigitalSignatureValueObject to be signed, securing the details within, and the signature, since it is signed by a certificate will remain secure.

The suggested change of message structure can be illustrated below in the following two images.

The boxes marked green indicate what is protected by the signature. However, as mentioned above, it sent on a non-secure channel, this may also be compromised.

Left: Current structure: Right: suggested change



A consequence of this suggested model however is that the orignal data will be signed in it’s Base64 form as part of the complete message. This then goes against the STT-005 comment, as that only consider that the payload is signed.

If a hash for only the payload is needed(As addition to the message signature), this could be added to the SECOM\_ExchangeMetadataObject, and this would then work for both the upload and upload link interface.

The Hash should be calculated on the original file if not encrypted, and on the encrypted file if sending as encrypted to not provide details of the encrypted content.

Based on questions in Test Cases

#### Test data and commands

Testdata used has been a 8kB rtz file. (NewYork-Gothenburg.rtz)

Commands for openSSL are listed in SECOM-TASK-2019-17\_DAtaEncryption.

#### Conclusions and recommendations to SECOM WG

##### The Scope of SECOM

Some reasoning behind the choices made for data protection using signatures and encryption and the relationship with S-100.

“SECOM data protection contains a protection scheme for the information exchange in line with IHO S-100. The data protection scope is between end-users.”

Is the scope for SECOM to only provide encryption as defined in S-100 and thereby limited to the definitions in S-100?

Or is the scope of SECOM as written, to enable secure communication and data protection of, but not limited to, S-100 products?

- If limited by S-100 the scope should be changed and then the purpose of SECOM makes less sense? As S-100 is already defined?

- If not limited by S-100, then SECOM can define how to secure communication unbound by S-100 but preferably not too different to S-100? (Supporting S-100, not limited by)

The difference may be small, but important.

If limited by S-100, we may only use keys as defined in S-100, their lengths, use of Init Vector(IV) etc.

If supporting S-100 we could extend with longer keys, as long as we support the shorter key defined in S-100.

The S-100 use of IV is odd, Mattias (SMA) has written more about this. But if we are to be limited by S-100 this is how we must continue.

An alternative would be to treat the payload as any kind of file and to view SECOM as a way to provide transportation of data, packed in a secure SECOM package.

SECOM is then separated from S-100 specifics regarding keys and IV. And at the same time possible to use for any kind of data, including S-100 products.

Doing so would also decouple S-100 products from SECOM, making it easier to update either standard.

##### Data security:

SECOM transport protection can only be provided between secure TLS connections.

Data protection can range end to end, if using encryption and signing. If this is the intended purpose, then the original data to transfer is what is to be signed.

This original data may then be compressed and/or encrypted as needed.

This setup however will limit the validation possible to do in the Vendor API when sending encrypted data. This is however how it is written in the SECOM scope in the beginning, the data protection scope is between end-users.

### Test Case n

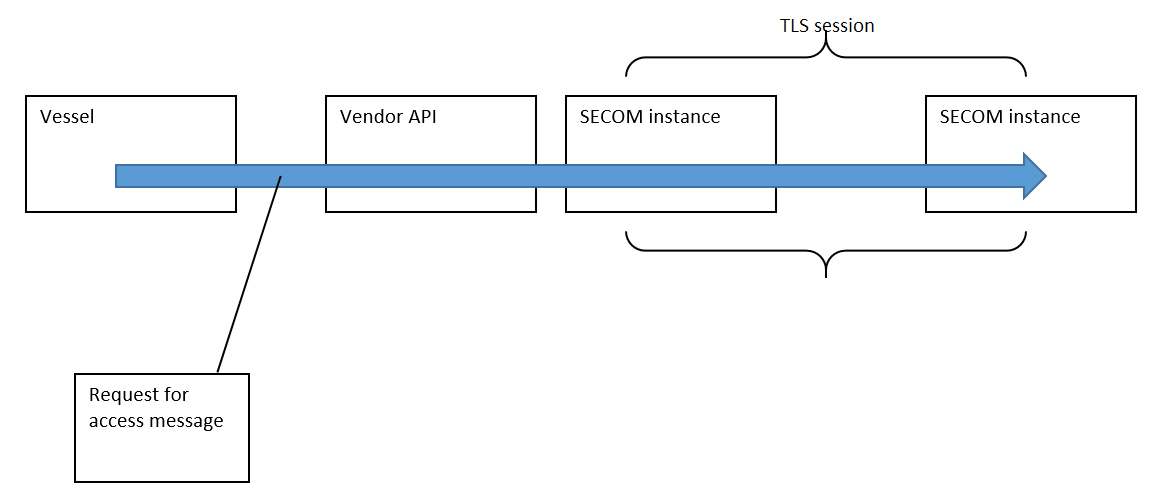
##### Request for Access

When sending information through SECOM, the communication takes place using TLS where the communicating parties exchange certificates and it is thus possible to identify a calling client by its client certificate.

The current way of requesting access does not include the identity of the party requesting access. This means that the identity available is the one available in the client certificate.

This then means that when communication need to take place in a two-step approach, example a vessel behind a vendor API, only the identity of the SECOM instance is able to request access, not the vessel identity.

(Or if a provider is hosting their SECOM instance behind a proxy, then the TLS session will terminate at the proxy, not even reaching the SECOM instance. This is common in load balancing and/or redundant setups)



A way of resolving this would be to include the identity of the vessel requesting the access in the message and signing the message. Preferably also the recipient identity and a timestamp should be included to stop playback attacks. This structure then needs to be applied to all interfaces and messages where authentication is required.

The downside of doing this would be that the request for access message would grow in size. However the new getPublicKey interface can help with that, see Minimizing message size below.

Any request that does involve authorization then should need to be signed by the sending instance to be sure of the origin of the request.

This complicated setup originates from that the data should be protected end to end. The data is protected in the TLS session, but the section between the vessel and its side of the SECOM Instance is outside the TLS session so it can only be protected using a different method like a digital signature.

##### Minimizing message size

In the current signed messages, the included public certificate takes up a lot of space. With the added GetPublicKey interface this could be reduced to only include the thumbprint of the signing certificate in the signed data. The receiver of signed data can then use the getPublicKey interface to fetch the complete public certificate from the sender (or from the PKI if possible) to verify the signature. This can be done once for the required certificate and then stored for future use until the certificate times out or is revoked.

The reduction in size due to this would be significant. A public certificate in Base64 can consist of 1600 characters. The thumbprint of the same is only 40 characters.

Suggestion is to move to use thumbprint for signature and use the getPublicKey interface towards the sender to fetch the public certificate and be able to verify the signature.