加密信息：

encryptedPassData —— 用苹果的证书链加密

ephemeralPublicKey —— 农行自己生成的临时加密公钥

activationData —— 用银联的密钥加密OTP，苹果无法解析

encryptedPassData 的生成过程：

**验证苹果的证书链：**

（见苹果规范6.3节）

certificates 类型为 Array，三个对象： leaf certificate, sub CA certificate, and, finally, the Apple

Root CA - G3 Certificate

**加密支付信息生成encrypted PassData**

（见苹果规范8.3节）

1. 从苹果证书链中的 leaf certificate 提取苹果的公钥

输入：苹果leaf证书

输出：Apple public key

2. 生成临时密钥对 ephemeral key pairs

没有提到输入输出，应该是农行根据算法自行生成？

3. 生成共享密文 shared secret

输入：步骤1中的苹果公钥、步骤2中的临时私钥

输出： shared secre

4. 组装KDF（单步密钥） 输入：

o Counter：固定00000001

o Shared Secret: 步骤3的共享密文

。 Algorithm ID Length：固定0D

。 Algorithm ID：固定69642D6165733235362D47434D

。 Party U Info：固定4170706C65

。 Ephemeral Public Key：步骤2的临时公钥

输出：将上述字段按顺序拼接起来，生成 KDF

5. 生成对称AES key

输入：步骤4中的KDF

输出：AES symmetric key

6. 加密支付敏感数据

输入：AES key, 支付敏感数据（json报文）

输出： Encrypted data + AES GCM MAC

7. 转成base64

输入： activationData、Ephemeral public key、Encrypted data + AES GCM MAC

输出： activationData、ephemeralPublicKey、encryptedData

8. 掌银客户端调用苹果接口 "PKAddPaymentPassRequest"，转成二进制数据传给苹果

疑问：

 P62有个提示，"The issuer’s implementation discards the ephemeral private key." ，为什 么丢弃私钥，私钥不是在步骤3中用到了吗？

 步骤7生成的 encryptedData，是不是就是传给掌银的encryptedPassData？

 encryptedData 的加密原始数据包含哪些？

**概念理解**

 FPAN: funding PAN

 DPAN: device PAN

 FPANID： primaryAccountIdentifier. The FPANID (an opaque identifier of the FPAN).

The issuer’s PNO specifies the format and value, which are only available after the first provisioning of a card.

 DPANID： deviceAccountIdentifier. The DPANID (an opaque identifier of the DPAN). The issuers PNO specifies the format and value.

 FPAN suffix： primaryAccountNumberSuffix. The last four digits of the FPAN.

 DPAN suffix：deviceAccountNumberSuffix. The last four digits of the DPAN.

Using these methods and properties, obtain the DPANID/FPANID and DPAN/FPAN suffix to determine whether to present the Add to Apple Wallet button.

**6.3 Certificates**

**PROVISIONING**

**Validating Certificates**

Using PKAddPaymentPassViewController, the system provides ECC public certificates to the issuer app, which passes them to the issuer host.

1. The issuer host needs to first verify that the certificate chain is rooted in the Apple Certificate

Authority. This is an extremely critical step of the process, and failing to validate the certificates may potentially result in the issuer host encrypting the card details for a party other than Apple.

2. The issuer host needs to validate the chain of trust with the provided leaf certificate, sub CA

certificate, and, finally, the Apple Root CA - G3 Certificate from Apple's Root Certificate Authority.

3. Additionally, the issuer host needs to ensure the OID 1.2.840.113635.100.6.39 is present and marked critical in the leaf certificate.

**Generating Keys**

Subsequently, the issuer host can extract the static public key and generate an ECC ephemeral key pair.

1. The issuer host uses the static public key from Apple and the generated ephemeral private key to derive a shared secret.

2. The issuer host inputs the shared secret into a key derivation function (KDF) to calculate the shared key. For details about generating the shared key, see Section 8.3.

3. The issuer host uses the derived shared key to encrypt the payment details (encryptedPassData), and provide it to the app, along with the ephemeralPublicKey and activationData.

Test vectors and step-by-step instructions are available in Section 8.3.

Important

Don’t perform cryptographic operations to support the transmission of payment data to Apple Wallet on the iOS device. The issuer host needs to perform the cryptographic functions.

**8.3 In-App Provisioning Test Vectors PROVISIONING**

This step-by-step guide details the process for using test vectors. To help the issuer test their issuer

host encryption, as well as their app interface with Apple Wallet, Apple provides test vectors (including a sample set of public certificates, plaintext, ephemeral key pair, shared secret, shared key, and cipher

text), as well as a detailed step-by-step guide for using them.

The following section contains an example with production-like data that the issuer can use to validate their backend implementation of In-App Provisioning cryptography.

Steps 1–6 cover the issuer server-side cryptography, and steps 7–8 refer to passing data to the issuer app with an example of how to correctly initialize the iOS APIs.

After the issuer validates their ability to generate each data point, the issuer has successfully validated their cryptography with the test vectors (steps 1 to 6).

Important

All data is hexadecimal-encoded unless otherwise stated.

**Step 1: Extract the Public Key from the Leaf Certificate**

After the issuer verifies the validity of the certificate chain, the first step consists of extracting the Apple public key from the leaf certificate.

|  |  |
| --- | --- |
| Input | // Apple leaf certificate (sample, PEM format, Base64-encoded):  -----BEGIN CERTIFICATE-----  MIIEEjCCA7igAwIBAgIIEccnFAKsD+UwCgYIKoZIzj0EAwIwgYExOzA5BgNVBAMMMlRlc3QgQXBwbGUgV29ybGR3a WRlIERldmVsb3BlcnMgUmVsYXRpb25zIENBIC0gRUNDMSAwHgYDVQQLDBdDZXJ0aWZpY2F0aW9uIEF1dGhvcm l0eTETMBEGA1UECgwKQXBwbGUgSW5jLjELMAkGA1UEBhMCVVMwHhcNMTkwODA3MjA1NzA5WhcNMjEwNzEz MDI1ODAwWjBtMTYwNAYDVQQDDC1lY2MtY3J5cHRvLXNlcnZpY2VzLWVuY2lwaGVybWVudF9VQzYtSW5NZW1v cnkxETAPBgNVBAsMCEFwcGxlUGF5MRMwEQYDVQQKDApBcHBsZSBJbmMuMQswCQYDVQQGEwJVUzBZMBM GByqGSM49AgEGCCqGSM49AwEHA0IABC4+XM9rmrBL56IvP6zP3nPIfocVU5SjSBVAiolsoYo3TaxmmvO/  YiD8hjdn9K9HUHxbwiH8ShmHTa85tAdOPrijggIrMIICJzAMBgNVHRMBAf8EAjAAMB8GA1UdIwQYMBaAFNbW1Vrl// 3CfDTDQ969aHZcNqm+ME8GCCsGAQUFBwEBBEMwQTA/  BggrBgEFBQcwAYYzaHR0cDovL29jc3AtdWF0LmNvcnAuYXBwbGUuY29tL29jc3AwNC10ZXN0d3dkcmNhZWNjMIIB HQYDVR0gBIIBFDCCARAwggEMBgkqhkiG92NkBQEwgf4wgcMGCCsGAQUFBwICMIG2DIGzUmVsaWFuY2Ugb24g dGhpcyBjZXJ0aWZpY2F0ZSBieSBhbnkgcGFydHkgYXNzdW1lcyBhY2NlcHRhbmNlIG9mIHRoZSB0aGVuIGFwcGxpY 2FibGUgc3RhbmRhcmQgdGVybXMgYW5kIGNvbmRpdGlvbnMgb2YgdXNlLCBjZXJ0aWZpY2F0ZSBwb2xpY3kgYW5 kIGNlcnRpZmljYXRpb24gcHJhY3RpY2Ugc3RhdGVtZW50cy4wNgYIKwYBBQUHAgEWKmh0dHA6Ly93d3cuYXBwbG UuY29tL2NlcnRpZmljYXRlYXV0aG9yaXR5LzBBBgNVHR8EOjA4MDagNKAyhjBodHRwOi8vY3JsLXVhdC5jb3JwLmF wcGxlLmNvbS9hcHBsZXd3ZHJjYWVjYy5jcmwwHQYDVR0OBBYEFK0uo8t+NMLt7kNoTicRH8xJMznQMA4GA1UdD wEB/  wQEAwIDKDASBgkqhkiG92NkBicBAf8EAgUAMAoGCCqGSM49BAMCA0gAMEUCIQCKEXnIsY2PZqMF2xHKehKgp/ ZywZ/9/TZ+AnpOA6mI/AIgTI94NSaIn7DLd47QTK760WILDOr0EdOHiExJMZwYp7c=  -----END CERTIFICATE----- |
| Operation | Parse the leaf certificate and extract the Apple public key. |
| Output | // Apple public key in uncompressed format (65 bytes):  042E3E5CCF6B9AB04BE7A22F3FACCFDE73C87E87155394A34815408A896CA18A374DAC669AF3BF6220FC863 767F4AF47507C5BC221FC4A19874DAF39B4074E3EB8 |

**Step 2: Ephemeral Key Pairs**

On the issuer host, generate the ephemeral key pairs.

|  |  |
| --- | --- |
| Output | // Ephemeral public key in uncompressed format (65 bytes):  0499A6F42E83EA4F150A78780FFB562C9CDB9B7507BC5D28CBFBF8CC3EF0AF68B36E60CB10DB69127830F7F 899492017089E3B73C83FCF0EBDF2C06B613C3F88B7  // Ephemeral private key (32 bytes):  7EEE47DEE108A08EDD2BCD2BB762A543CA23EA96C9AF09AD54BEB9FA3CE1A026 |

The ephemeral private key is a hexadecimal-encoded integer value. The associated curve is NIST P-256.

**Step 3: Elliptic Curve Diffie–Hellman Key Exchange**

The Elliptic Curve Diffie–Hellman algorithm generates the shared secret as input keys use the Apple public key from the leaf certificate and the ephemeral private key generates on the issuer host.

|  |  |
| --- | --- |
| Input | // Apple public key in uncompressed format (from step 1):  042E3E5CCF6B9AB04BE7A22F3FACCFDE73C87E87155394A34815408A896CA18A374DAC669AF3BF6220FC863 767F4AF47507C5BC221FC4A19874DAF39B4074E3EB8  // Ephemeral private key (from step 2):  7EEE47DEE108A08EDD2BCD2BB762A543CA23EA96C9AF09AD54BEB9FA3CE1A026 |

Operation Use the Apple public key and the ephemeral private key as parameters in the shared

secret function.

|  |  |
| --- | --- |
| Output | // Shared secret (32 bytes):  A88B995FECBDF756515ED42BA53A6CCCA4F5936F69CF4D15352C94C592B347B1 |

Important

The issuer’s implementation discards the ephemeral private key.

**Step 4: NIST Single-Step Key Derivation Function (KDF)**

The NIST single-step KDF parameterizes with the SHA-256 hash function to derive a 256-bit key. The following are the inputs to the KDF:

|  |  |
| --- | --- |
| Input | // Counter (4 bytes):  00000001  // Shared Secret (from step 3, 32 bytes):  A88B995FECBDF756515ED42BA53A6CCCA4F5936F69CF4D15352C94C592B347B1  // Other Info (84 bytes):  // Algorithm ID Length (1 byte):  0D  // Algorithm ID ("id-aes256-GCM", 13 bytes):  69642D6165733235362D47434D  // Party U Info ("Apple", 5 bytes):  4170706C65  // Ephemeral Public Key (from step 2, 65 bytes):  0499A6F42E83EA4F150A78780FFB562C9CDB9B7507BC5D28CBFBF8CC3EF0AF68B36E60CB10DB69127830 F7F899492017089E3B73C83FCF0EBDF2C06B613C3F88B7 |

Operation Generate the other information as NIST SP 800-56A, Section 5.8.1 specifies.

Concatenate the values in the order above.

|  |  |
| --- | --- |
| Output | // KDF Input (120 bytes):  00000001A88B995FECBDF756515ED42BA53A6CCCA4F5936F69CF4D15352C94C592B347B10D69642D61657332 35362D47434D4170706C650499A6F42E83EA4F150A78780FFB562C9CDB9B7507BC5D28CBFBF8CC3EF0AF68B 36E60CB10DB69127830F7F899492017089E3B73C83FCF0EBDF2C06B613C3F88B7 |

Note

Counter, Algorithm ID Length, Algorithm ID, and Party U Info are all static. Shared Secret and Ephemeral Public Key are dynamic and different for each provisioning attempt.

**Step 5: Symmetric AES Key**

From the KDF input, the issuer can generate the AES key, which step 6 uses for encryption.

|  |  |
| --- | --- |
| Input | // KDF input (from step 4):  00000001A88B995FECBDF756515ED42BA53A6CCCA4F5936F69CF4D15352C94C592B347B10D69642D61657332 35362D47434D4170706C650499A6F42E83EA4F150A78780FFB562C9CDB9B7507BC5D28CBFBF8CC3EF0AF68B 36E60CB10DB69127830F7F899492017089E3B73C83FCF0EBDF2C06B613C3F88B7 |

Operation As NIST SP 800-56A, Section 5.8.1 specifies, apply the SHA-256 hash function to the

KDF input to generate the shared AES key.

|  |  |
| --- | --- |
| Output | // AES symmetric key (32 bytes):  083080D3D0C521C02CD3AE2134363D09EA50DFF914677FAB9E22F18F9C28A3B9 |

Important

The issuer needs to apply the hash function to the binary data of the KDF input, not its hexadecimal representation.

**Step 6: AES GCM Encryption**

The AES symmetric key encrypts the JSON payload with the AES cipher in GCM mode with null byte IV. The resulting output includes the AES GCM message authentication code (MAC) after the encrypted

data.

|  |  |
| --- | --- |
| Input | // AES key (from step 5):  083080D3D0C521C02CD3AE2134363D09EA50DFF914677FAB9E22F18F9C28A3B9  // JSON payload (67 bytes), UTF-8 encoded:  {"Parameter1":"Value1","Parameter2":"Value2","Parameter3":"Value3"} |

Operation AES-256 encryption in GCM mode with 12 null byte IV. The AES MAC GCM appends to

the end of the encrypted data.

|  |  |
| --- | --- |
| Output | // Encrypted data + AES GCM MAC (67 + 16 = 83 bytes):  E3EF6BA2FFA05B6985FE129E3CB6845C4EA1E94AE98D31A538A4E24906FB720D764D640894CD9DE7CEC0011 4396651A1CCAEDCF480C57A959E925C04492B9CF85FC711FAB3CBED10DC2BA99A2BB063CEFF8DE1 |

Important

The length of the output data needs to match the length of the input data plus the length of the AES GCM MAC. The issuer needs to ensure the cryptography routine outputs the MAC AES GCM after the encrypted data.

**Step 7: Transferring Data to the Issuer App**

After generating the encrypted data in step 6, the issuer host generates the activation data (the cryptographic OTP required to activate the card by the PNO or service provider).

After generating the activation data, the issuer needs to encode the activation data, the ephemeral

public key, and the encrypted data as Base64 strings and send them as a JSON response to the app.

For more information about the input, see PKAddPaymentPassRequest.

|  |  |
| --- | --- |
| Input | // Activation data:  // network-specific  // Ephemeral public key (from step 2, 65 bytes):  0499A6F42E83EA4F150A78780FFB562C9CDB9B7507BC5D28CBFBF8CC3EF0AF68B36E60CB10DB69127830F7F8 99492017089E3B73C83FCF0EBDF2C06B613C3F88B7  // Encrypted data + AES GCM MAC (from step 6) (67 + 16 = 83 bytes):  E3EF6BA2FFA05B6985FE129E3CB6845C4EA1E94AE98D31A538A4E24906FB720D764D640894CD9DE7CEC00114 396651A1CCAEDCF480C57A959E925C04492B9CF85FC711FAB3CBED10DC2BA99A2BB063CEFF8DE1 |

Operation Encode the activation data, the ephemeral public key, and the encrypted data with

GCM MAC as a Base64 string and create a JSON response using the resultant strings.

|  |  |
| --- | --- |
| Output | // JSON payload sent back to client iOS app (Base64-encoded):  {  "activationData":"bmV0d29yayBzcGVjaWZpYw==",  "ephemeralPublicKey":"BJmm9C6D6k8VCnh4D/tWLJzbm3UHvF0oy/v4zD7wr2izbmDLENtpEngw9/ iZSSAXCJ47c8g/zw698sBrYTw/iLc=",  “encryptedData”:”4+9rov+gW2mF/  hKePLaEXE6h6UrpjTGlOKTiSQb7cg12TWQIlM2d587AARQ5ZlGhzK7c9IDFepWeklwESSuc+F/ HEfqzy+0Q3CupmiuwY87/jeE="  } |

**Step 8: ‘PKAddPaymentPassRequest’**

After the app receives the JSON response from step 7, the issuer can use init(base64Encoded:options:) to initialize the relevant field of that class in a binary format. The API accepts most common Base64

formats.

The example below details how to initialize the field in binary by starting from its Base64 representation:

|  |  |
| --- | --- |
| Example | // Base64 representation  ephemeralPublicKeyBase64 = “BJmm9C6D6k8VCnh4D/tWLJzbm3UHvF0oy/v4zD7wr2izbmDLENtpEngw9/ iZSSAXCJ47c8g/zw698sBrYTw/iLc="  ephemeralPublicKey = Data(base64Encoded: ephemeralPublicKeyBase64, options: []) |

After the system initializes the data, PKAddPaymentPassRequest contains the card data necessary to add a card to Apple Wallet.

Refer to Section 9 for additional information about debugging the iOS API initialization.