Industry Contribution for OmniAir WG Review / Recommendation for Standards / Specification, Testing, and Certification

Contribution Type: [Standard Change]
Specification Affected: [IEEE Std 1609.2-2025 Section 6.3.42 & 6.3.45]
Description Affected: [Public Key Origin type for ECIES Key Derivation Determination]
Applicable OmniAir WG: [CyberSecurity]
WG Review: [Initial 2025-08-04]
WG Recommendation: [Posted on website until Standards Consideration]

Certification Test Spec Effect: [Incorporate into 1609.2]

Formatted for Public Posting: [Roebuck]
Document Identifier: [1609.2:2025 Note 3]

OmniAir Guidance Note GN-1609.2-2025-03:

How to determine Public Key Origin type for ECIES Key Derivation

William Whyte – Qualcomm Inc

2026-04-24 d1 Original, 2026-08-13 d2 Formatting

Introduction

This Guidance Note provides guidance as to how OmniAir will interpret the subject standard (IEEE 1609.2-2025) with respect to material in that standard that is unclear or appears to be internally inconsistent.

The publication of this Guidance Note is not a commitment on the part of OmniAir to develop a test that will test the interpretation given in this Guidance Note. However, if such a test is developed, it will be consistent with the material given in this Guidance Note.

This Guidance Note applies to the indicated dated version of the subject standard. If a future version of the standard contains the same unclarity or inconsistency, a separate Guidance Note will be issued to apply to that standard. OmniAir members who will participate in the development of revisions of the subject standard may choose to draw attention to the unclarity or inconsistency identified in this Guidance Note, and to propose contributions to the revised standard to make it clear and consistent with the material given in this Guidance Note.

Technical Content: Issue

In 1609.2 encryption with ECIES, first a public key operation is carried out to obtain a shared secret, and then the shared secret is input to a Key Derivation Function (KDF) to obtain the symmetric keys that are used to encrypt and MAC the plaintext. The KDF basically hashes the shared secret along with other inputs to obtain the output keys. In 1609.2, the other inputs (cases) depend on the source of the encryption public key:

1. If the encryption public key was in a certificate, the input is the hash of the certificate
2. If the encryption public key was in the HeaderInfo of a SignedData within an Ieee1609Dot2Data d (i.e., the encryption key is d.signedData.tbsData.-headerInfo.encryptionKey.public), the input is the hash of the Ieee1609Dot2Data
3. If the encryption public key was obtained from a different source, the input is the hash of the empty string.

In 1609.2-2025, the description of case (2) above in 6.3.42 can be interpreted to mean that it applies to any case where the public key is in a signed SPDU, including when it is in the payload. This would be wrong; case (2) applies only when the public key is in the encryptionKey field in the HeaderInfo. This is explained properly in 5.3.5.1 but, since the ASN.1 specification in clause 6 is for many developers the main entry point to 1609.2, it's important that everything in clause 6 is as clear and specific as possible. This Guidance Note proposes language for 6.3.42 (and also 6.3.43, Annex C.8) to ensure that the source of the encryption key for case 2 is specified consistently and precisely throughout the document.

(For background on the distinction between cases 2 and 3: the idea is that if the encryption key is in the HeaderInfo, the security services will discover the encryption key when carrying out verification and will automatically store the hash of the SPDU, so it is available naturally; while if the key is in the payload, this might not be discovered until after the verification has been carried out and the 1609.2 envelope has been discarded.)

Technical Content: Resolution

IEEE 1609.2-2025 will be interpreted as follows. Paragraphs that are changed from the published version of the standard are highlighted in green.

6.3.42 RecipientInfo

```
RecipientInfo ::= CHOICE {
  pskRecipInfo        PreSharedKeyRecipientInfo,
  symmRecipInfo       SymmRecipientInfo,
  certRecipInfo       PKRecipientInfo,
  signedDataRecipInfo PKRecipientInfo,
  rekRecipInfo        PKRecipientInfo
}

SequenceOfRecipientInfo ::= SEQUENCE OF RecipientInfo
```

This data structure is used to transfer the data encryption key to an individual recipient of an `EncryptedData`. The option `pskRecipInfo` is selected if the `EncryptedData` was encrypted using the static encryption key approach specified in 5.3.4. The other options are selected if the `EncryptedData` was encrypted using the ephemeral encryption key approach specified in 5.3.4. The meanings of the choices are as follows:

— `pskRecipInfo`: The data was encrypted directly using a pre-shared symmetric key.

— `symmRecipInfo`: The data was encrypted with a data encryption key, and the data encryption key was encrypted using a symmetric key.

— `certRecipInfo`: The data was encrypted with a data encryption key, the data encryption key was encrypted using a public key encryption scheme, where the public encryption key was obtained from a certificate. In this case, the parameter P1 to ECIES as defined in 5.3.5 is the hash of the certificate, calculated with the whole-certificate hash

algorithm, determined as described in 6.4.3, applied to the COER-encoded certificate, canonicalized as defined in the definition of `Certificate`.

NOTE 1—If the encryption algorithm is SM2, there is no equivalent of the parameter P1 and so no input to the encryption process that uses the hash of the certificate.

— `signedDataRecipInfo`: The data was encrypted with a data encryption key, the data encryption key was encrypted using a public key encryption scheme, where the public encryption key was obtained as the public response encryption key from the `HeaderInfo` of a `SignedData` within an `Ieee1609Dot2Data` *d* (i.e., the encryption key is *d*.`signedData.tbsData.headerInfo.encryptionKey.public`). In this case, if ECIES is the encryption algorithm, then the parameter P1 to ECIES as defined in 5.3.5 is the SHA-256 hash of the `Ieee1609-Dot2Data` of type `SignedData` containing the response encryption key, canonicalized as defined in the definition of `Ieee1609Dot2Data`.

NOTE 2—If the encryption algorithm is SM2, there is no equivalent of the parameter P1 and so no input to the encryption process that uses the hash of the `Ieee1609Dot2Data`.

NOTE 3—If the public encryption key is not obtained as *d*.`signedData.tbsData.headerInfo.encryption-Key.public`, for example if it is obtained from a field in *d*.`signedData.tbsData.payload`, then the option `rekRecipInfo` is used rather than `signedDataRecipInfo`.

— `rekRecipInfo`: The data was encrypted with a data encryption key, the data encryption key was encrypted using a public key encryption scheme, where the public encryption key was not obtained from a certificate and was not obtained from the `HeaderInfo` of a `SignedData` within an `Ieee1609Dot2Data` (see previous item in this list for more details). In this case, the SDEE specification is expected to specify how the public key is obtained, and if ECIES is the encryption algorithm, then the parameter P1 to ECIES as defined in 5.3.5 is the hash of the empty string.

NOTE 4—If the encryption algorithm is SM2, there is no equivalent of the parameter P1 and so no input to the encryption process that uses the hash of the empty string.

NOTE 5—This option (`rekRecipInfo`) is used if the public encryption key is obtained from a field in *d*.`signedData.-tbsData.payload` for some `Ieee1609Dot2Data` *d*.

See C.8 for guidance on when it may be appropriate to use each of these approaches.

NOTE—The material input to encryption is the bytes of the encryption key with no headers, encapsulation, or length indication. Contrast this to encryption of data, where the data is encapsulated in an `Ieee1609Dot2Data`.

6.3.45 PKRecipientInfo

```
PKRecipientInfo ::= SEQUENCE {
  recipientId HashedId8,
  encKey      EncryptedDataEncryptionKey
}
```

This data structure contains the following fields:

— `recipientId` contains the hash of the container for the encryption public key as specified in the definition of `RecipientInfo`. Specifically, depending on the choice indicated by the containing `RecipientInfo` structure:

  — If the containing `RecipientInfo` structure indicates `certRecipInfo`, this field contains the `HashedId8` of the certificate. The `HashedId8` is calculated with the whole-certificate hash algorithm,

determined as described in 6.4.3, applied to the COER-encoded certificate, canonicalized as defined in the definition of `Certificate`.

— If the containing `RecipientInfo` structure indicates `signedDataRecipInfo`, this field contains the `HashedId8` of the `Ieee1609Dot2Data` *d* that contained the encryption key as *d*.`signedData.tbsData.headerInfo.encryptionKey.public`. The `Ieee1609Dot2Data` *d* is canonicalized per 6.3.4 prior to hashing. The `HashedId8` is calculated with the hash algorithm determined as specified in 5.3.9.5.

— If the containing `RecipientInfo` structure indicates `rekRecipInfo`, this field contains the `HashedId8` of the COER encoding of a `PublicEncryptionKey` structure containing the response encryption key. The `HashedId8` is calculated with the hash algorithm determined as specified in 5.3.9.5.

— `encKey` contains the encrypted data encryption key, where the data encryption key is input to the data encryption key encryption process with no headers, encapsulation, or length indication.

C.8 Source of encryption keys

This standard supports three means for a sending SDEE to obtain encryption public keys to produce an encrypted SPDU:

— From a certificate

— From the encryptionKey field in the `HeaderInfo` of a `SignedData`

— By some other means

In the first two cases, the key identifier in the `RecipientInfo` is calculated by hashing the "container" (the certificate or the signed SPDU); in the third case, the key identifier is calculated by hashing only the public key. The advantage of hashing the "container" is to prevent misbinding attacks. In these attacks, an attacker tricks one victim into encrypting a message that the victim thinks is meant for one party and is in fact sent to another party. For example, say Alice has a public key. Mallory sends this public key to Bob, and Bob encrypts a message, thinking it is for Mallory. Mallory then forwards the encrypted message to Alice; Alice decrypts it and thinks that it is intended for her because it was encrypted with her encryption key.

This attack is thwarted by hashing the container. In the above case, whether Mallory had managed to get Alice's public key issued as the encryption key in a certificate for Mallory, or instead had included it in a signed SPDU, the hashed container would be identified with Mallory. Alice would expect that if a message was intended for her, the hashed container would be her certificate or a signed SPDU that she had previously sent, and so the attempt to persuade her that the encrypted SPDU was encrypted for her would fail because (1) the container hash in the `RecipientInfo` would not match any container hash that Alice had stored; and (2) the container hash that Alice provides as parameter P1 to ECIES, as specified in 5.3.5, would not match the container hash that Bob used when encrypting. Therefore, this standard recommends that SDEE designers who use public key encryption make use of either public keys in certificates or public keys in signed SPDUs and not use raw public keys.

NOTE—The use of raw public keys does not mitigate misbinding threats.

For an SDEE designer choosing between using a public key from a certificate or a public key from a signed SPDU:

— If the public key is in a certificate, there is one long-term decryption key. This makes key management and storage simpler on the device, and it carries the risk that if the decryption key is compromised, all encrypted SPDUs encrypted with that key can be decrypted. In this scenario, the lifetime of the decryption key is essentially the lifetime of the certificate, so if the device is physically compromised in that time, then a significant number of past communications could be revealed.

— If the public key is in a signed SPDU, there may during the course of its lifetime be many decryption keys to be managed by any device that hosts SDEEs that sign SPDUs with encryption keys and receive the encrypted SPDUs for decryption. The device will need to store each decryption key along with the canonicalized hash of the signed

SPDU that contained the corresponding encryption key for at least the length of time in which it expects to receive responses. This creates more key management complexity than is the case for encryption keys in certificates. However, the advantage is that if one decryption key is compromised, only messages encrypted with that key will be compromised. In this scenario, an encryption key may be expected to be used one time only, and the corresponding decryption key can be deleted once the encrypted SPDU has been decrypted. This provides greater protection for past messages in the event of device compromise than is provided by the alternative model of long-lived encryption keys in certificates.

The SDEE designer may select whether encryption keys are contained in certificates or signed SPDUs taking the above considerations into account.

The SDEE designer should also take into consideration that it is expected that if a signed SPDU *d* contains a response encryption key in *d*.`signedData.tbsData.headerInfo.encryptionKey.public`, then that response encryption key is generated by the SDS instance that signed the SPDU and so the decryption key is only available to that SDS. This means that if an SPDU is intended to enable an encrypted response to any entity other than the specific SDEE instance that created the signed SPDU, the preferred design approach is for the response encryption key to be provided in the payload rather than the `HeaderInfo` of the signed SPDU.

Change History

2026-04-24 d1 (William Whyte) – Original

2026-08-13 d2 (Randy Roebuck) - Formatting