User Guide

AWS Payment Cryptography

Copyright © 2026 Amazon Web Services, Inc. and/or its affiliates. All rights reserved.

AWS Payment Cryptography: User Guide

Copyright © 2026 Amazon Web Services, Inc. and/or its affiliates. All rights reserved.

Amazon's trademarks and trade dress may not be used in connection with any product or service that is not Amazon's, in any manner that is likely to cause confusion among customers, or in any manner that disparages or discredits Amazon. All other trademarks not owned by Amazon are the property of their respective owners, who may or may not be affiliated with, connected to, or sponsored by Amazon.

Table of Contents

What is AWS Payment Cryptography? 1
Concepts 2
Industry terminology 4
Common key types 4
Other terms 7
Related services 12
For more information 13
Endpoints 13
Control plane endpoints 13
Data plane endpoints 16
Getting started 19
Prerequisites 19
Step 1: Create a key 20
Step 2: Generate a CVV2 value using the key 21
Step 3: Verify the value generated in step 2 21
Step 4: Perform a negative test 22
Step 5: (Optional) Clean up 22
Managing keys 24
Creating keys 24
Creating a 3KEY TDES base derivation key 25
Creating a 2KEY TDES key for CVV/CVV2 27
Creating an HMAC key 28
Creating an AES-256 key 29
Creating a PIN Encryption Key (PEK) 30
Creating an asymmetric (RSA) key 31
Creating a PIN Verification Value (PVV) Key 32
Creating an asymmetric ECC key 33
Listing keys 34
Enabling and disabling keys 35
Start key usage 35
Stop key usage 37
Replicating keys 39
Benefits of Multi-Region key replication 39
How Multi-Region key replication works 39

- Limitations and considerations 39
- Enabling Multi-Region key replication 40
- Disabling Multi-Region key replication 42
- Security considerations 43
- Best practices 43
- Pricing 44
- Deleting keys 44
- About the waiting period 45
- Importing and exporting keys 49
- Import keys 51
- Export keys 76
- Advanced Topics 98
- Using aliases 106
- About aliases 107
- Using aliases in your applications 110
- Related APIs 111
- Get keys 111
- Get the public key/certificate associated with a key pair 113
- Tagging keys 114
- About tags in AWS Payment Cryptography 114
- Viewing key tags in the console 116
- Managing key tags with API operations 116
- Controlling access to tags 118
- Using tags to control access to keys 123
- Understanding key attributes 126
- Symmetric Keys 126
- Asymmetric Keys 128

Data operations 130
- Encrypt, Decrypt and Re-encrypt data 130
- Encrypt data 131
- Decrypt data 137
- Generate and verify card data 141
- Generate card data 142
- Verify card data 143
- Generate, translate and verify PIN data 145
- Translate PIN data 146

- Generate PIN data 148
- Verify PIN data 152
- Verify auth request (ARQC) cryptogram 156
  - Building transaction data 157
  - Transaction data padding 157
  - Examples 159
- Generate and verify MAC 160
  - Generate MAC 162
  - Verify MAC 165
- Key types for specific data operations 167
  - GenerateCardData 168
  - VerifyCardData 169
  - GeneratePinData (for VISA/ABA schemes) 170
  - GeneratePinData (for IBM3624) 171
  - VerifyPinData (for VISA/ABA schemes) 172
  - VerifyPinData (for IBM3624) 173
  - Decrypt Data 174
  - Encrypt Data 175
  - Translate Pin Data 176
  - Generate/Verify MAC 177
  - GenerateMacEmvPinChange 178
  - VerifyAuthRequestCryptogram 180
  - Import/Export Key 180
  - Unused key types 181

Common use cases 182
- Issuers and issuer processors 182
  - General Functions 182
  - Network specific functions 202
- Acquiring and payment facilitators 227
  - Using Dynamic Keys 228

Region specific features 231
- AS2805 231
  - Initial Key(KEK) exchange 233
  - Validation of KEK 234
  - Creation and transmission of working keys 237
  - Exporting working keys 239

- Pin Translation ... 240
- Mac Generation and Validation ... 241

Security ... 242

- Data protection ... 242
  - Protecting key material ... 244
  - Data encryption ... 244
  - Encryption at rest ... 244
  - Encryption in transit ... 245
  - Internetwork traffic privacy ... 245
- Resilience ... 246
  - Regional isolation ... 246
  - Multi-tenant design ... 246
- Infrastructure security ... 247
  - Isolation of physical hosts ... 247
- Use Amazon VPC and AWS PrivateLink ... 248
  - Considerations for AWS Payment Cryptography VPC endpoints ... 249
  - Creating a VPC endpoint for AWS Payment Cryptography ... 249
  - Connecting to a VPC endpoint ... 250
  - Controlling access to a VPC endpoint ... 251
  - Using a VPC endpoint in a policy statement ... 254
  - Logging your VPC endpoint ... 258
- Hybrid post-quantum TLS ... 260
  - About post-quantum TLS ... 262
  - About PQC ... 262
  - How to use it ... 262
- Security best practices ... 266

Compliance validation ... 268

- Compliance of the service ... 268
- PIN Compliance ... 269
  - Common Topics ... 269
  - Assessment Scope ... 271
  - Transaction Processing Operations ... 273
- P2PE Compliance ... 279

Identity and access management ... 280

- Audience ... 280
- Authenticating with identities ... 280

AWS account root user ... 281
IAM users and groups ... 281
IAM roles ... 281
Managing access using policies ... 281
Identity-based policies ... 282
Resource-based policies ... 282
Access control lists (ACLs) ... 282
Other policy types ... 283
Multiple policy types ... 283
How AWS Payment Cryptography works with IAM ... 283
AWS Payment Cryptography Identity-based policies ... 284
Authorization based on AWS Payment Cryptography tags ... 286
Identity-based policy examples ... 286
Policy best practices ... 287
Using the console ... 288
Allow users to view their own permissions ... 288
Ability to access all aspects of AWS Payment Cryptography ... 289
Ability to call APIs using specified keys ... 290
Ability to specifically deny a resource ... 291
Troubleshooting ... 292
Monitoring ... 293
CloudTrail logs ... 293
AWS Payment Cryptography information in CloudTrail ... 294
Control plane events in CloudTrail ... 295
Data events in CloudTrail ... 295
Understanding AWS Payment Cryptography Control Plane log file entries ... 296
Understanding AWS Payment Cryptography Data plane log file entries ... 300
Cryptographic details ... 302
Design goals ... 303
Foundations ... 304
Cryptographic primitives ... 304
Entropy and random number generation ... 305
Symmetric key operations ... 305
Asymmetric key operations ... 305
Key storage ... 306
Key import using symmetric keys ... 306

- Key import using asymmetric keys ... 306
- Key export ... 306
- Derived Unique Key Per Transaction (DUKPT) protocol ... 307
- Key hierarchy ... 307
- Internal operations ... 310
  - HSM protection ... 310
  - General key management ... 313
  - Management of customer keys ... 316
  - Communication security ... 319
  - Logging and monitoring ... 319
- Customer operations ... 320
  - Generating keys ... 320
  - Importing keys ... 321
  - Exporting keys ... 321
  - Deleting keys ... 322
  - Rotating keys ... 322

Quotas ... 323

Document history ... 325

What is AWS Payment Cryptography?

AWS Payment Cryptography is a managed AWS service that provides access to cryptographic functions and key management used in payment processing in accordance with payment card industry (PCI) standards without the need for you to procure dedicated payment HSM instances. AWS Payment Cryptography provides customers performing payment functions such as acquirers, payment facilitators, networks, switches, processors, and banks with the ability to move their payment cryptographic operations closer to applications in the cloud and minimize dependencies on auxiliary data centers or colocation facilities containing dedicated payment HSMs.

The service is designed to meet applicable industry rules including PCI PIN, PCI P2PE, and PCI DSS, and the service leverages hardware that it is PCI PTS HSM V3 and FIPS 140-2 Level 3 certified. It is designed to support low latency and high levels of up-time and resilience. AWS Payment Cryptography is fully elastic and eliminates many of the operational requirements of on premises HSMs, such as the need to provision hardware, securely manage key material, and to maintain emergency backups in secure facilities. AWS Payment Cryptography also provides you with the option to share keys with your partners electronically, eliminating the need to share paper clear text components.

You can use the AWS Payment Cryptography Control Plane API to create and manage keys.

You can use the AWS Payment Cryptography Data Plane API to use encryption keys for payment-related transaction processing and associated cryptographic operations.

AWS Payment Cryptography provides important features that you can use to manage your keys:

- Create and manage symmetric and asymmetric AWS Payment Cryptography keys, including TDES, AES, and RSA keys and specify their intended purpose such as for CVV generation or DUKPT key derivation.
- Automatically store your AWS Payment Cryptography keys securely, protected by hardware security modules (HSMs) while enforcing key separation between use cases.
- Create, delete, list, and update aliases, which are "friendly names" that can be used to access or control access to your AWS Payment Cryptography keys.
- Tag your AWS Payment Cryptography keys for identification, grouping, automation, access control, and cost tracking.

- Import and export symmetric keys between AWS Payment Cryptography and your HSM (or 3rd parties) using Key Encryption Keys (KEK) following TR-31(Interoperable Secure Key Exchange Key Block Specification).
- Import and export symmetric Key Encryption Keys (KEK) between AWS Payment Cryptography and other systems using asymmetric key pairs following by using electronic means such as TR-34 (Method For Distribution Of Symmetric Keys Using Asymmetric Techniques).

You can use your AWS Payment Cryptography keys in cryptographic operations, such as:

- Encrypt, decrypt, and re-encrypt data with symmetric or asymmetric AWS Payment Cryptography keys.
- Securely translate sensitive data (such as cardholder pins) between encryption keys without exposing the clear text in accordance with PCI PIN rules.
- Generate or validate cardholder data such as CVV, CVV2 or ARQC.
- Generate and validate cardholder pins.
- Generate or validate MAC signatures.

Concepts

Learn the basic terms and concepts used in AWS Payment Cryptography and how you can use them to help you protect your data.

Alias

A user-friendly name that is associated with an AWS Payment Cryptography key. The alias can be used interchangeably with key ARN in many of the AWS Payment Cryptography API operations. Aliases allow keys to be rotated or otherwise changed without impacting your application code. The alias name is a string of up to 256 characters. It uniquely identifies an associated AWS Payment Cryptography key within an account and region. In AWS Payment Cryptography, alias names always begin with `alias/`.

The format of an alias name is as follows:

```
alias/<alias-name>
```

For example:

```
alias/sampleAlias2
```

Key ARN

The key ARN is the Amazon Resource Name (ARN) of a key entry in AWS Payment Cryptography. It is a unique, fully qualified identifier for the AWS Payment Cryptography key. A key ARN includes an AWS account, region, and a randomly generated ID. The ARN is not related or derived from the key material. As they are automatically assigned during create or import operations, these values are not idempotent. Importing the same key multiple times will result in multiple key ARNs with their own lifecycle.

The format of a key ARN is as follows:

```
arn:<partition>:payment-cryptography:<region>:<account-id>:alias/<alias-name>
```

The following is a sample key ARN:

```
arn:aws:payment-cryptography:us-east-2:111122223333:key/kwapwa6qaifllw2h
```

Key Identifier

A Key Identifier is a reference to a key and one (or more) of them are typical inputs to AWS Payment Cryptography operations. Valid key identifiers could be either a Key Arn a Key Alias.

AWS Payment Cryptography keys

AWS Payment Cryptography keys (keys) are used for all cryptographic functions. Keys are either generated directly by you using the create key command or added to the system by you calling key import. The origin of a key can be determined by reviewing the attribute KeyOrigin. AWS Payment Cryptography also supports derived or intermediate keys used during cryptographic operations such as those used by DUKPT.

These keys have both immutable and mutable attributes defined at creation. Attributes, such as algorithm, length, and usage are defined at creation and cannot be changed. Others, such as effective date or expiration date, can be modified. See the AWS Payment Cryptography API Reference for a complete list of AWS Payment Cryptography Key attributes.

AWS Payment Cryptography keys have key types, principally defined by ANSI X9 TR 31, that restrict their use to their intended purpose as specified in PCI PIN v3.1 Requirement 19.

Attributes are bound to keys using key blocks when stored, shared with other accounts, or exported as specified in PCI PIN v3.1 Requirement 18-3.

Keys are identified in the AWS Payment Cryptography platform using a unique value known as a key Amazon Resource Name (ARN).

Note

Key ARN is generated when a key is initially created or imported into the AWS Payment Cryptography service. Thus, if adding the same key material multiple times using the import key functionality, the same key material will be located under multiple key ARNS but each with a different key lifecycle.

Industry terminology

Topics

- Common key types
- Other terms

Common key types

AWS Payment Cryptography key

An AWS Payment Cryptography Key exists in a single AWS Region. It consists of key metadata and material stored in the AWS Payment Cryptography Service. A Key may be imported from an external source as a TR-31 key block or generated by the AWS Payment Cryptography Service.

AWK

An acquirer working key (AWK) is a key typically used to exchange data between an acquirer/ acquirer processor and a network (such as Visa or Mastercard). Historically AWK leverages 3DES for encryption and would be represented as TR31_P0_PIN_ENCRYPTION_KEY.

BDK

A base derivation key (BDK) is a working key used to derive subsequent keys and is commonly used as part of PCI PIN and PCI P2PE DUKPT process. It is denoted as *TR31_B0_BASE_DERIVATION_KEY*.

CMK

A card master key (CMK) is one or more card specific key(s) typically derived from a Issuer Master Key, *PAN* and *PSN* and are typically *3DES* keys. These keys are stored on the EMV Chip during personalization. Examples of CMKs include AC, SMI and SMC keys.

CMK-AC

An application cryptogram (AC) key is used as part of EMV transactions to generate the transaction cryptogram and is a type of card master key.

CMK-SMI

An secure messaging integrity (SMI) key is used as part of EMV to verify the integrity of payloads sent to the card using MAC such as pin update scripts. It is a type of card master key.

CMK-SMC

An secure messaging confidentiality (SMC) key is used as part of EMV to encrypt data sent to the card such as pin updates. It is a type of card master key.

CVK

A card verification key (CVK) is a key used for generating CVV, CVV2 and similar values using a defined algorithm as well as validating an input. It is denoted as a *TR31_C0_CARD_VERIFICATION_KEY*.

IMK

An issuer master key (IMK) is a master key used as part of EMV chip card personalization. Typically there will be 3 IMKs - one each for AC (cryptogram), SMI (script master key for integrity/signature), and SMC (script master key for confidentiality/encryption) keys.

IK

An initial key (IK) is the first key used in the DUKPT process and derives from the Base Derivation Key (BDK). No transactions are processed on this key, but it is used to derive future keys that will be used for transactions. The derivation method for creating an IK was defined in X9.24-1:2017. When an TDES BDK is used, X9.24-1:2009 is the applicable standard and IK is replaced by Initial Pin Encryption Key (IPEK).

IPEK

An initial PIN encryption key (IPEK) is the initial key used in the DUKPT process and derives from the Base Derivation Key (BDK). No transactions are processed on this key, but it is used to derive future keys that will be used for transactions. IPEK is a misnomer as this key can also be used to derive data encryption and mac keys. The derivation method for creating an IPEK was defined in X9.24-1:2009. When an AES BDK is used, X9.24-1:2017 is the applicable standard and IPEK is replaced by Initial Key (IK).

IWK

An issuer working key (IWK) is a key typically used to exchange data between an issuer/issuer processor and a network (such as Visa or Mastercard). Historically IWK leverages 3DES for encryption and is represented as *TR31_P0_PIN_ENCRYPTION_KEY*.

KBPK

A key block encryption key(KBPK) is a type of symmetric key used to protect key blocks and thus wrap/encrypt other keys. A KBPK is similar to a KEK but a KEK directly protects the key material whereas in TR-31 and similar schemes, the KBPK only indirectly protects the working key. When using TR-31, *TR31_K1_KEY_BLOCK_PROTECTION_KEY* is the correct key type, although TR31_K0_KEY_ENCRYPTION_KEY is supported interchangeably for historical purposes.

KEK

A key encryption key (KEK) is a key used to encrypt other keys either for transmission or storage. Keys meant for protecting other keys typically have a KeyUsage of *TR31_K0_KEY_ENCRYPTION_KEY* according to the TR-31 standard.

PEK

A PIN encryption key (PEK) is a type of working key used for encrypting PINs either for storage or transmission between two parties. IWK and AWK are two examples of specific uses of pin encryption keys. These keys are represented as *TR31_P0_PIN_ENCRYPTION_KEY*.

PGK

PGK (Pin Generation Key) is another name for a Pin Verification Key. It's not actually used to generate pins (which by default are cryptographically random numbers) but instead are used to generate verification values such as PVV.

PRK

The Primary Region key is the authoritative replication source for a given Payment Cryptography key for which replication has been enabled. PRK is a reference to a source Payment Cryptography key role in a Multi-Region key replication configuration. When replication is enabled on an Payment Cryptography key, it is referred to as the PRK for that specific keys replication configuration.

PVK

A PIN verification key (PVK) is a type of working key used for generating PIN verification values such as PVV. The two most common kinds are *TR31_V1_IBM3624_PIN_VERIFICATION_KEY* used for generating IBM3624 offset values and *TR31_V2_VISA_PIN_VERIFICATION_KEY* used for Visa/ABA verification values. This can also be known as a Pin Generation Key.

RRK

Replica Region keys are the replicated key material and metadata copied securely from the PRK to a configured replica AWS Region. An RRK is a read only replica of an Payment Cryptography key. RRK is a reference the role a specific key plays in a Multi-Region key replication configuration. Any key metadata changes, including replication settings must be applied to the PRK.

Other terms

ARQC

Authorization Request Cryptogram (ARQC) is a cryptogram generated at transaction time by an EMV standard chip card (or equivalent contactless implementation). Typically, an ARQC is generated by a chip card and forwarded to an issuer or their agent to verify at transaction time.

CVV

A card verification value is a static secret value that was traditionally embedded on a magnetic stripe and used to validate the authenticity of a transaction. The algorithm is also used for other purposes such as iCVV, CAVV, CVV2. It may not be embedded in this way for other use cases.

CVV2

A card verification value 2 is a static secret value that was traditionally printed on the front (or back) of a payment card and is used to verify authenticity for card not present payments (such as on the phone or online). It uses the same algorithm as CVV but the service code is set to 000.

iCVV

iCVV is a CVV2-like value but embedded with the track2 equivalent data on a EMV(Chip) card. This value is calculated using a service code of 999 and is different than the CVV1/CVV2 to prevent stolen information from being used to create new payment credentials of a different type. For instance, if chip transaction data was obtained, it is not possible to use this data to generate a magnetic stripe(CVV1) or for online purchases (CVV2).

It uses a ??? key

DUKPT

Derived Unique Key Per Transaction (DUKPT) is a key management standard typically used to define the use of one-time use encryption keys on physical POS/POI. Historically DUKPT leverages 3DES for encryption. The industry standard for DUKPT is defined in ANSI X9.24-3-2017.

ECC

ECC (Elliptic Curve Cryptography) is a public key cryptography system that uses the mathematics of elliptic curves to create encryption keys. ECC provides the same security level as traditional methods like RSA but with much shorter key lengths, providing equivalent security in a more efficient manner. This is especially relevant for use cases where RSA is not a practical solution (RSA key length > 4096 bits). AWS Payment Cryptography supports curves defined by NIST for use in ECDH operations.

ECDH

ECDH (Elliptic Curve Diffie-Hellman) is a key agreement protocol that allows two parties to establish a shared secret (such as a KEK or a PEK). In ECDH, Party A and B each have their own public-private key pairs and exchange public keys with each other (in the form of certificates for AWS Payment Cryptography) as well as key derivation metadata (derivation method, hash type and shared info). Both parties multiply their private key by the other's public key and due to elliptic curve properties, both parties are able to derive(generate) the resulting key.

EMV

EMV (originally Europay, Mastercard, Visa) is a technical body that works with payment stakeholders to create interoperable payment standards and technologies. One example standard is for chip/contactless cards and the payment terminals they interact with, including the cryptography used. EMV key derivation refers to method(s) of generating unique keys for each payment card based on an initial set of keys such as an `IMK`

HSM

A Hardware Security Module (HSM) is a physical device that protects cryptographic operations (for example, encryption, decryption, and digital signatures) as well as the underlying keys used for these operations.

KCAAS

A Key Custodian As A Service (KCAAS) provides a variety of services relating to key management. For payment keys, they can typically convert paper-based key components to electronic forms supported by AWS Payment Cryptography or convert electronically protected keys to paper-based components that might be required by certain vendors. They may also provide key escrow services for keys whose loss would be detrimental to your ongoing opeations. KCAAS vendors are able to help customers offload the operational burden of managing key material outside a secure service such as AWS Payment Cryptography in a way compliant with PCI DSS, PCI PIN, and PCI P2PE standards.

KCV

Key Check Value (KCV) refers to a variety of checksum methods primary used to compare to keys to each other without having access to the actual key material. KCV have also been used for integrity validation (especially when exchanging keys), although this role is now included as part of key block formats such as `TR-31`. For TDES keys, the KCV is computed by encrypting 8 bytes, each with value of zero, with the key to be checked and retaining the 3 highest order bytes of the encrypted result. For AES keys, the KCV is computed using a CMAC algorithm where the input data is 16 bytes of zero and retaining the 3 highest order bytes of the encrypted result.

KDH

A Key Distribution Host (KDH) is a device or system that is sending keys in a key exchange process such as TR-34. When sending keys from AWS Payment Cryptography, it is considered the KDH.

KIF

A Key Injection Facility (KIF) is a secure facility used for initializing payment terminals including loading them with encryption keys.

KRD

A Key Receiving Device (KRD) is a device that is receiving keys in a key exchange process such as TR-34. When sending keys to AWS Payment Cryptography, it is considered the KRD.

KSN

A Key Serial Number (KSN) is a value used as an input to DUKPT encryption/decryption to create unique encryption keys per transaction. The KSN typically consists of a BDK identifier,a semi-unique terminal ID as well as a transaction counter that increments on each transition processed on a given payment terminal. Per X9.24, for TDES the 10 byte KSN typically consists of 24 bits for the Key Set ID, 19 bits for the terminal ID and 21 bits for the transaction counter although the boundary between Key Set ID and terminal ID has no impact on the function of AWS Payment Cryptography. For AES, the 12 byte KSN typically consists of 32 bits for the BDK ID, 32 bits for the derivation identifier(ID) and 32 bits for the transaction counter.

MPoC

MPoC (Mobile Point of Sale on Commercial hardware) is a PCI standard that addresses the security requirements for solutions that enable merchants to accept cardholder PINs or contactless payments using a smartphone or other commercial off-the-shelf (COTS) mobile devices.

PAN

A Primary Account Number (PAN) is a unique identifier for an account such as a credit or debit card. Typically 13-19 digits in length. The first 6-8 digits identifies the network and the issuing bank.

PIN Block

A block of data containing a PIN during processing or transmission as well as other data elements. PIN block formats standardize the content of the PIN block and how it can be processed to retrieve the PIN. Most PIN block are composed of the PIN, the PIN length, and frequently contain part or all of the PAN. AWS Payment Cryptography supports ISO 9564-1 formats 0, 1, 3 and 4. Format 4 is required for AES keys. When verifying or translating PINs, there is a need to specify the PIN block of the incoming or outgoing data.

POI

Point of Interaction (POI), also frequently used anonymously with Point of Sale (POS), is the hardware device that the cardholder interacts with to present their payment credential. An example of a POI is the physical terminal in a merchant location. For the list of certified PCI PTS POI terminals, see the PCI website.

PSN

PAN Sequence Number (PSN) is a numeric value used to differentiate multiple cards issued with the same PAN.

Public key

When using asymmetric ciphers (RSA, ECC), the public key is the public component of a public-private key pair. The public key can be shared and distributed to entities that need to encrypt data for the owner of the public-private key pair. For digital signature operations, the public key is used to verify the signature.

Private key

When using asymmetric ciphers (RSA,ECC), the private key is the private component of a public-private key pair. The private key is used to decrypt data or create digital signatures. Similar to symmetric AWS Payment Cryptography keys, private keys are securely created by HSMs. They are decrypted only into the volatile memory of the HSM and only for the time needed to process your cryptographic request.

PVV

A PIN verification value (PVV) is a type of cryptographic output that can be used to verify a pin without storing the actual pin. Although it is a generic term, in the context of AWS Payment Cryptography, PVV refers to the Visa or ABA PVV method. This PVV is a four digit number whose inputs are card number, pan sequence number, the pan itself and a PIN verification key. During the validation stage, AWS Payment Cryptography internally recreates the PVV using the transaction data and compares it again the value that has been stored by the AWS Payment Cryptography customer. In this way, it is conceptually similar to a cryptographic hash or MAC.

RSA Wrap/Unwrap

RSA wrap uses an asymmetric key to wrap a symmetric key (such as a TDES key) for transmission to another system. Only the system with the matching private key can decrypt the payload and load the symmetric key. Conversely, RSA unwrap, will securely decrypt a key

encrypted using RSA and then load the key into the AWS Payment Cryptography. RSA wrap is a low level method of exchanging keys and does not transmit keys in key block format and does not utilize payload signing by the sending party. Alternate controls should be considered to ascertain providence and key attributes are not mutated.

TR-34 also utilizes RSA internally, but is a separate format and is not interoperable.

TR-31

TR-31 (formally defined as ANSI X9 TR 31) is a key block format that is defined by the American National Standards Institute (ANSI) to support defining key attributes in the same data structure as the key data itself. The TR-31 key block format defines a set of key attributes that are tied to the key so that they are held together. AWS Payment Cryptography uses TR-31 standardized terms whenever possible to ensure proper key separation and key purpose. TR-31 has been superseded by ANSI X9.143-2022.

TR-34

TR-34 is an implementation of ANSI X9.24-2 that described a protocol to securely distribute symmetric keys (such as 3DES and AES) using asymmetric techniques (such as RSA). AWS Payment Cryptography uses TR-34 methods to permit secure import and export of keys.

X9.143

X9.143 is a key block format that is defined by the American National Standards Institute (ANSI) to support securing a key and key attributes in the same data structure. The key block format defines a set of key attributes that are tied to the key so that they are held together. AWS Payment Cryptography uses X9.143 standardized terms whenever possible to ensure proper key separation and key purpose. X9.143 replaces the earlier TR-31 proposal although in most cases they are backwards and forward compatible and the terms are often used interchangeably.

Related services

AWS Key Management Service

AWS Key Management Service (AWS KMS) is a managed service that makes it easy for you to create and control the cryptographic keys that are used to protect your data. AWS KMS uses hardware security modules (HSMs) to protect and validate your AWS KMS keys.

AWS CloudHSM

AWS CloudHSM provides customers with dedicated general purpose HSM instances in the AWS Cloud. AWS CloudHSM can provide a variety of cryptographic functions such as creating keys, data signing or encrypting and decrypting data.

For more information

- To learn about the terms and concepts used in AWS Payment Cryptography, see AWS Payment Cryptography Concepts.
- For information about the AWS Payment Cryptography Control Plane API, see AWS Payment Cryptography Control Plane API Reference.
- For information about the AWS Payment Cryptography Data Plane API, see AWS Payment Cryptography Data Plane API Reference.
- For detailed technical information about how AWS Payment Cryptography uses cryptography and secures AWS Payment Cryptography keys, see Cryptographic details.

Endpoints for AWS Payment Cryptography

To connect programmatically to AWS Payment Cryptography, you use an endpoint, the URL of the entry point for the service. The AWS SDKs and the command line tools automatically use the default endpoint for the service in an AWS Region based on the region context of a request, so there's typically no need to explicitly set these values. When needed, you can specify a different endpoint for your API requests.

Control plane endpoints

Region Name	Region	Endpoint	Protocol	
US East (Ohio)	us-east-2	controlplane.payment-cryptography.us-east-2.amazonaws.com controlplane.payment-cryptography.us-east-2.api.aws	HTTPS HTTPS	

Region Name	Region	Endpoint	Protocol
US East (N. Virginia)	us-east-1	controlplane.payment-cryptography.us-east-1.amazonaws.com controlplane.payment-cryptography.us-east-1.api.aws	HTTPS HTTPS
US West (Oregon)	us-west-2	controlplane.payment-cryptography.us-west-2.amazonaws.com controlplane.payment-cryptography.us-west-2.api.aws	HTTPS HTTPS
Africa (Cape Town)	af-south-1	controlplane.payment-cryptography.af-south-1.amazonaws.com controlplane.payment-cryptography.af-south-1.api.aws	HTTPS HTTPS
Asia Pacific (Hyderabad)	ap-south-2	controlplane.payment-cryptography.ap-south-2.amazonaws.com controlplane.payment-cryptography.ap-south-2.api.aws	HTTPS HTTPS
Asia Pacific (Mumbai)	ap-south-1	controlplane.payment-cryptography.ap-south-1.amazonaws.com controlplane.payment-cryptography.ap-south-1.api.aws	HTTPS HTTPS
Asia Pacific (Osaka)	ap-northeast-3	controlplane.payment-cryptography.ap-northeast-3.amazonaws.com controlplane.payment-cryptography.ap-northeast-3.api.aws	HTTPS HTTPS

Region Name	Region	Endpoint	Protocol
Asia Pacific (Singapore)	ap-southeast-1	controlplane.payment-cryptography.ap-southeast-1.amazonaws.com controlplane.payment-cryptography.ap-southeast-1.api.aws	HTTPS HTTPS
Asia Pacific (Sydney)	ap-southeast-2	controlplane.payment-cryptography.ap-southeast-2.amazonaws.com controlplane.payment-cryptography.ap-southeast-2.api.aws	HTTPS HTTPS
Asia Pacific (Tokyo)	ap-northeast-1	controlplane.payment-cryptography.ap-northeast-1.amazonaws.com controlplane.payment-cryptography.ap-northeast-1.api.aws	HTTPS HTTPS
Canada (Central)	ca-central-1	controlplane.payment-cryptography.ca-central-1.amazonaws.com controlplane.payment-cryptography.ca-central-1.api.aws	HTTPS HTTPS
Europe (Frankfurt)	eu-central-1	controlplane.payment-cryptography.eu-central-1.amazonaws.com controlplane.payment-cryptography.eu-central-1.api.aws	HTTPS HTTPS
Europe (Ireland)	eu-west-1	controlplane.payment-cryptography.eu-west-1.amazonaws.com controlplane.payment-cryptography.eu-west-1.api.aws	HTTPS HTTPS

Region Name	Region	Endpoint	Protocol
Europe (London)	eu-west-2	controlplane.payment-cryptography.eu-west-2.amazonaws.com controlplane.payment-cryptography.eu-west-2.api.aws	HTTPS HTTPS
Europe (Paris)	eu-west-3	controlplane.payment-cryptography.eu-west-3.amazonaws.com controlplane.payment-cryptography.eu-west-3.api.aws	HTTPS HTTPS

Data plane endpoints

Region Name	Region	Endpoint	Protocol
US East (Ohio)	us-east-2	dataplane.payment-cryptography.us-east-2.amazonaws.com dataplane.payment-cryptography.us-east-2.api.aws	HTTPS HTTPS
US East (N. Virginia)	us-east-1	dataplane.payment-cryptography.us-east-1.amazonaws.com dataplane.payment-cryptography.us-east-1.api.aws	HTTPS HTTPS
US West (Oregon)	us-west-2	dataplane.payment-cryptography.us-west-2.amazonaws.com dataplane.payment-cryptography.us-west-2.api.aws	HTTPS HTTPS

Region Name	Region	Endpoint	Protocol
Africa (Cape Town)	af-south-1	dataplane.payment-cryptography.af-south-1.amazonaws.com dataplane.payment-cryptography.af-south-1.api.aws	HTTPS HTTPS
Asia Pacific (Hyderabad)	ap-south-2	dataplane.payment-cryptography.ap-south-2.amazonaws.com dataplane.payment-cryptography.ap-south-2.api.aws	HTTPS HTTPS
Asia Pacific (Mumbai)	ap-south-1	dataplane.payment-cryptography.ap-south-1.amazonaws.com dataplane.payment-cryptography.ap-south-1.api.aws	HTTPS HTTPS
Asia Pacific (Osaka)	ap-northeast-3	dataplane.payment-cryptography.ap-northeast-3.amazonaws.com dataplane.payment-cryptography.ap-northeast-3.api.aws	HTTPS HTTPS
Asia Pacific (Singapore)	ap-southeast-1	dataplane.payment-cryptography.ap-southeast-1.amazonaws.com dataplane.payment-cryptography.ap-southeast-1.api.aws	HTTPS HTTPS
Asia Pacific (Sydney)	ap-southeast-2	dataplane.payment-cryptography.ap-southeast-2.amazonaws.com dataplane.payment-cryptography.ap-southeast-2.api.aws	HTTPS HTTPS

Region Name	Region	Endpoint	Protocol
Asia Pacific (Tokyo)	ap-northeast-1	dataplane.payment-cryptography.ap-northeast-1.amazonaws.com dataplane.payment-cryptography.ap-northeast-1.api.aws	HTTPS HTTPS
Canada (Central)	ca-central-1	dataplane.payment-cryptography.ca-central-1.amazonaws.com dataplane.payment-cryptography.ca-central-1.api.aws	HTTPS HTTPS
Europe (Frankfurt)	eu-central-1	dataplane.payment-cryptography.eu-central-1.amazonaws.com dataplane.payment-cryptography.eu-central-1.api.aws	HTTPS HTTPS
Europe (Ireland)	eu-west-1	dataplane.payment-cryptography.eu-west-1.amazonaws.com dataplane.payment-cryptography.eu-west-1.api.aws	HTTPS HTTPS
Europe (London)	eu-west-2	dataplane.payment-cryptography.eu-west-2.amazonaws.com dataplane.payment-cryptography.eu-west-2.api.aws	HTTPS HTTPS
Europe (Paris)	eu-west-3	dataplane.payment-cryptography.eu-west-3.amazonaws.com dataplane.payment-cryptography.eu-west-3.api.aws	HTTPS HTTPS

Getting started with AWS Payment Cryptography

To get started with AWS Payment Cryptography, you'll first want to create keys and then use them in various cryptographic operations. The below tutorial provides a simple use case of generating a key to be used for generating/verifying CVV2 values. To try out other examples and to explore deployment patterns within AWS, please try out the following AWS Payment Cryptography Workshop or explore our sample project available on GitHub

This tutorial walks you through creating a single key and performing cryptographic operations using the key. Afterward, you delete the key if you no longer want it, which completes the key lifecycle.

Warning

Examples throughout this user guide may use sample values. We *strongly recommend* not using sample values in a production environment such as key serial numbers.

Topics

- Prerequisites
- Step 1: Create a key
- Step 2: Generate a CVV2 value using the key
- Step 3: Verify the value generated in step 2
- Step 4: Perform a negative test
- Step 5: (Optional) Clean up

Prerequisites

Before you begin, make sure that:

- You have permission to access the service. For more information, see IAM policies.
- You have the AWS CLI installed. You can also use AWS SDKs or AWS APIs to access AWS Payment Cryptography, but the instructions in this tutorial use the AWS CLI.

Step 1: Create a key

The first step is to create a key. For this tutorial, you create a CVK double-length 3DES (2KEY TDES) key for generating and verifying CVV/CVV2 values.

```
$ aws payment-cryptography create-key --exportable --key-attributes
 KeyAlgorithm=TDES_2KEY,KeyUsage=TR31_C0_CARD_VERIFICATION_KEY,KeyClass=SYMMETRIC_KEY,KeyModesO
```

The response echoes back the request parameters, including an ARN for subsequent calls as well as a Key Check Value (KCV).


```
{
    "Key": {
        "KeyArn": "arn:aws:payment-cryptography:us-east-2:111122223333:key/
tqv5yij6wtxx64pi",
        "KeyAttributes": {
            "KeyUsage": "TR31_C0_CARD_VERIFICATION_KEY",
            "KeyClass": "SYMMETRIC_KEY",
            "KeyAlgorithm": "TDES_2KEY",
            "KeyModesOfUse": {
                "Encrypt": false,
                "Decrypt": false,
                "Wrap": false,
                "Unwrap": false,
                "Generate": true,
                "Sign": false,
                "Verify": true,
                "DeriveKey": false,
                "NoRestrictions": false
            }
        },
        "KeyCheckValue": "CADDA1",
        "KeyCheckValueAlgorithm": "ANSI_X9_24",
        "Enabled": true,
        "Exportable": true,
        "KeyState": "CREATE_COMPLETE",
        "KeyOrigin": "AWS_PAYMENT_CRYPTOGRAPHY",
        "CreateTimestamp": "2023-06-05T06:41:46.648000-07:00",
        "UsageStartTimestamp": "2023-06-05T06:41:46.626000-07:00"
    }
}
```

Take note of the `KeyArn` that represents the key, for example *arn:aws:payment-cryptography:us-east-2:111122223333:key/tqv5yij6wtxx64pi*. You need that in the next step.

Step 2: Generate a CVV2 value using the key

In this step, you generate a CVV2 for a given PAN and expiration date using the key from step 1.

```
$  aws payment-cryptography-data generate-card-validation-data \
    --key-identifier arn:aws:payment-cryptography:us-east-2:111122223333:key/
tqv5yij6wtxx64pi \
    --primary-account-number=171234567890123 \
    --generation-attributes CardVerificationValue2={CardExpiryDate=0123}
```

```
{
    "CardDataGenerationKeyCheckValue": "CADDA1",
    "CardDataGenerationKeyIdentifier": "arn:aws:payment-cryptography:us-
east-2:111122223333:key/tqv5yij6wtxx64pi",
    "CardDataType": "CARD_VERIFICATION_VALUE_2",
    "CardDataValue": "144"
}
```

Take note of the `cardDataValue`, in this case the 3-digit number 144. You need that in the next step.

Step 3: Verify the value generated in step 2

In this example, you validate the CVV2 from step 2 using the key you created in step 1.

Run the following command to validate the CVV2.

```
$  aws payment-cryptography-data  verify-card-validation-data \
    --key-identifier arn:aws:payment-cryptography:us-east-2:111122223333:key/
tqv5yij6wtxx64pi \
    --primary-account-number=171234567890123 \
    --verification-attributes CardVerificationValue2={CardExpiryDate=0123} \
    --validation-data 144
```

```
{
    "KeyArn": "arn:aws:payment-cryptography:us-east-2:111122223333:key/
tqv5yij6wtxx64pi",
```

```
    "KeyCheckValue": "CADDA1"
}
```

The service returns an HTTP response of 200 to indicate that it validated the CVV2.

Step 4: Perform a negative test

In this step, you create a negative test where the CVV2 is not correct and does not validate. You attempt to validate an incorrect CVV2 using the key you created in step 1. This is an expected operation for example if a cardholder entered the wrong CVV2 at checkout.

```
$  aws payment-cryptography-data  verify-card-validation-data \
    --key-identifier arn:aws:payment-cryptography:us-east-2:111122223333:key/
tqv5yij6wtxx64pi \
    --primary-account-number=171234567890123 \
    --verification-attributes CardVerificationValue2={CardExpiryDate=0123} \
    --validation-data 999
```

```
Card validation data verification failed.
```

The service returns an HTTP response of 400 with the message "Card validation data verification failed" and a reason of INVALID_VALIDATION_DATA.

Step 5: (Optional) Clean up

Now you can delete the key you created in step 1. To minimize unrecoverable changes, the default key deletion period is seven days.

```
$ aws payment-cryptography delete-key \
    --key-identifier=arn:aws:payment-cryptography:us-east-2:111122223333:key/
tqv5yij6wtxx64pi
```

```
{
    "Key": {
        "CreateTimestamp": "2022-10-27T08:27:51.795000-07:00",
        "DeletePendingTimestamp": "2022-11-03T13:37:12.114000-07:00",
        "Enabled": true,
        "Exportable": true,
```

```
        "KeyArn": "arn:aws:payment-cryptography:us-east-2:111122223333:key/
tqv5yij6wtxx64pi",
        "KeyAttributes": {
            "KeyAlgorithm": "TDES_3KEY",
            "KeyClass": "SYMMETRIC_KEY",
            "KeyModesOfUse": {
                "Decrypt": true,
                "DeriveKey": false,
                "Encrypt": true,
                "Generate": false,
                "NoRestrictions": false,
                "Sign": false,
                "Unwrap": true,
                "Verify": false,
                "Wrap": true
            },
            "KeyUsage": "TR31_C0_CARD_VERIFICATION_KEY"
        },
        "KeyCheckValue": "CADDA1",
        "KeyCheckValueAlgorithm": "ANSI_X9_24",
        "KeyOrigin": "AWS_PAYMENT_CRYPTOGRAPHY",
        "KeyState": "DELETE_PENDING",
        "UsageStartTimestamp": "2022-10-27T08:27:51.753000-07:00"
    }
}
```

Take note of two fields in the output. The `deletePendingTimestamp` is set to seven days in the future by default. The keyState is set to `DELETE_PENDING`. You can cancel this deletion any time before the scheduled deletion time by calling [`restore-key`](#).

Managing keys

To get started with AWS Payment Cryptography, create an AWS Payment Cryptography key.

This section explains how to create and manage various AWS Payment Cryptography key types throughout their lifecycle. You'll learn how to create, view, and edit keys, as well as how to tag keys, create key aliases, and enable or disable keys.

An AWS Payment Cryptography key is a regional resource. If you intend to use a given key in multiple AWS Regions, you can enable Multi-Region key replication which securely copies key material and metadata to AWS Regions you specify within the same AWS Partition and Account. The source key in Multi-Region key replication is known as the Primary Region key (PRK) and this remains the authoritative source for all key management activities. The replicated key is known as the Replica Region key(RRK) and this is a read-only replica of the PRK. You should consider using Multi-Region keys with your keys to meet design goals around availability, disaster recovery, and low latency.

Topics

- Creating keys
- Listing keys
- Enabling and disabling keys
- Replicating AWS Payment Cryptography keys
- Deleting keys
- Importing and exporting keys
- Using aliases
- Get keys
- Tagging keys
- Understanding key attributes for AWS Payment Cryptography key

Creating keys

You can create AWS Payment Cryptography keys using the **CreateKey** API operation. When you create a key, you specify attributes such as the key algorithm, key usage, permitted operations, and whether it's exportable. You can't change these properties after you create the AWS Payment Cryptography key.

Note

If Multi-Region key replication is enabled for your AWS account and you create an Payment Cryptography key, this key will automatically become a Primary Region key (PRK). PRK is replicated even if you don't specify the `--replication-regions` parameter in the **CreateKey** command. For more information, see How Multi-Region key replication works.

Examples

- Creating a 3KEY TDES base derivation key
- Creating a 2KEY TDES key for CVV/CVV2
- Creating an HMAC key
- Creating an AES-256 key
- Creating a PIN Encryption Key (PEK)
- Creating an asymmetric (RSA) key
- Creating a PIN Verification Value (PVV) Key
- Creating an asymmetric ECC key

Creating a 3KEY TDES base derivation key

Example

This command creates creates a 3KEY TDES derivation key that will be replicated to US East (Ohio) and US West (Oregon) regions. The response includes the reques parameters, an Amazon Resource Name (ARN) for subsequent calls, and a Key Check Value (KCV).

```
$ aws payment-cryptography create-key --exportable --key-attributes \
     "KeyUsage=TR31_B0_BASE_DERIVATION_KEY, \
     KeyClass=SYMMETRIC_KEY,KeyAlgorithm=TDES_3KEY, \
     KeyModesOfUse={NoRestrictions=true}" \
     --replication-regions us-east-2 --region us-west-2
```

Example output:

```
{
    "Key": {
        "CreateTimestamp": "2022-10-26T16:04:11.642000-07:00",
```

```
        "Enabled": true,
        "Exportable": true,
        "KeyArn": "FE23D3",
        "KeyAttributes": {
            "KeyAlgorithm": "TDES_3KEY",
            "KeyClass": "SYMMETRIC_KEY",
            "KeyModesOfUse": {
                "Decrypt": false,
                "DeriveKey": true,
                "Encrypt": false,
                "Generate": false,
                "NoRestrictions": false,
                "Sign": false,
                "Unwrap": false,
                "Verify": true,
                "Wrap": false
            },
            "KeyUsage": "TR31_B0_BASE_DERIVATION_KEY"
        },
        "KeyCheckValue": "FE23D3",
        "KeyCheckValueAlgorithm": "ANSI_X9_24",
        "KeyOrigin": "AWS_PAYMENT_CRYPTOGRAPHY",
        "KeyState": "CREATE_COMPLETE",
        "UsageStartTimestamp": "2022-10-26T16:04:11.559000-07:00"
}
```

Creating a 2KEY TDES key for CVV/CVV2

Example

This command creates a 2KEY TDES key for generating and verifying CVV/CVV2 values. The response includes the request parameters, an Amazon Resource Name (ARN) for subsequent calls, and a Key Check Value (KCV).

```
$ aws payment-cryptography create-key --exportable --key-attributes
 KeyAlgorithm=TDES_2KEY, \
    KeyUsage=TR31_C0_CARD_VERIFICATION_KEY,KeyClass=SYMMETRIC_KEY, \
    KeyModesOfUse='{Generate=true,Verify=true}'
```

Example output:

```
{
    "Key": {
        "CreateTimestamp": "2022-10-26T16:04:11.642000-07:00",
        "Enabled": true,
        "Exportable": true,
        "KeyArn": "arn:aws:payment-cryptography:us-
east-2:111122223333:key/7f7g4spf3xcklhzu",
        "KeyAttributes": {
            "KeyAlgorithm": "TDES_2KEY",
            "KeyClass": "SYMMETRIC_KEY",
            "KeyModesOfUse": {
                "Decrypt": false,
                "DeriveKey": false,
                "Encrypt": false,
                "Generate": true,
                "NoRestrictions": false,
                "Sign": false,
                "Unwrap": false,
                "Verify": true,
                "Wrap": false
            },
            "KeyUsage": "TR31_C0_CARD_VERIFICATION_KEY"
        },
        "KeyCheckValue": "AEA5CD",
        "KeyCheckValueAlgorithm": "ANSI_X9_24",
        "KeyOrigin": "AWS_PAYMENT_CRYPTOGRAPHY",
        "KeyState": "CREATE_COMPLETE",
        "UsageStartTimestamp": "2022-10-26T16:04:11.559000-07:00"
    }
}
```

Creating an HMAC key

Example

HMAC keys are used for generating or verifying hash message authentication codes (HMAC). With HMAC keys, the hash type is assigned at the time of key creation (such as HMAC_SHA224 and HMAC_SHA512) and cannot be modified.

```
$ aws payment-cryptography create-key --exportable --key-attributes
 KeyAlgorithm=HMAC_SHA512,KeyUsage=TR31_M7_HMAC_KEY,KeyClass=SYMMETRIC_KEY,KeyModesOfUse='{Gene
 = true,Verify = true}'
```

Example output:

```
{
 "Key": {
 "KeyArn": "arn:aws:payment-cryptography:us-east-2:111122223333:key/qnobl5lghrzunce6",
 "KeyAttributes": {
 "KeyUsage": "TR31_M7_HMAC_KEY",
 "KeyClass": "SYMMETRIC_KEY",
 "KeyAlgorithm": "HMAC_SHA512",
 "KeyModesOfUse": {
 "Encrypt": false,
 "Decrypt": false,
 "Wrap": false,
 "Unwrap": false,
 "Generate": true,
 "Sign": false,
 "Verify": true,
 "DeriveKey": false,
 "NoRestrictions": false
 }
 },
 "KeyCheckValue": "2976E7",
 "KeyCheckValueAlgorithm": "HMAC",
 "Enabled": true,
 "Exportable": true,
 "KeyState": "CREATE_COMPLETE",
 "KeyOrigin": "AWS_PAYMENT_CRYPTOGRAPHY",
 "CreateTimestamp": "2025-07-30T10:06:12.142000-07:00",
 "UsageStartTimestamp": "2025-07-30T10:06:12.128000-07:00"
 }
}
```

Creating an AES-256 key

Example

This command creates an AES-256 symmetric key for data encryption and decryption. AES keys provide strong encryption for sensitive data and are commonly used in payment processing for encrypting cardholder data and other sensitive information, however TDES is more commonly used for issuer use cases like EMV.

```
$ aws payment-cryptography create-key --exportable --key-attributes
 KeyAlgorithm=AES_256,KeyUsage=TR31_D0_SYMMETRIC_DATA_ENCRYPTION_KEY,KeyClass=SYMMETRIC_KEY,Key
```

Example output:


```
{
    "Key": {
        "CreateTimestamp": "2025-02-02T10:15:30.142000-08:00",
        "Enabled": true,
        "Exportable": true,
        "KeyArn": "arn:aws:payment-cryptography:us-east-1:111122223333:key/
kwapwa6qaifllw2h",
        "KeyAttributes": {
            "KeyAlgorithm": "AES_256",
            "KeyClass": "SYMMETRIC_KEY",
            "KeyModesOfUse": {
                "Decrypt": true,
                "DeriveKey": false,
                "Encrypt": true,
                "Generate": false,
                "NoRestrictions": false,
                "Sign": false,
                "Unwrap": true,
                "Verify": false,
                "Wrap": true
            },
            "KeyUsage": "TR31_D0_SYMMETRIC_DATA_ENCRYPTION_KEY"
        },
        "KeyCheckValue": "2976F5",
        "KeyCheckValueAlgorithm": "CMAC",
        "KeyOrigin": "AWS_PAYMENT_CRYPTOGRAPHY",
        "KeyState": "CREATE_COMPLETE",
        "UsageStartTimestamp": "2025-02-02T10:15:30.128000-08:00"
    }
}
```

Creating a PIN Encryption Key (PEK)

Example

This command creates a 3KEY TDES key for encrypting PIN values although pin keys can also be AES depending on your need for interoperability. You can use this key to securely store PINs or decrypt PINs during verification, such as in a transaction. The response includes the request parameters, an ARN for subsequent calls, and a KCV.

```
$ aws payment-cryptography create-key --exportable --key-attributes \
    KeyAlgorithm=TDES_3KEY,KeyUsage=TR31_P0_PIN_ENCRYPTION_KEY, \

 KeyClass=SYMMETRIC_KEY,KeyModesOfUse='{Encrypt=true,Decrypt=true,Wrap=true,Unwrap=true}'
```

Example output:


```
{
    "Key": {
        "CreateTimestamp": "2022-10-27T08:27:51.795000-07:00",
        "Enabled": true,
        "Exportable": true,
        "KeyArn": "arn:aws:payment-cryptography:us-east-2:111122223333:key/
ivi5ksfsuplneuyt",
        "KeyAttributes": {
            "KeyAlgorithm": "TDES_3KEY",
            "KeyClass": "SYMMETRIC_KEY",
            "KeyModesOfUse": {
                "Decrypt": true,
                "DeriveKey": false,
                "Encrypt": true,
                "Generate": false,
                "NoRestrictions": false,
                "Sign": false,
                "Unwrap": true,
                "Verify": false,
                "Wrap": true
            },
            "KeyUsage": "TR31_P0_PIN_ENCRYPTION_KEY"
        },
        "KeyCheckValue": "7CC9E2",
        "KeyCheckValueAlgorithm": "ANSI_X9_24",
        "KeyOrigin": "AWS_PAYMENT_CRYPTOGRAPHY",
        "KeyState": "CREATE_COMPLETE",
        "UsageStartTimestamp": "2022-10-27T08:27:51.753000-07:00"
    }
}
```

Creating an asymmetric (RSA) key

Example

This command generates a new asymmetric RSA 2048-bit key pair. It creates a new private key and its matching public key. You can retrieve the public key using the getPublicCertificate API.

```
$ aws payment-cryptography create-key --exportable \
    --key-attributes
 KeyAlgorithm=RSA_2048,KeyUsage=TR31_D1_ASYMMETRIC_KEY_FOR_DATA_ENCRYPTION, \
    KeyClass=ASYMMETRIC_KEY_PAIR,KeyModesOfUse='{Encrypt=true,
 Decrypt=True,Wrap=True,Unwrap=True}'
```

Example output:

```
{
    "Key": {
        "CreateTimestamp": "2022-11-15T11:15:42.358000-08:00",
        "Enabled": true,
        "Exportable": true,
        "KeyArn": "arn:aws:payment-cryptography:us-east-2:111122223333:key/
nsq2i3mbg6sn775f",
        "KeyAttributes": {
            "KeyAlgorithm": "RSA_2048",
            "KeyClass": "ASYMMETRIC_KEY_PAIR",
            "KeyModesOfUse": {
                "Decrypt": true,
                "DeriveKey": false,
                "Encrypt": true,
                "Generate": false,
                "NoRestrictions": false,
                "Sign": false,
                "Unwrap": true,
                "Verify": false,
                "Wrap": true
            },
            "KeyUsage": "TR31_D1_ASYMMETRIC_KEY_FOR_DATA_ENCRYPTION"
        },
        "KeyCheckValue": "40AD487F",
        "KeyCheckValueAlgorithm": "SHA-1",
        "KeyOrigin": "AWS_PAYMENT_CRYPTOGRAPHY",
        "KeyState": "CREATE_COMPLETE",
        "UsageStartTimestamp": "2022-11-15T11:15:42.182000-08:00"
    }
}
```

Creating a PIN Verification Value (PVV) Key

Example

This command creates a 3KEY TDES key for generating PVV values. You can use this key to generate a PVV that can be compared against a subsequently calculated PVV. The response includes the request parameters, an ARN for subsequent calls, and a KCV.

```
$ aws payment-cryptography create-key --exportable \
    --key-attributes KeyAlgorithm=TDES_3KEY,KeyUsage=TR31_V2_VISA_PIN_VERIFICATION_KEY,
 \
    KeyClass=SYMMETRIC_KEY,KeyModesOfUse='{Generate=true,Verify=true}'
```

Example output:


```
{
    "Key": {
        "CreateTimestamp": "2022-10-27T10:22:59.668000-07:00",
        "Enabled": true,
        "Exportable": true,
        "KeyArn": "arn:aws:payment-cryptography:us-
east-2:111122223333:key/37y2tsl45p5zjbh2",
        "KeyAttributes": {
            "KeyAlgorithm": "TDES_3KEY",
            "KeyClass": "SYMMETRIC_KEY",
            "KeyModesOfUse": {
                "Decrypt": false,
                "DeriveKey": false,
                "Encrypt": false,
                "Generate": true,
                "NoRestrictions": false,
                "Sign": false,
                "Unwrap": false,
                "Verify": true,
                "Wrap": false
            },
            "KeyUsage": "TR31_V2_VISA_PIN_VERIFICATION_KEY"
        },
        "KeyCheckValue": "7F2363",
        "KeyCheckValueAlgorithm": "ANSI_X9_24",
        "KeyOrigin": "AWS_PAYMENT_CRYPTOGRAPHY",
        "KeyState": "CREATE_COMPLETE",
        "UsageStartTimestamp": "2022-10-27T10:22:59.614000-07:00"
    }
}
```

Creating an asymmetric ECC key

Example

This command generates an ECC key pair for establishing an ECDH (Elliptic Curve Diffie-Hellman) key agreement between two parties. With ECDH, each party generates its own ECC key pair with key purpose K3 and mode of use X, and they exchange public keys. Both parties then use their private key and the received public key to establish a shared derived key.

To maintain the single-use principle of cryptographic keys in payments, we recommend not reusing ECC key pairs for multiple purposes, such as ECDH key derivation and signing.

```
$ aws payment-cryptography create-key --exportable \
    --key-attributes
 KeyAlgorithm=ECC_NIST_P256,KeyUsage=TR31_K3_ASYMMETRIC_KEY_FOR_KEY_AGREEMENT, \
    KeyClass=ASYMMETRIC_KEY_PAIR,KeyModesOfUse='{DeriveKey=true}'
```

Example output:


```
{
    "Key": {
        "CreateTimestamp": "2024-10-17T01:31:55.908000+00:00",
        "Enabled": true,
        "Exportable": true,
        "KeyArn": "arn:aws:payment-cryptography:us-east-2:111122223333:key/
wc3rjsssguhxtilv",
        "KeyAttributes": {
            "KeyAlgorithm": "ECC_NIST_P256",
            "KeyClass": "ASYMMETRIC_KEY_PAIR",
            "KeyModesOfUse": {
                "Decrypt": false,
                "DeriveKey": true,
                "Encrypt": false,
                "Generate": false,
                "NoRestrictions": false,
                "Sign": false,
                "Unwrap": false,
                "Verify": false,
                "Wrap": false
            },
            "KeyUsage": "TR31_K3_ASYMMETRIC_KEY_FOR_KEY_AGREEMENT"
        },
        "KeyCheckValue": "7E34F19F",
        "KeyCheckValueAlgorithm": "SHA-1",
        "KeyOrigin": "AWS_PAYMENT_CRYPTOGRAPHY",
        "KeyState": "CREATE_COMPLETE",
        "UsageStartTimestamp": "2024-10-17T01:31:55.866000+00:00"
    }
}
```

Listing keys

Use the **ListKeys** operation to get a list of keys accessible to you in your account and Region.

Example

```
$ aws payment-cryptography list-keys
```

Example output:

```
{
    "Keys": [
        {
            "CreateTimestamp": "2022-10-12T10:58:28.920000-07:00",
            "Enabled": false,
            "Exportable": true,
            "KeyArn": "arn:aws:payment-cryptography:us-
east-2:111122223333:key/37y2tsl45p5zjbh2",
            "KeyAttributes": {
                "KeyAlgorithm": "TDES_3KEY",
                "KeyClass": "SYMMETRIC_KEY",
                "KeyModesOfUse": {
                    "Decrypt": true,
                    "DeriveKey": false,
                    "Encrypt": true,
                    "Generate": false,
                    "NoRestrictions": false,
                    "Sign": false,
                    "Unwrap": true,
                    "Verify": false,
                    "Wrap": true
                },
                "KeyUsage": "TR31_P1_PIN_GENERATION_KEY"
            },
            "KeyCheckValue": "7F2363",
            "KeyCheckValueAlgorithm": "ANSI_X9_24",
            "KeyOrigin": "AWS_PAYMENT_CRYPTOGRAPHY",
            "KeyState": "CREATE_COMPLETE",
            "UsageStopTimestamp": "2022-10-27T14:19:42.488000-07:00"
        }
    ]
}
```

Enabling and disabling keys

You can disable and re-enable AWS Payment Cryptography keys. When you create key, it is enabled by default. If you disable a key, it cannot be used in any cryptographic operation until you re-enable it. Start/stop usage commands take immediate effect, so it's recommended that you review usage before making such a change. You can also set a change (start or stop usage) to take effect in the future using the optional `timestamp` parameter.

Because it's temporary and easily undone, disabling an AWS Payment Cryptography key is a safer alternative to deleting an AWS Payment Cryptography key, an action that is destructive and irreversible. If you are considering deleting an AWS Payment Cryptography key, disable it first and ensure that you will not need to use the key to encrypt or decrypt data in the future.

Topics

- Start key usage
- Stop key usage

Start key usage

Key usage must be enabled in order to use a key for cryptographic operations. If a key is not enabled, you can use this operation to make it usable. The field `UsageStartTimestamp` will represent when the key became/will become active. This will be in the past for an enabled token, and in the future if pending activation.

Example

In this example, a key is requested to be enabled for key usage. The response includes the key information and the enable flag has been transitioned to true. This will also be reflected in list-keys response object.

```
$ aws payment-cryptography start-key-usage --key-identifier "arn:aws:payment-
cryptography:us-east-2:111122223333:key/alsuwfxug3pgy6xh"
```

```
{
      "Key": {
      "CreateTimestamp": "2022-10-12T10:58:28.920000-07:00",
      "Enabled": true,
      "Exportable": true,
      "KeyArn": "arn:aws:payment-cryptography:us-east-2:111122223333:key/
alsuwfxug3pgy6xh",
      "KeyAttributes": {
          "KeyAlgorithm": "TDES_3KEY",
          "KeyClass": "SYMMETRIC_KEY",
          "KeyModesOfUse": {
              "Decrypt": true,
              "DeriveKey": false,
              "Encrypt": true,
              "Generate": false,
              "NoRestrictions": false,
              "Sign": false,
              "Unwrap": true,
              "Verify": false,
              "Wrap": true
          },
          "KeyUsage": "TR31_P1_PIN_GENERATION_KEY"
      },
      "KeyCheckValue": "369D",
      "KeyCheckValueAlgorithm": "ANSI_X9_24",
      "KeyOrigin": "AWS_PAYMENT_CRYPTOGRAPHY",
      "KeyState": "CREATE_COMPLETE",
      "UsageStartTimestamp": "2022-10-27T14:09:59.468000-07:00"
  }
}
```

Stop key usage

If you no longer plan to use a key, you can stop the key usage to prevent further cryptographic operations. This operation is not permanent, so you are able to reverse it using starting key usage. You can also set a key to be disabled in the future. The field `UsageStopTimestamp` will represent when the key became/will become disabled.

Example

In this example, it's requested to stop key usage in the future. After execution, this key cannot be used for cryptographic operations unless re-enabled via start key usage The response includes the key information and the enable flag has been transitioned to false. This will also be reflected in list-keys response object.

```
$ aws payment-cryptography stop-key-usage --key-identifier "arn:aws:payment-cryptography:us-east-2:111122223333:key/alsuwfxug3pgy6xh"
```


```
{
  "Key": {
      "CreateTimestamp": "2022-10-12T10:58:28.920000-07:00",
      "Enabled": false,
      "Exportable": true,
      "KeyArn": "arn:aws:payment-cryptography:us-east-2:111122223333:key/alsuwfxug3pgy6xh",
      "KeyAttributes": {
          "KeyAlgorithm": "TDES_3KEY",
          "KeyClass": "SYMMETRIC_KEY",
          "KeyModesOfUse": {
              "Decrypt": true,
              "DeriveKey": false,
              "Encrypt": true,
              "Generate": false,
              "NoRestrictions": false,
              "Sign": false,
              "Unwrap": true,
              "Verify": false,
              "Wrap": true
          },
          "KeyUsage": "TR31_P1_PIN_GENERATION_KEY"
      },
      "KeyCheckValue": "369D",
      "KeyCheckValueAlgorithm": "ANSI_X9_24",
      "KeyOrigin": "AWS_PAYMENT_CRYPTOGRAPHY",
      "KeyState": "CREATE_COMPLETE",
      "UsageStopTimestamp": "2022-10-27T14:09:59.468000-07:00"
  }
}
```

Replicating AWS Payment Cryptography keys

AWS Payment Cryptography supports Multi-Region key replication, allowing you to securely distribute key material and metadata from any given AWS Payment Cryptography Key to one or more AWS Regions within the same AWS partition and account.

The source key is known as the Primary Region key (PRK) and remains the authoritative source for all key management activities while both the PRK and the Replica Region keys (RRK) can be used for cryptographic operations in their respective AWS Regions.

Benefits of Multi-Region key replication

The following outlines some benefits of Multi-Region key replication.

- *Easier setup for highly available applications* - AWS Payment Cryptography handles key distribution for you so you can use a key in multiple AWS Regions without needing to create decoupled copies of a given key.
- *High availability and low latency keys* - With Multi-Region key replication, you can access your keys in multiple AWS Regions making them highly available, resulting in lower latency.
- *Key material durability* - Replica Region keys are complete key replicas and can be used independently of their Primary Region key in cryptographic operations. A RRK provides a durable replica in the event of a catastrophic data loss of a PRK.

How Multi-Region key replication works

When Multi-Region key replication is enabled, the AWS Payment Cryptography service uses secure key distribution mechanisms to copy key material and metadata to the replica AWS Regions you specify. Changes to a Primary Region key metadata, such as key attributes, state, and enablement, are automatically replicated to the Replica Region keys.

Limitations and considerations

The following are some Multi-Region key replication limitations and considerations.

- You must enable this feature for either an AWS Region or specific Payment Cryptography keys.
  - If this feature is enabled for an AWS Region, all AWS Payment Cryptography keys created after enablement will be replicate to the specified AWS Region. Keys created in this Region will

become Primary Region keys. Existing keys in this Region will not be automatically replicated. You can enable Multi-Region key replication for existing keys within an AWS Region at the key level.

  - Each AWS Region can have unique Multi-Region key replication settings.
  - A key's Multi-Region replication settings takes precedence over the AWS Region Multi-Region key replication setting.
- A Replica Region key cannot be configured to replicate to other AWS Regions.
- Multi-Region key replication is available for symmetric Payment Cryptography keys like Triple Data Encryption Standard (3DES), Advanced Encryption Standard (AES), and Hash-based Message Authentication Code (HMAC).
- Asymmetric Payment Cryptography keys do not support Multi-Region key replication.
- Replica Region key are read-only keys. All changes to the Primary Region key will be applied to the Replica Region keys.
- Primary Region key changes are eventually consistent with Replica Region keys.
- Payment Cryptography keys can only be replicated with the same AWS partition and account.
- Replica Region key count towards your AWS account level AWS Payment Cryptography limit.
- The Primary Region key and Replica Region key use the same key identifier which allows you to reference both keys by the same ARN in IAM policies.
- You must have `CreateKey` permissions in the replica AWS Region for replication to succeed.

Enabling Multi-Region key replication

There are two ways you can enable Multi-Region key replication for AWS Payment Cryptography keys.

1. **AWS Region**: Multi-Region key replication is applied to all new keys created in that AWS Region when enabled. This method provides consistent replication for all keys.
2. **Specific AWS Payment Cryptography keys**: You can manage Multi-Region key replication for individual keys allowing a more granular level of control.

Once Multi-Region key replication is enabled, your Payment Cryptography keys will replicate to the AWS Regions you specify.

Important

Multi-Region key replication cannot be paused. Your keys are automatically replicated to the AWS Regions you specify once replication is enabled. Multi-Region key replication can be disabled for a specific AWS Region or Payment Cryptography keys. You must remove the AWS Region as a replication region from the Primary Region key to delete the Replica Region key.

Alternatively, you can call the `StopKeyUsage` API or **stop-key-usage** CLI command on your PRK to stop the usage of both the PRK and all associated RRKs. You'll be unable to use these keys in cryptographic operations. Using `StopKeyUsage` API or **stop-key-usage** CLI command will not stop the ongoing Multi-Region key replication enabled for your PRK.

You can check Multi-Region key replication settings for AWS Payment Cryptography keys in a specific AWS Region by calling the `GetDefaultKeyReplicationRegions` API or **get-default-key-replication-regions** CLI command. Keys in the AWS Region where you call this API action or command will become your PRK.

Use the following procedures to enable Multi-Region key replication.

For AWS Region

- Use the following command to enable Multi-Region key replication for an AWS Region you specify. In this example, Multi-Region key replication is enabled in US East (Ohio) and US West (Oregon). To use this command, replace the *italicized placeholder text* in the example command with your own information.

```
aws payment-cryptography enable-default-key-replication-regions \
    --replication-regions us-east-2 us-west-2
```

Note

Enabling Multi-Region key replication for an AWS Region will not change the replication configuration of any existing AWS Payment Cryptography keys. You can enable this feature for existing keys at the key level. Only keys created after Multi-Region key replication is enabled for an AWS Region will use the region replication settings.

For specific AWS Payment Cryptography keys

- Use the following command to enable Multi-Region key replication for specific Payment Cryptography keys. In this example, Multi-Region key replication is enabled in US East (Ohio). To use this command, replace the *italicized placeholder text* in the example command with your own information.

```
aws payment-cryptography add-key-replication-regions \
    --key-identifier arn:aws:payment-cryptography:us-east-2:111122223333:key/kwapwa6qaifllw2h \
    --replication-regions us-east-2
```

Alternatively, you can create a new Payment Cryptography key with this feature enabled by including the replication AWS Regions in your create key request.

Note

The key replication settings takes precedence over the AWS Region replication setting.

Disabling Multi-Region key replication

If you want to disable Multi-Region key replication, you can call either the **disable-default-key-replication** or **remove-key-replication-regions** CLI commands, depending on how Multi-Region key replication is enabled. You'll need to specify the key's ARN and the AWS Region to disable Multi-Region key replication.

Considerations

Replication region key deletions are eventually consistent.

You can check Multi-Region key replication settings for AWS Payment Cryptography keys in a specific AWS Region by calling the `GetDefaultKeyReplicationRegions` API or **get-default-key-replication-regions** CLI command.

Use the following procedures to disable Multi-Region key replication.

For AWS Region

- Use the following command to disable Multi-Region key replication for an AWS Region you specify. In this example, Multi-Region key replication is disabled in US East (Ohio). To use this command, replace the *italicized placeholder text* in the example command with your own information.

```
aws payment-cryptography disable-default-key-replication-regions \
    --replication-regions us-east-2
```

For specific AWS Payment Cryptography keys

- Use the following command to disable Multi-Region key replication for a specific Payment Cryptography key. In this example, Multi-Region key replication is disabling in US East (Ohio). To use this command, replace the *italicized placeholder text* in the example command with your own information.

```
aws payment-cryptography remove-key-replication-regions \
    --key-identifier arn:aws:payment-cryptography:us-
east-2:111122223333:key/kwapwa6qaifllw2h \
    --replication-regions us-east-2
```

Security considerations

The following are security considerations when using Multi-Region key replication for your Payment Cryptography keys. For more information, see Security best practices for AWS Payment Cryptography.

- Limit sharing key materials.
- Follow the principal of least privileges permissions when creating IAM policies.
- You can't make changes to the Replica Region key as it is a read-only key.

Best practices

The following are some best practices when using Multi-Region key replication with AWS Payment Cryptography keys.

- Ensure your application continues to work even if the Multi-Region key replication to the specified AWS Region is not immediate. If you need to know when Multi-Region key replication is complete, you can monitor with the GetKey API action. You can monitor key replication events with AWS CloudTrail.
- Test and implement automated deployment processes in case of fail-over from one AWS Region to another Region.

Pricing

You're charged for Replica Region keys you create with AWS Payment Cryptography. These keys are charged per AWS Region. For the latest Payment Cryptography pricing information, see the AWS Payment Cryptography pricing page.

Deleting keys

Deleting an AWS Payment Cryptography key deletes the key material and all metadata associated with the key and is irreversible unless a copy of the key is available outside of AWS Payment Cryptography. After a key is deleted, you can no longer decrypt the data that was encrypted under that key, which means that data may become unrecoverable. You should delete a key only when you are sure that you don't need to use it anymore and no other parties are utilizing this key. If you are not sure, consider stopping key usage instead of deleting it. You can re-enable a disabled key if you need to use it again later, but you cannot recover a deleted AWS Payment Cryptography key unless you are able to re-import it from another source.

Before deleting a key, you should ensure that you no longer need the key. AWS Payment Cryptography does not store the results of cryptographic operations like CVV2 and is unable to determine if a key is needed for any persistent cryptographic material.

AWS Payment Cryptography never deletes keys belonging to active AWS accounts unless you explicitly schedule them for deletion and the mandatory waiting period expires.

However, you might choose to delete an AWS Payment Cryptography key for one or more of the following reasons:

- To complete the key lifecycle for a key that you no longer need
- To avoid the management overhead associated with maintaining unused AWS Payment Cryptography keys

Note

If you close or delete your AWS account, your AWS Payment Cryptography key become inaccessible. You do not need to schedule deletion of your AWS Payment Cryptography key separate from closing the account.

AWS Payment Cryptography records an entry in your AWS CloudTrail log when you schedule deletion of the AWS Payment Cryptography key and when the AWS Payment Cryptography key is actually deleted.

When using Multi-Region key replication, deleting an Payment Cryptography key that is an Primary Region key (PRK), the Replica Region keys (RRK) will also be automatically deleted. A RRK cannot be deleted like a PRK. If you want to delete a RRK, you'll need to modify the replication regions for your PRK.

About the waiting period

Because deleting a key is irreversible, AWS Payment Cryptography requires you to set a waiting period of between 3–180 days. The default waiting period is seven days.

However, the actual waiting period might be up to 24 hours longer than the one you scheduled. To get the actual date and time when the AWS Payment Cryptography key will be deleted, use the GetKey operations. Be sure to note the time zone.

During the waiting period, the AWS Payment Cryptography key status and key state is **Pending deletion**.

Note

An AWS Payment Cryptography key pending deletion cannot be used in any cryptographic operations.

After the waiting period ends, AWS Payment Cryptography deletes the AWS Payment Cryptography key, its aliases, and all related AWS Payment Cryptography metadata.

Use the waiting period to ensure that you don't need the AWS Payment Cryptography key now or in the future. If you find that you do need the key during the waiting period, you can cancel

key deletion before the waiting period ends. After the waiting period ends, you cannot cancel key deletion, and the service deletes the key.

Example

In this example, a key is requested to be deleted. Besides the basic key information, two relevant fields are that key state has been changed to DELETE_PENDING and deletePendingTimestamp represents when the key is currently scheduled to delete.

```
$ aws payment-cryptography delete-key \
                    --key-identifier arn:aws:payment-cryptography:us-east-2:111122223333:key/kwapwa6qaifllw2h
```


```
  {
    "Key": {
        "KeyArn": "arn:aws:payment-cryptography:us-east-2:111122223333:key/kwapwa6qaifllw2h",
        "KeyAttributes": {
            "KeyUsage": "TR31_V2_VISA_PIN_VERIFICATION_KEY",
            "KeyClass": "SYMMETRIC_KEY",
            "KeyAlgorithm": "TDES_3KEY",
            "KeyModesOfUse": {
                "Encrypt": false,
                "Decrypt": false,
                "Wrap": false,
                "Unwrap": false,
                "Generate": true,
                "Sign": false,
                "Verify": true,
                "DeriveKey": false,
                "NoRestrictions": false
            }
        },
        "KeyCheckValue": "0A3674",
        "KeyCheckValueAlgorithm": "ANSI_X9_24",
        "Enabled": false,
        "Exportable": true,
        "KeyState": "DELETE_PENDING",
        "KeyOrigin": "AWS_PAYMENT_CRYPTOGRAPHY",
        "CreateTimestamp": "2023-06-05T12:01:29.969000-07:00",
        "UsageStopTimestamp": "2023-06-05T14:31:13.399000-07:00",
        "DeletePendingTimestamp": "2023-06-12T14:58:32.865000-07:00"
    }
}
```

Example

In this example, a pending deletion is cancelled. Once completed successfully, a key will no longer be deleted per the previous schedule. The response contains the basic key information; additionally, two relevant fields have changed - `KeyState` and `deletePendingTimestamp`. `KeyState` is returned to a value of CREATE_COMPLETE, while `DeletePendingTimestamp` is removed.

```
$ aws payment-cryptography restore-key --key-identifier arn:aws:payment-
cryptography:us-east-2:111122223333:key/kwapwa6qaifllw2h
```


```
{
    "Key": {
        "KeyArn": "arn:aws:payment-cryptography:us-east-2:111122223333:key/
kwapwa6qaifllw2h",
        "KeyAttributes": {
            "KeyUsage": "TR31_V2_VISA_PIN_VERIFICATION_KEY",
            "KeyClass": "SYMMETRIC_KEY",
            "KeyAlgorithm": "TDES_3KEY",
            "KeyModesOfUse": {
                "Encrypt": false,
                "Decrypt": false,
                "Wrap": false,
                "Unwrap": false,
                "Generate": true,
                "Sign": false,
                "Verify": true,
                "DeriveKey": false,
                "NoRestrictions": false
            }
        },
        "KeyCheckValue": "0A3674",
        "KeyCheckValueAlgorithm": "ANSI_X9_24",
        "Enabled": false,
        "Exportable": true,
        "KeyState": "CREATE_COMPLETE",
        "KeyOrigin": "AWS_PAYMENT_CRYPTOGRAPHY",
        "CreateTimestamp": "2023-06-08T12:01:29.969000-07:00",
        "UsageStopTimestamp": "2023-06-08T14:31:13.399000-07:00"
    }
}
```

Importing and exporting keys

You can import AWS Payment Cryptography keys from other solutions and export them to other solutions, such as HSMs. Many customers exchange keys with service providers using import and export functionality. We designed AWS Payment Cryptography to use a modern, electronic approach to key management that helps you maintain compliance and controls. We recommend using standards-based electronic key exchange instead of paper-based key components.

Minimum key strengths and the effect on import and export functions

PCI requires specific minimum key strengths for cryptographic operations, key storage, and key transmission. These requirements can change when PCI standards are revised. The rules specify that wrapping keys used for storage or transport must be at least as strong as the key being protected. We enforce this requirement automatically during export and prevent keys from being protected by weaker keys, as shown in the following table.

The following table shows the supported combinations of wrapping keys, keys to protect, and protection methods.

	Wrapping Key											
Key To Protect	**TDES**	**TDES**	**AES_**	**AES_**	**AES_**	**RSA_**	**RSA_**	**RSA_**	**ECC_**	**ECC_**	**ECC_**	**Notes**
TDES_2KEY	TR-3	TR-3	TR-3	TR-3	TR-3	TR-3 RSA	TR-3 RSA	TR-3 RSA	ECDH	ECDH	ECDH	
TDES_3KEY	✗ Not supp	TR-3	TR-3	TR-3	TR-3	TR-3 RSA	TR-3 RSA	TR-3 RSA	ECDH	ECDH	ECDH	
AES_128	✗ Not supp	✗ Not supp	TR-3	TR-3	TR-3	✗ Not supp	TR-3 RSA	TR-3 RSA	ECDH	ECDH	ECDH	
AES_192	✗ Not supp	✗ Not supp	✗ Not supp	TR-3	TR-3	✗ Not supp	✗ Not supp	✗ Not supp	✗ Not supp	ECDH	ECDH	

	Wrapping Key											
Key To Protect	**TDES**	**TDES**	**AES_**	**AES_**	**AES_**	**RSA_**	**RSA_**	**RSA_**	**ECC_**	**ECC_**	**ECC_**	**Notes**
AES_256	✗ Not supp	✗ Not supp	✗ Not supp	✗ Not supp	TR-3	✗ Not supp	✗ Not supp	✗ Not supp	✗ Not supp	✗ Not supp	ECDH	

For more information, see Appendix D - Minimum and Equivalent Key Sizes and Strengths for Approved Algorithms in the PCI HSM standards.

Key Encryption Key (KEK) Exchange

We recommend using ANSI X9.24 TR-34 standard. This initial key type can be called a Key Encryption Key (KEK), Zone Master Key (ZMK), or Zone Control Master Key (ZCMK). If your systems or partners don't support TR-34 yet you can use RSA Wrap/Unwrap. If your needs include exchanging AES-256 keys, you can use ECDH.

If you need to continue processing paper key components until all partners support electronic key exchange, consider using an offline HSM or utilizing a 3rd party key custodian as a service.

Note

To import your own test keys or to synchronize keys with your existing HSMs, please see the AWS Payment Cryptography sample code on GitHub.

Working Key (WK) Exchange

We use industry standards (ANSI X9.24 TR 31-2018 and X9.143) for exchanging working keys. This requires that you've already exchanged a KEK using TR-34, RSA Wrap, ECDH or similar schemes. This approach meets the PCI PIN requirement to cryptographically bind key material to its type and usage at all times. Working keys include acquirer working keys, issuer working keys, BDK, and IPEK.

Topics

- Import keys

- Export keys
- Advanced Topics

Import keys

Important

Examples require the latest version of the AWS CLI V2. Before getting started, make sure that you've upgraded to the latest version.

Contents

- Introduction to importing keys
- Importing symmetric keys
  - Import keys using asymmetric techniques (TR-34)
  - Import keys using asymmetric techniques (ECDH)
  - Import keys using asymmetric techniques (RSA Unwrap)
  - Import symmetric keys using a pre-established key exchange key (TR-31)
- Importing asymmetric (RSA, ECC) public keys
  - Importing RSA public keys
  - Importing ECC public keys

Introduction to importing keys

Note

When importing keys using X9.143, TR-31 or TR-34 key blocks, AWS Payment Cryptography typically retains (but does not utilize) any optional headers. The HM(HMAC hash type) header is used during cryptographic operations. The KP header (KCV of wrapping key) is specific to the import process and is not retained.

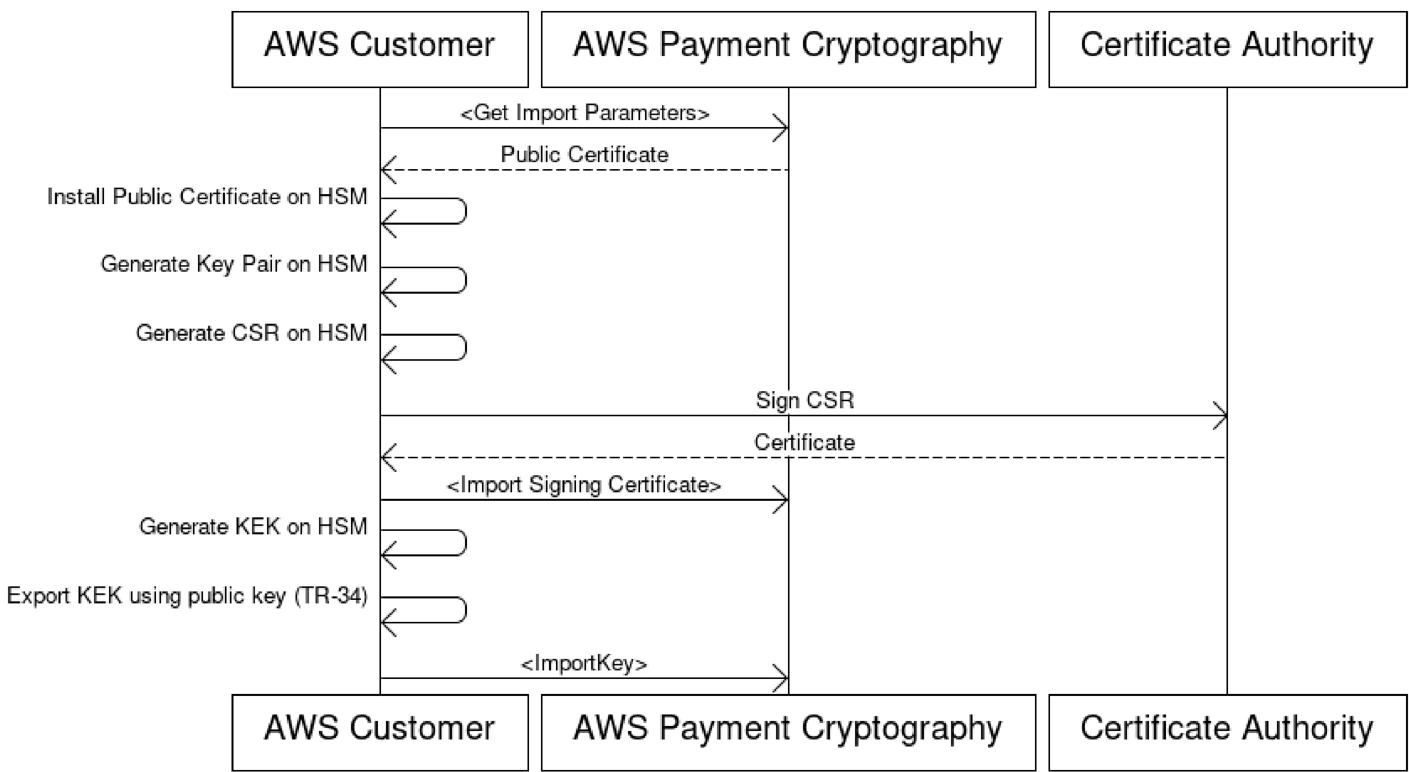
When exchanging keys with a counterparty, it is typically to first exchange a key exchange key (KEK). This key will then be used to protect subsequent keys. Using electronic formats, the KEK may

be exchanged use asymmetric techniques such as TR-34, ECDH or RSA wrap. Subsequent keys will be exchanged using a symmetric key exchange such as TR-31. This KEK will be long lived and may only be updated every few years based on policy and its defined crypto period.

If only one or two keys are being exchanged, you may also chose to use asymmetric techniques to directly exchange that key such as a BDK. AWS Payment Cryptography supports both methods of key exchange.

Importing symmetric keys

Import keys using asymmetric techniques (TR-34)



TR-34 uses RSA asymmetric cryptography to encrypt and sign symmetric keys for exchange. This ensures both confidentiality (encryption) and integrity (signature) of the wrapped key.

To import your own keys, check out the AWS Payment Cryptography sample project on GitHub. For instructions on how to import/export keys from other platforms, sample code is available on GitHub or consult the user guide for those platforms.

1. **Call the Initialize Import command**

Call `get-parameters-for-import` to initialize the import process. This API generates a key pair for key imports, signs the key, and returns the certificate and certificate root. Encrypt the key to be exported using this key. In TR-34 terminology, this is known as the KRD Cert. These certificates are base64 encoded, short-lived, and intended only for this purpose. Save the `ImportToken` value.

```
$ aws payment-cryptography get-parameters-for-import \
    --key-material-type TR34_KEY_BLOCK \
    --wrapping-key-algorithm RSA_2048
```



```
{
    "ImportToken": "import-token-bwxli6ocftypneu5",
    "ParametersValidUntilTimestamp": 1698245002.065,
    "WrappingKeyCertificateChain": "LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0....",
    "WrappingKeyCertificate": "LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0....",
    "WrappingKeyAlgorithm": "RSA_2048"
}
```


2. **Install public certificate on key source system**

   With most HSMs, you need to install, load, or trust the public certificate generated in step 1 to export keys using it. This could include the entire certificate chain or just the root certificate from step 1, depending on the HSM.

3. **Generate key pair on source system and provide certificate chain to AWS Payment Cryptography**

   To ensure integrity of the transmitted payload, the sending party (Key Distribution Host or KDH) signs it. Generate a public key for this purpose and create a public key certificate (X509) to provide back to AWS Payment Cryptography.

   When transferring keys from an HSM, create a key pair on that HSM. The HSM, a third party, or a service such as AWS Private CA can generate the certificate.

   Load the root certificate to AWS Payment Cryptography using the `importKey` command with KeyMaterialType of `RootCertificatePublicKey` and KeyUsageType of `TR31_S0_ASYMMETRIC_KEY_FOR_DIGITAL_SIGNATURE`.

   For intermediate certificates, use the `importKey` command with KeyMaterialType of `TrustedCertificatePublicKey` and KeyUsageType of

TR31_S0_ASYMMETRIC_KEY_FOR_DIGITAL_SIGNATURE. Repeat this process for multiple intermediate certificates. Use the KeyArn of the last imported certificate in the chain as an input to subsequent import commands.

Note

Don't import the leaf certificate. Provide it directly during the import command.

4. **Export key from source system**

   Many HSMs and related systems support exporting keys using the TR-34 norm. Specify the public key from step 1 as the KRD (encryption) cert and the key from step 3 as the KDH (signing) cert. To import to AWS Payment Cryptography, specify the format as TR-34.2012 non-CMS two pass format, which may also be referred to as the TR-34 Diebold format.

5. **Call Import Key**

   Call the importKey API with a KeyMaterialType of TR34_KEY_BLOCK. Use the keyARN of the last CA imported in step 3 for certificate-authority-public-key-identifier, the wrapped key material from step 4 for key-material, and the leaf certificate from step 3 for signing-key-certificate. Include the import-token from step 1.

```
$ aws payment-cryptography import-key \
    --key-material='{"Tr34KeyBlock": { \
    "CertificateAuthorityPublicKeyIdentifier": "arn:aws:payment-cryptography:us-
east-2:111122223333:key/zabouwe3574jysdl", \
    "ImportToken": "import-token-bwxli6ocftypneu5", \
    "KeyBlockFormat": "X9_TR34_2012", \
    "SigningKeyCertificate":
 "LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0tCk1JSUV2RENDQXFTZ0F3SUJ...", \
    "WrappedKeyBlock":
 "308205A106092A864886F70D010702A08205923082058E020101310D300B06096086480165030402013082003.
 \
    }'
```

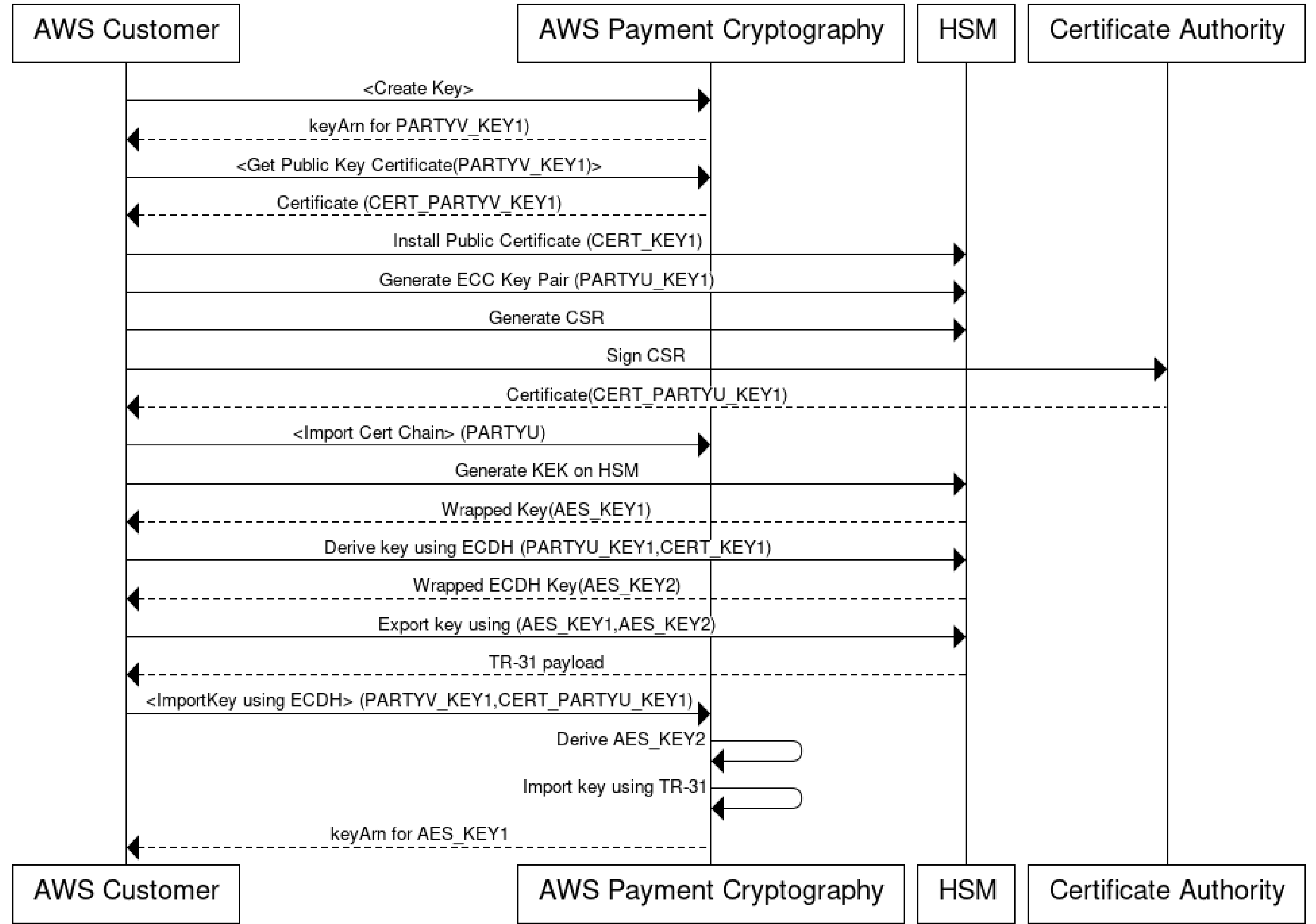
```
{
  "Key": {
    "CreateTimestamp": "2023-06-13T16:52:52.859000-04:00",
    "Enabled": true,
    "KeyArn": "arn:aws:payment-cryptography:us-east-2:111122223333:key/
ov6icy4ryas4zcza",
```

```
    "KeyAttributes": {
      "KeyAlgorithm": "TDES_3KEY",
      "KeyClass": "SYMMETRIC_KEY",
      "KeyModesOfUse": {
        "Decrypt": true,
        "DeriveKey": false,
        "Encrypt": true,
        "Generate": false,
        "NoRestrictions": false,
        "Sign": false,
        "Unwrap": true,
        "Verify": false,
        "Wrap": true
      },
      "KeyUsage": "TR31_K1_KEY_ENCRYPTION_KEY"
    },
    "KeyCheckValue": "CB94A2",
    "KeyCheckValueAlgorithm": "ANSI_X9_24",
    "KeyOrigin": "EXTERNAL",
    "KeyState": "CREATE_COMPLETE",
    "UsageStartTimestamp": "2023-06-13T16:52:52.859000-04:00"
  }
}
```

6. **Use imported key for cryptographic operations or subsequent import**

   If the imported KeyUsage was TR31_K0_KEY_ENCRYPTION_KEY, you can use this key for subsequent key imports using TR-31. For other key types (such as TR31_D0_SYMMETRIC_DATA_ENCRYPTION_KEY), you can use the key directly for cryptographic operations.

Import keys using asymmetric techniques (ECDH)



Elliptic Curve Diffie-Hellman (ECDH) uses ECC asymmetric cryptography to establish a shared key between two parties without requiring pre-exchanged keys. ECDH keys are ephemeral, so AWS Payment Cryptography does not store them. In this process, a one-time KBPK/KEK is derived using ECDH. That derived key is immediately used to wrap the actual key that you want to transfer, which could be another KBPK, an IPEK key, or other key types.

When importing, the sending system is commonly known as Party U (Initiator) and AWS Payment Cryptography is known as Party V (Responder).

Note

While ECDH can be used to exchange any symmetric key type, it is the only approach that can securely transfer AES-256 keys.

1. **Generate ECC Key Pair**

   Call `create-key` to create an ECC key pair for this process. This API generates a key pair for key imports or exports. At creation, specify what kind of keys can be derived using this ECC key. When using ECDH to exchange (wrap) other keys, use a value of `TR31_K1_KEY_BLOCK_PROTECTION_KEY`.

   Note

   Although low-level ECDH generates a derived key that can be used for any purpose, AWS Payment Cryptography limits the accidental reuse of a key for multiple purposes by allowing a key to only be used for a single derived-key type.

```
$ aws payment-cryptography create-key --exportable --key-attributes
 KeyAlgorithm=ECC_NIST_P256,KeyUsage=TR31_K3_ASYMMETRIC_KEY_FOR_KEY_AGREEMENT,KeyClass=ASYM
 --derive-key-usage "TR31_K1_KEY_BLOCK_PROTECTION_KEY"
```

```
{
                  "Key": {
                      "KeyArn": "arn:aws:payment-cryptography:us-
east-2:111122223333:key/wc3rjsssguhxtilv",
                      "KeyAttributes": {
                          "KeyUsage": "TR31_K3_ASYMMETRIC_KEY_FOR_KEY_AGREEMENT",
                          "KeyClass": "ASYMMETRIC_KEY_PAIR",
                          "KeyAlgorithm": "ECC_NIST_P256",
                          "KeyModesOfUse": {
                              "Encrypt": false,
                              "Decrypt": false,
                              "Wrap": false,
                              "Unwrap": false,
                              "Generate": false,
                              "Sign": false,
```

```
                    "Verify": false,
                    "DeriveKey": true,
                    "NoRestrictions": false
                }
            },
            "KeyCheckValue": "2432827F",
            "KeyCheckValueAlgorithm": "CMAC",
            "Enabled": true,
            "Exportable": true,
            "KeyState": "CREATE_COMPLETE",
            "KeyOrigin": "AWS_PAYMENT_CRYPTOGRAPHY",
            "CreateTimestamp": "2025-03-28T22:03:41.087000-07:00",
            "UsageStartTimestamp": "2025-03-28T22:03:41.068000-07:00"
        }
    }
```

2. **Get Public Key Certificate**

   Call `get-public-key-certificate` to receive the public key as an X.509 certificate signed by your account's CA that is specific to AWS Payment Cryptography in a specific region.

   Example

```
$ aws payment-cryptography get-public-key-certificate \
                --key-identifier arn:aws:payment-cryptography:us-east-2:111122223333:key/wc3rjsssguhxtilv
```

```
{
            "KeyCertificate": "LS0tLS1CRUdJTi...",
            "KeyCertificateChain": "LS0tLS1CRUdJT..."
      }
```

3. **Install public certificate on counterparty system (Party U)**

   With many HSMs, you need to install, load, or trust the public certificate generated in step 1 to export keys using it. This could include the entire certificate chain or just the root certificate from step 1, depending on the HSM. Consult your HSM documentation for more information.

4. **Generate ECC key pair on source system and provide certificate chain to AWS Payment Cryptography**

In ECDH, each party generates a key pair and agrees on a common key. For AWS Payment Cryptography to derive the key, it needs the counterparty's public key in X.509 public key format.

When transferring keys from an HSM, create a key pair on that HSM. For HSMs that support key blocks, the key header will look similar to `D0144K3EX00E0000`. When creating the certificate, you generally generate a CSR on the HSM and then the HSM, a third party, or a service such as AWS Private CA can generate the certificate.

Load the root certificate to AWS Payment Cryptography using the `importKey` command with KeyMaterialType of `RootCertificatePublicKey` and KeyUsageType of `TR31_S0_ASYMMETRIC_KEY_FOR_DIGITAL_SIGNATURE`.

For intermediate certificates, use the `importKey` command with KeyMaterialType of `TrustedCertificatePublicKey` and KeyUsageType of `TR31_S0_ASYMMETRIC_KEY_FOR_DIGITAL_SIGNATURE`. Repeat this process for multiple intermediate certificates. Use the `KeyArn` of the last imported certificate in the chain as an input to subsequent import commands.

Note

Don't import the leaf certificate. Provide it directly during the import command.

5. **Derive one-time key using ECDH on Party U HSM**

   Many HSMs and related systems support establishing keys using ECDH. Specify the public key from step 1 as the public key and the key from step 3 as the private key. For allowable options, such as derivation methods, see the API guide.

   Note

   The derivation parameters such as hash type must match exactly on both sides. Otherwise, you will generate a different key.

6. **Export key from source system**

   Finally, export the key you want to transport to AWS Payment Cryptography using standard TR-31 commands. Specify the ECDH derived key as the KBPK. The key to be exported can be

any TDES or AES key subject to TR-31 valid combinations, as long as the wrapping key is at least as strong as the key to be exported.

7. **Call Import Key**

   Call the `import-key` API with a KeyMaterialType of `DiffieHellmanTr31KeyBlock`. Use the KeyARN of the last CA imported in step 3 for `certificate-authority-public-key-identifier`, the wrapped key material from step 4 for `key-material`, and the leaf certificate from step 3 for `public-key-certificate`. Include the private key ARN from step 1.

```
$ aws payment-cryptography import-key \
          --key-material='{
            "DiffieHellmanTr31KeyBlock": {
              "CertificateAuthorityPublicKeyIdentifier": "arn:aws:payment-
cryptography:us-east-2:111122223333:key/swseahwtq2oj6zi5",
              "DerivationData": {
                "SharedInformation": "1234567890"
              },
              "DeriveKeyAlgorithm": "AES_256",
              "KeyDerivationFunction": "NIST_SP800",
              "KeyDerivationHashAlgorithm": "SHA_256",
              "PrivateKeyIdentifier": "arn:aws:payment-cryptography:us-
east-2:111122223333:key/wc3rjsssguhxtilv",
              "PublicKeyCertificate":
 "LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0tCk1JSUN....",
              "WrappedKeyBlock":
 "D0112K1TB00E0000D603CCA8ACB71517906600FF8F0F195A38776A7190A0EF0024F088A5342DB98E2735084A7
            }
          }'
```

```
{
        "Key": {
          "CreateTimestamp": "2025-03-13T16:52:52.859000-04:00",
          "Enabled": true,
          "KeyArn": "arn:aws:payment-cryptography:us-east-2:111122223333:key/
ov6icy4ryas4zcza",
          "KeyAttributes": {
            "KeyAlgorithm": "TDES_3KEY",
            "KeyClass": "SYMMETRIC_KEY",
            "KeyModesOfUse": {
              "Decrypt": true,
```

```
                "DeriveKey": false,
                "Encrypt": true,
                "Generate": false,
                "NoRestrictions": false,
                "Sign": false,
                "Unwrap": true,
                "Verify": false,
                "Wrap": true
              },
              "KeyUsage": "TR31_K1_KEY_ENCRYPTION_KEY"
            },
            "KeyCheckValue": "CB94A2",
            "KeyCheckValueAlgorithm": "ANSI_X9_24",
            "KeyOrigin": "EXTERNAL",
            "KeyState": "CREATE_COMPLETE",
            "UsageStartTimestamp": "2025-03-13T16:52:52.859000-04:00"
          }
        }
```


8. **Use imported key for cryptographic operations or subsequent import**

   If the imported KeyUsage was TR31_K0_KEY_ENCRYPTION_KEY, you can use this key for subsequent key imports using TR-31. For other key types (such as TR31_D0_SYMMETRIC_DATA_ENCRYPTION_KEY), you can use the key directly for cryptographic operations.

Import keys using asymmetric techniques (RSA Unwrap)

Overview: AWS Payment Cryptography supports RSA wrap/unwrap for key exchange when TR-34 isn't feasible. Like TR-34, this technique uses RSA asymmetric cryptography to encrypt symmetric keys for exchange. However, unlike TR-34, this method doesn't have the sending party sign the payload. Also, this RSA wrap technique doesn't maintain the integrity of the key metadata during transfer because it doesn't include key blocks.

Note

> You can use RSA wrap to import or export TDES and AES-128 keys.

1. **Call the Initialize Import command**

Call **get-parameters-for-import** to initialize the import process with a KeyMaterialType of KEY_CRYPTOGRAM. Use RSA_2048 for the WrappingKeyAlgorithm when exchanging TDES keys. Use RSA_3072 or RSA_4096 when exchanging TDES or AES-128 keys. This API generates a key pair for key imports, signs the key using a certificate root, and returns both the certificate and certificate root. Encrypt the key to be exported using this key. These certificates are short-lived and intended only for this purpose.

```
$ aws payment-cryptography get-parameters-for-import \
    --key-material-type KEY_CRYPTOGRAM \
    --wrapping-key-algorithm RSA_4096
```

```
{
  "ImportToken": "import-token-bwxli6ocftypneu5",
  "ParametersValidUntilTimestamp": 1698245002.065,
  "WrappingKeyCertificateChain": "LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0....",
  "WrappingKeyCertificate": "LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0....",
  "WrappingKeyAlgorithm": "RSA_4096"
}
```

2. **Install public certificate on key source system**

   With many HSMs, you need to install, load, or trust the public certificate (and/or its root) generated in step 1 to export keys using it.

3. **Export key from source system**

   Many HSMs and related systems support exporting keys using RSA wrap. Specify the public key from step 1 as the encryption cert (WrappingKeyCertificate). If you need the chain of trust, use the WrappingKeyCertificateChain from step 1. When exporting the key from your HSM, specify the format as RSA, with Padding Mode = PKCS#1 v2.2 OAEP (with SHA 256 or SHA 512).

4. **Call import-key**

   Call the **import-key** API with a KeyMaterialType of KeyMaterial. You need the ImportToken from step 1 and the key-material (wrapped key material) from step 3. Provide the key parameters (such as Key Usage) because RSA wrap doesn't use key blocks.

```
$ cat import-key-cryptogram.json
```

```
{
  "KeyMaterial": {
    "KeyCryptogram": {
      "Exportable": true,
      "ImportToken": "import-token-bwxli6ocftypneu5",
      "KeyAttributes": {
        "KeyAlgorithm": "AES_128",
        "KeyClass": "SYMMETRIC_KEY",
        "KeyModesOfUse": {
          "Decrypt": true,
          "DeriveKey": false,
          "Encrypt": true,
          "Generate": false,
          "NoRestrictions": false,
          "Sign": false,
          "Unwrap": true,
          "Verify": false,
          "Wrap": true
        },
        "KeyUsage": "TR31_K0_KEY_ENCRYPTION_KEY"
      },
      "WrappedKeyCryptogram": "18874746731....",
      "WrappingSpec": "RSA_OAEP_SHA_256"
    }
  }
}
```

```
$ aws payment-cryptography import-key --cli-input-json file://import-key-
cryptogram.json
```

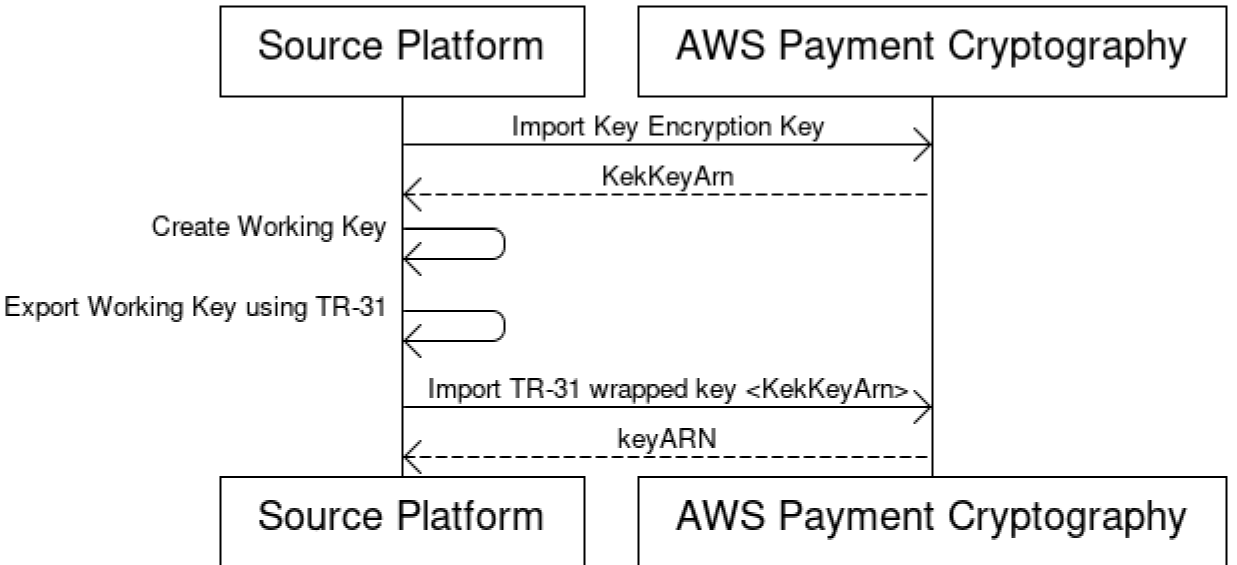
```
{
  "Key": {
    "KeyOrigin": "EXTERNAL",
    "Exportable": true,
    "KeyCheckValue": "DA1ACF",
    "UsageStartTimestamp": 1697643478.92,
    "Enabled": true,
    "KeyArn": "arn:aws:payment-cryptography:us-east-2:111122223333:key/
kwapwa6qaifllw2h",
    "CreateTimestamp": 1697643478.92,
    "KeyState": "CREATE_COMPLETE",
```

```
    "KeyAttributes": {
      "KeyAlgorithm": "AES_128",
      "KeyModesOfUse": {
        "Encrypt": true,
        "Unwrap": true,
        "Verify": false,
        "DeriveKey": false,
        "Decrypt": true,
        "NoRestrictions": false,
        "Sign": false,
        "Wrap": true,
        "Generate": false
      },
      "KeyUsage": "TR31_K0_KEY_ENCRYPTION_KEY",
      "KeyClass": "SYMMETRIC_KEY"
    },
    "KeyCheckValueAlgorithm": "CMAC"
  }
}
```

5. **Use imported key for cryptographic operations or subsequent import**

   If the imported KeyUsage was TR31_K0_KEY_ENCRYPTION_KEY or TR31_K1_KEY_BLOCK_PROTECTION_KEY, you can use this key for subsequent key imports using TR-31. If the key type was any other type (such as TR31_D0_SYMMETRIC_DATA_ENCRYPTION_KEY), you can use the key directly for cryptographic operations.

Import symmetric keys using a pre-established key exchange key (TR-31)



When exchanging multiple keys or supporting key rotation, partners typically first exchange an initial key encryption key (KEK). You can do this using techniques such as paper key components or, for AWS Payment Cryptography, using TR-34.

After establishing a KEK, you can use it to transport subsequent keys (including other KEKs). AWS Payment Cryptography supports this key exchange using ANSI TR-31, which is widely used and supported by HSM vendors.

1. **Import Key Encryption Key (KEK)**

   Make sure you've already imported your KEK and have the keyARN (or keyAlias) available.

2. **Create key on source platform**

   If the key doesn't exist, create it on the source platform. Alternatively, you can create the key on AWS Payment Cryptography and use the **export** command.

3. **Export key from source platform**

   When exporting, specify the export format as TR-31. The source platform will ask for the key to export and the key encryption key to use.

4. **Import into AWS Payment Cryptography**

When calling the **import-key** command, use the keyARN (or alias) of your key encryption key for `WrappingKeyIdentifier`. Use the output from the source platform for `WrappedKeyBlock`.

Example

```
$ aws payment-cryptography import-key \
    --key-material='{"Tr31KeyBlock": { \
    "WrappingKeyIdentifier": "arn:aws:payment-cryptography:us-
east-2:111122223333:key/ov6icy4ryas4zcza", \
    "WrappedKeyBlock":
 "D0112B0AX00E00002E0A3D58252CB67564853373D1EBCC1E23B2ADE7B15E967CC27B85D5999EF58E11662991F
 \
    }'
```

```
{
  "Key": {
    "KeyArn": "arn:aws:payment-cryptography:us-east-2:111122223333:key/
kwapwa6qaifllw2h",
    "KeyAttributes": {
      "KeyUsage": "TR31_D0_SYMMETRIC_DATA_ENCRYPTION_KEY",
      "KeyClass": "SYMMETRIC_KEY",
      "KeyAlgorithm": "AES_128",
      "KeyModesOfUse": {
        "Encrypt": true,
        "Decrypt": true,
        "Wrap": true,
        "Unwrap": true,
        "Generate": false,
        "Sign": false,
        "Verify": false,
        "DeriveKey": false,
        "NoRestrictions": false
      }
    },
    "KeyCheckValue": "0A3674",
    "KeyCheckValueAlgorithm": "CMAC",
    "Enabled": true,
    "Exportable": true,
    "KeyState": "CREATE_COMPLETE",
    "KeyOrigin": "EXTERNAL",
    "CreateTimestamp": "2023-06-02T07:38:14.913000-07:00",
    "UsageStartTimestamp": "2023-06-02T07:38:14.857000-07:00"
  }
}
```

Importing asymmetric (RSA, ECC) public keys

All certificates imported must be at least as strong as their issuing(predecessor) certificate in the chain. This means that a RSA_2048 CA can only be used to protect a RSA_2048 leaf certificate and an ECC certificate must be protected by another ECC certificate of equivalent strength. An ECC P384 certificate can only be issued by a P384 or P521 CA. All certificates must be unexpired at the time of import.

Importing RSA public keys

AWS Payment Cryptography supports importing public RSA keys as X.509 certificates. To import a certificate, first import its root certificate. All certificates must be unexpired at the time of import. The certificate should be in PEM format and base64 encoded.

1. **Import Root Certificate into AWS Payment Cryptography**

   Use the following command to import the root certificate:

Example

2. **Import Public Key Certificate into AWS Payment Cryptography**

   You can now import a public key. As TR-34 and ECDH rely on passing the leaf certificate at run-time, this option is only used when encrypting data using a public key from another system. KeyUsage will be set to TR31_D1_ASYMMETRIC_KEY_FOR_DATA_ENCRYPTION.

Example

```
$ aws payment-cryptography import-key \
    --key-material='{"Tr31KeyBlock": { \
    "WrappingKeyIdentifier": "arn:aws:payment-cryptography:us-
east-2:111122223333:key/ov6icy4ryas4zcza", \
    "WrappedKeyBlock":
 "D0112B0AX00E00002E0A3D58252CB67564853373D1EBCC1E23B2ADE7B15E967CC27B85D5999EF58E11662991F
 \
    }'
```

```
{
  "Key": {
    "CreateTimestamp": "2023-08-08T18:55:46.815000+00:00",
    "Enabled": true,
    "KeyArn": "arn:aws:payment-cryptography:us-
east-2:111122223333:key/4kd6xud22e64wcbk",
    "KeyAttributes": {
      "KeyAlgorithm": "RSA_4096",
      "KeyClass": "PUBLIC_KEY",
      "KeyModesOfUse": {
        "Decrypt": false,
        "DeriveKey": false,
        "Encrypt": false,
        "Generate": false,
        "NoRestrictions": false,
        "Sign": false,
        "Unwrap": false,
        "Verify": true,
        "Wrap": false
      },
      "KeyUsage": "TR31_S0_ASYMMETRIC_KEY_FOR_DIGITAL_SIGNATURE"
    },
    "KeyOrigin": "EXTERNAL",
    "KeyState": "CREATE_COMPLETE",
    "UsageStartTimestamp": "2023-08-08T18:55:46.815000+00:00"
  }
}
```

Importing ECC public keys

AWS Payment Cryptography supports importing public ECC keys as X.509 certificates. To import a certificate, first import its root CA certificate and any intermediate certificates. All certificates must be unexpired at the time of import. The certificate should be in PEM format and base64 encoded.

1. **Import ECC Root Certificate into AWS Payment Cryptography**

   Use the following command to import the root certificate:

Example

2. **Import Intermediate Certificate into AWS Payment Cryptography**

   Use the following command to import an intermediate certificate:

Example

3. **Import Public Key Certificate(Leaf) into AWS Payment Cryptography**

   Although you can import a leaf ECC certificate, there is currently no defined functions in AWS Payment Cryptography for it besides storage. This is because when using ECDH functions, the leaf certificate is passed at runtime.

Export keys

Contents

- Export symmetric keys
  - Export keys using asymmetric techniques (TR-34)
  - Export keys using asymmetric techniques (ECDH)
  - Export keys using asymmetric techniques (RSA Wrap)
  - Export symmetric keys using a pre-established key exchange key (TR-31)
- Export DUKPT Initial Keys (IPEK/IK)
- Specify key block headers for export
  - Common Headers
- Export asymmetric (RSA) keys

Export symmetric keys

Important

Make sure you have the latest version of AWS CLI before you begin. To upgrade, see Installing the AWS CLI.

Export keys using asymmetric techniques (TR-34)

TR-34 uses RSA asymmetric cryptography to encrypt and sign symmetric keys for exchange. The encryption protects confidentiality, while the signature ensures integrity. When you export keys, AWS Payment Cryptography acts as the key distribution host (KDH), and your target system becomes the key receiving device (KRD).

Note

If your HSM supports TR-34 export but not TR-34 import, we recommend that you first establish a shared KEK between your HSM and AWS Payment Cryptography using TR-34. You can then use TR-31 to transfer your remaining keys.

1. **Initialize the export process**

   Run **get-parameters-for-export** to generate a key pair for key exports. We use this key pair to sign the TR-34 payload. In TR-34 terminology, this is the KDH signing certificate. The certificates are short-lived and valid only for the duration specified in `ParametersValidUntilTimestamp`.

   Note

   All certificates are in base64 encoding.

   Example

   ```
   $ aws payment-cryptography get-parameters-for-export \
       --signing-key-algorithm RSA_2048 \
       --key-material-type TR34_KEY_BLOCK
   ```

   ```
   {
     "SigningKeyCertificate":
    "LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0tCk1JSUV2RENDQXFTZ0F3SUJ...",
     "SigningKeyCertificateChain": "LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS....",
     "SigningKeyAlgorithm": "RSA_2048",
     "ExportToken": "export-token-au7pvkbsq4mbup6i",
     "ParametersValidUntilTimestamp": "2023-06-13T15:40:24.036000-07:00"
   }
   ```

2. **Import the AWS Payment Cryptography certificate to your receiving system**

   Import the certificate chain from step 1 to your receiving system.

3. **Set up your receiving system's certificates**

To protect the transmitted payload, the sending party (KDH) encrypts it. Your receiving system (typically your HSM or your partner's HSM) needs to generate a public key and create an X.509 public key certificate. You can use AWS Private CA to generate certificates, but you can use any certificate authority.

After you have the certificate, import the root certificate to AWS Payment Cryptography using the **ImportKey** command. Set `KeyMaterialType` to `RootCertificatePublicKey` and `KeyUsageType` to `TR31_S0_ASYMMETRIC_KEY_FOR_DIGITAL_SIGNATURE`.

We use `TR31_S0_ASYMMETRIC_KEY_FOR_DIGITAL_SIGNATURE` as the `KeyUsageType` because this is the root key that signs the leaf certificate. You don't need to import leaf certificates into AWS Payment Cryptography—you can pass them inline.

Note

If you previously imported the root certificate, skip this step. For intermediate certificates, use `TrustedCertificatePublicKey`.

4. **Export your key**

Call the **ExportKey** API with `KeyMaterialType` set to `TR34_KEY_BLOCK`. You need to provide:

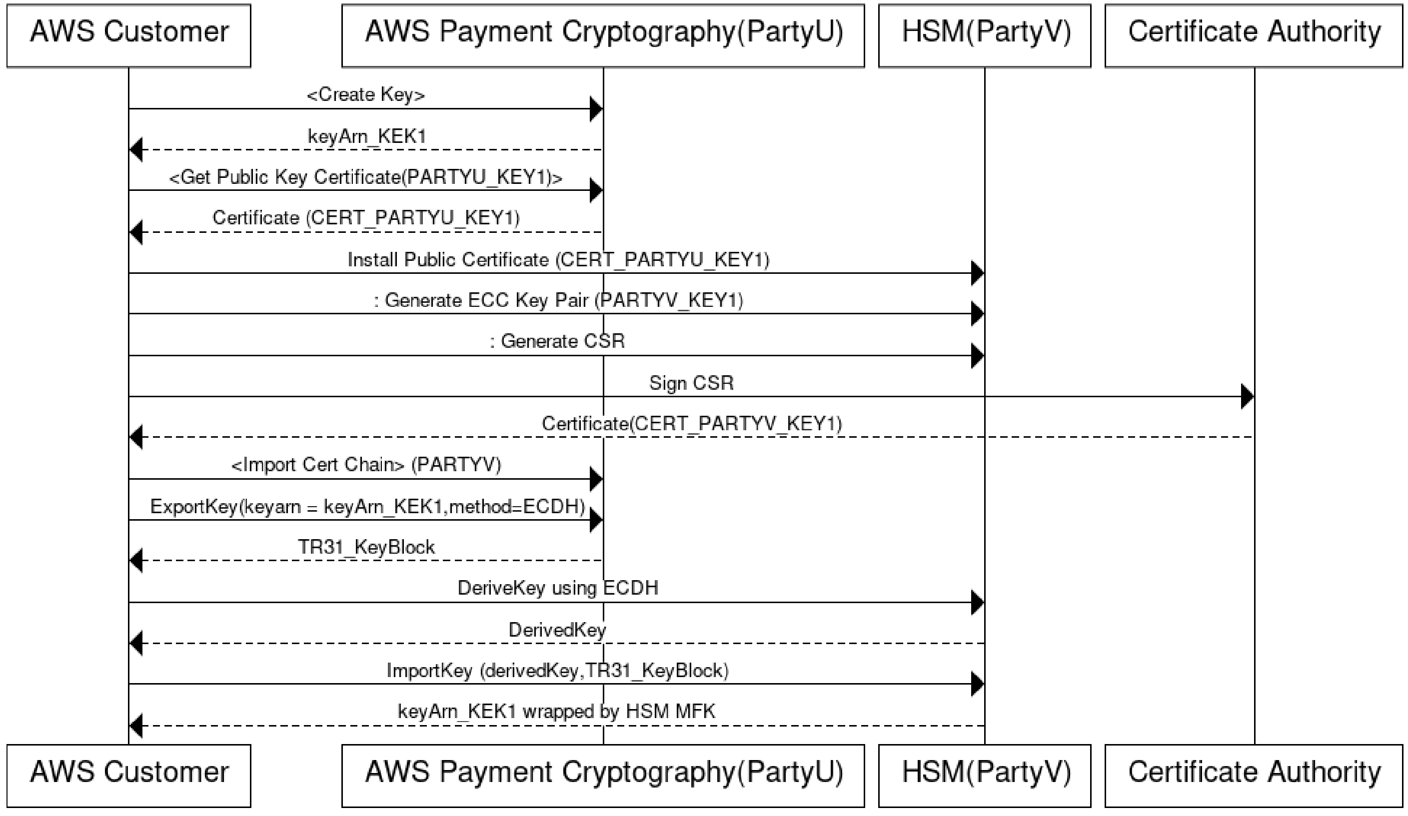
- The keyARN of the root CA from step 3 as the `CertificateAuthorityPublicKeyIdentifier`
- The leaf certificate from step 3 as the `WrappingKeyCertificate`
- The keyARN (or alias) of the key you want to export as the `--export-key-identifier`
- The export-token from step 1

Example

```
$ aws payment-cryptography export-key \
    --export-key-identifier "example-export-key" \
    --key-material '{"Tr34KeyBlock": { \
    "CertificateAuthorityPublicKeyIdentifier": "arn:aws:payment-cryptography:us-
east-2:111122223333:key/4kd6xud22e64wcbk", \
    "ExportToken": "export-token-au7pvkbsq4mbup6i", \
    "KeyBlockFormat": "X9_TR34_2012", \
    "WrappingKeyCertificate":
 "LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0tCk1JSUV2RENDQXFXZ0F3SUJBZ0lSQ..."} \
    }'
```

```
{
  "WrappedKey": {
    "KeyMaterial": "308205A106092A864886F70D010702A08205923082058...",
    "WrappedKeyMaterialFormat": "TR34_KEY_BLOCK"
  }
}
```

Export keys using asymmetric techniques (ECDH)



Elliptic Curve Diffie-Hellman (ECDH) uses ECC asymmetric cryptography to establish a shared key between two parties without requiring pre-exchanged keys. ECDH keys are ephemeral, so AWS Payment Cryptography does not store them. In this process, a one-time KBPK/KEK is derived using ECDH. That derived key is immediately used to wrap the key you want to transfer, which could be another KBPK, a BDK, an IPEK key, or other key types.

When exporting, AWS Payment Cryptography is referred to as Party U (Initiator) and the receiving system is known as Party V (Responder).

Note

ECDH can be used to exchange any symmetric key type, but it is the only approach that can be used to transfer AES-256 keys if a KEK is not already established.

1. **Generate ECC Key Pair**

Call `create-key` to create an ECC key pair for this process. This API generates a key pair for key imports or exports. At creation, specify what kind of keys can be derived using this ECC key. When using ECDH to exchange (wrap) other keys, use a value of `TR31_K1_KEY_BLOCK_PROTECTION_KEY`.

Note

Although low-level ECDH generates a derived key that can be used for any purpose, AWS Payment Cryptography limits the accidental reuse of a key for multiple purposes by allowing a key to only be used for a single derived-key type.

```
$ aws payment-cryptography create-key --exportable --key-attributes
 KeyAlgorithm=ECC_NIST_P256,KeyUsage=TR31_K3_ASYMMETRIC_KEY_FOR_KEY_AGREEMENT,KeyClass=ASYM
 --derive-key-usage "TR31_K1_KEY_BLOCK_PROTECTION_KEY"
```


```
{
        "Key": {
            "KeyArn": "arn:aws:payment-cryptography:us-east-2:111122223333:key/
wc3rjsssguhxtilv",
            "KeyAttributes": {
                "KeyUsage": "TR31_K3_ASYMMETRIC_KEY_FOR_KEY_AGREEMENT",
                "KeyClass": "ASYMMETRIC_KEY_PAIR",
                "KeyAlgorithm": "ECC_NIST_P256",
                "KeyModesOfUse": {
                    "Encrypt": false,
                    "Decrypt": false,
                    "Wrap": false,
                    "Unwrap": false,
                    "Generate": false,
                    "Sign": false,
                    "Verify": false,
                    "DeriveKey": true,
                    "NoRestrictions": false
                }
            },
            "KeyCheckValue": "2432827F",
            "KeyCheckValueAlgorithm": "CMAC",
            "Enabled": true,
            "Exportable": true,
```

```
            "KeyState": "CREATE_COMPLETE",
            "KeyOrigin": "AWS_PAYMENT_CRYPTOGRAPHY",
            "CreateTimestamp": "2025-03-28T22:03:41.087000-07:00",
            "UsageStartTimestamp": "2025-03-28T22:03:41.068000-07:00"
        }
    }
```

2. **Get Public Key Certificate**

   Call `get-public-key-certificate` to receive the public key as an X.509 certificate signed by your account's CA that is specific to AWS Payment Cryptography in a specific region.

   Example

```
$ aws payment-cryptography get-public-key-certificate \
          --key-identifier arn:aws:payment-cryptography:us-east-2:111122223333:key/wc3rjsssguhxtilv
```

```
{
      "KeyCertificate": "LS0tLS1CRUdJTi...",
      "KeyCertificateChain": "LS0tLS1CRUdJT..."
    }
```

3. **Install public certificate on counterparty system (Party V)**

   With many HSMs, you need to install, load, or trust the public certificate generated in step 1 to establish keys. This could include the entire certificate chain or just the root certificate, depending on the HSM. Consult your HSM documentation for specific instructions.

4. **Generate ECC key pair on source system and provide certificate chain to AWS Payment Cryptography**

   In ECDH, each party generates a key pair and agrees on a common key. For AWS Payment Cryptography to derive the key, it needs the counterparty's public key in X.509 public key format.

   When transferring keys from an HSM, create a key pair on that HSM. For HSMs that support key blocks, the key header will look similar to `D0144K3EX00E0000`. When creating the certificate, you generally generate a CSR on the HSM, and then the HSM, a third party, or a service such as AWS Private CA can generate the certificate.

Load the root certificate to AWS Payment Cryptography using the `importKey` command with KeyMaterialType of `RootCertificatePublicKey` and KeyUsageType of `TR31_S0_ASYMMETRIC_KEY_FOR_DIGITAL_SIGNATURE`.

For intermediate certificates, use the `importKey` command with KeyMaterialType of `TrustedCertificatePublicKey` and KeyUsageType of `TR31_S0_ASYMMETRIC_KEY_FOR_DIGITAL_SIGNATURE`. Repeat this process for multiple intermediate certificates. Use the `KeyArn` of the last imported certificate in the chain as an input to subsequent export commands.

Note

Don't import the leaf certificate. Provide it directly during the export command.

5. **Derive key and export key from AWS Payment Cryptography**

When exporting, the service derives a key using ECDH and then immediately uses it as the KBPK to wrap the key to export using TR-31. The key to be exported can be any TDES or AES key subject to TR-31 valid combinations, as long as the wrapping key is at least as strong as the key to be exported.

```
$ aws payment-cryptography export-key \
            --export-key-identifier arn:aws:payment-cryptography:us-
west-2:529027455495:key/e3a65davqhbpjm4h \
            --key-material='{
              "DiffieHellmanTr31KeyBlock": {
                "CertificateAuthorityPublicKeyIdentifier": "arn:aws:payment-
cryptography:us-east-2:111122223333:key/swseahwtq2oj6zi5",
                "DerivationData": {
                  "SharedInformation": "ADEF567890"
                },
                "DeriveKeyAlgorithm": "AES_256",
                "KeyDerivationFunction": "NIST_SP800",
                "KeyDerivationHashAlgorithm": "SHA_256",
                "PrivateKeyIdentifier": "arn:aws:payment-cryptography:us-
east-2:111122223333:key/wc3rjsssguhxtilv",
                "PublicKeyCertificate": "LS0tLS1CRUdJTiBDRVJUSUZJQ0FUR..."
              }
            }'
```

```
{
            "WrappedKey": {
                "WrappedKeyMaterialFormat": "TR31_KEY_BLOCK",
                "KeyMaterial":
 "D0112K1TB00E00007012724C0FAAF64DA50E2FF4F9A94DF50441143294E0E995DB2171554223EAA56D078C4CF
                "KeyCheckValue": "E421AD",
                "KeyCheckValueAlgorithm": "ANSI_X9_24"
            }
        }
```

6. **Derive one-time key using ECDH on Party V HSM**

   Many HSMs and related systems support establishing keys using ECDH. Specify the public key from step 1 as the public key and the key from step 3 as the private key. For allowable options, such as derivation methods, see the API guide.

   Note

   The derivation parameters such as hash type must match exactly on both sides. Otherwise, you will generate a different key.

7. **Import key to target system**

   Finally, import the key from AWS Payment Cryptography using standard TR-31 commands. Specify the ECDH derived key as the KBPK and use the TR-31 key block that was previously exported from AWS Payment Cryptography.

Export keys using asymmetric techniques (RSA Wrap)

When TR-34 isn't available, you can use RSA wrap/unwrap for key exchange. Like TR-34, this method uses RSA asymmetric cryptography to encrypt symmetric keys. However, RSA wrap doesn't include:

- Payload signing by the sending party
- Key blocks that maintain key metadata integrity during transport

Note

You can use RSA wrap to export TDES and AES-128 keys.

1. **Create an RSA key and certificate on your receiving system**

   Create or identify an RSA key for receiving the wrapped key. We require keys to be in X.509 certificate format. Make sure the certificate is signed by a root certificate that you can import into AWS Payment Cryptography.

2. **Import the root public certificate to AWS Payment Cryptography**

   Use **import-key** with the `--key-material` option to import the certificate

```
$ aws payment-cryptography import-key \
    --key-material='{"RootCertificatePublicKey": { \
    "KeyAttributes": { \
    "KeyAlgorithm": "RSA_4096", \
    "KeyClass": "PUBLIC_KEY", \
    "KeyModesOfUse": {"Verify": true}, \
    "KeyUsage": "TR31_S0_ASYMMETRIC_KEY_FOR_DIGITAL_SIGNATURE"}, \
    "PublicKeyCertificate": "LS0tLS1CRUdJTiBDRV..."} \
    }'
```

```
{
  "Key": {
    "CreateTimestamp": "2023-09-14T10:50:32.365000-07:00",
    "Enabled": true,
    "KeyArn": "arn:aws:payment-cryptography:us-east-2:111122223333:key/
nsq2i3mbg6sn775f",
    "KeyAttributes": {
      "KeyAlgorithm": "RSA_4096",
      "KeyClass": "PUBLIC_KEY",
      "KeyModesOfUse": {
        "Decrypt": false,
        "DeriveKey": false,
        "Encrypt": false,
        "Generate": false,
        "NoRestrictions": false,
        "Sign": false,
        "Unwrap": false,
```

```
          "Verify": true,
          "Wrap": false
        },
        "KeyUsage": "TR31_S0_ASYMMETRIC_KEY_FOR_DIGITAL_SIGNATURE"
      },
      "KeyOrigin": "EXTERNAL",
      "KeyState": "CREATE_COMPLETE",
      "UsageStartTimestamp": "2023-09-14T10:50:32.365000-07:00"
    }
}
```

3. **Export your key**

   Tell AWS Payment Cryptography to export your key using your leaf certificate. You need to specify:

   - The ARN for the root certificate you imported in step 2
   - The leaf certificate for export
   - The symmetric key to export

   The output is a hex-encoded binary wrapped (encrypted) version of your symmetric key.

Example – Exporting a key

```
$ cat export-key.json
```

```
{
  "ExportKeyIdentifier": "arn:aws:payment-cryptography:us-east-2:111122223333:key/
tqv5yij6wtxx64pi",
  "KeyMaterial": {
    "KeyCryptogram": {
      "CertificateAuthorityPublicKeyIdentifier": "arn:aws:payment-cryptography:us-
east-2:111122223333:key/zabouwe3574jysdl",
      "WrappingKeyCertificate": "LS0tLS1CRUdJTiBDEXAMPLE...",
      "WrappingSpec": "RSA_OAEP_SHA_256"
    }
  }
}
```

```
$ aws payment-cryptography export-key \
    --cli-input-json file://export-key.json
```

```
{
  "WrappedKey": {
    "KeyMaterial":
 "18874746731E9E1C4562E4116D1C2477063FCB08454D757D81854AEAEE0A52B1F9D303FA29C02DC82AE778535
    "WrappedKeyMaterialFormat": "KEY_CRYPTOGRAM"
  }
}
```

4. **Import the key to your receiving system**

   Many HSMs and related systems support importing keys using RSA unwrap (including AWS Payment Cryptography). When importing, specify:

   - The public key from step 1 as the encryption certificate
   - The format as RSA
   - Padding Mode as PKCS#1 v2.2 OAEP (with SHA 256)

Note

We output the wrapped key in hexBinary format. You might need to convert the format if your system requires a different binary representation, such as base64.

Export symmetric keys using a pre-established key exchange key (TR-31)

When exchanging multiple keys or supporting key rotation, you typically first exchange an initial key encryption key (KEK) using paper key components or, with AWS Payment Cryptography, using TR-34. After establishing a KEK, you can use it to transport subsequent keys, including other KEKs. We support this key exchange using ANSI TR-31, which is widely supported by HSM vendors.

1. **Set up your Key Encryption Key (KEK)**

   Make sure you have already exchanged your KEK and have the keyARN (or keyAlias) available.

2. **Create your key on AWS Payment Cryptography**

   Create your key if it doesn't already exist. Alternatively, you can create the key on your other system and use the import command.

3. **Export your key from AWS Payment Cryptography**

   When exporting in TR-31 format, specify the key you want to export and the wrapping key to use.

Example – Exporting a key using TR31 key block

```
$ aws payment-cryptography export-key \
    --key-material='{"Tr31KeyBlock": \
    { "WrappingKeyIdentifier": "arn:aws:payment-cryptography:us-
east-2:111122223333:key/ov6icy4ryas4zcza" }}' \
    --export-key-identifier arn:aws:payment-cryptography:us-
east-2:111122223333:key/5rplquuwozodpwsp
```

```
{
  "WrappedKey": {
    "KeyCheckValue": "73C263",
    "KeyCheckValueAlgorithm": "ANSI_X9_24",
    "KeyMaterial":
 "D0144K0AB00E0000A24D3ACF3005F30A6E31D533E07F2E1B17A2A003B338B1E79E5B3AD4FBF7850FACF9A3784
    "WrappedKeyMaterialFormat": "TR31_KEY_BLOCK"
  }
}
```

4. **Import the key to your system**

   Use your system's import key implementation to import the key.

Export DUKPT Initial Keys (IPEK/IK)

When using DUKPT, you can generate a single Base Derivation Key (BDK) for a fleet of terminals. The terminals don't have direct access to the BDK. Instead, each terminal receives a unique initial terminal key, known as IPEK or Initial Key (IK). Each IPEK is derived from the BDK using a unique Key Serial Number (KSN).

The KSN structure varies by encryption type:

- For TDES: The 10-byte KSN includes:
  - 24 bits for the Key Set ID
  - 19 bits for the terminal ID
  - 21 bits for the transaction counter
- For AES: The 12-byte KSN includes:
  - 32 bits for the BDK ID

- 32 bits for the derivation identifier (ID)
- 32 bits for the transaction counter

We provide a mechanism to generate and export these initial keys. You can export the generated keys using TR-31, TR-34, or RSA wrap methods. Note that IPEK keys are not persisted and can't be used for subsequent operations on AWS Payment Cryptography.

We don't enforce the split between the first two parts of the KSN. If you want to store the derivation identifier with the BDK, you can use AWS tags.

> **Note**
>
> The counter portion of the KSN (32 bits for AES DUKPT) isn't used for IPEK/IK derivation. For example, inputs of 12345678901234560001 and 12345678901234569999 will generate the same IPEK.

```
$ aws payment-cryptography export-key \
    --key-material='{"Tr31KeyBlock": { \
    "WrappingKeyIdentifier": "arn:aws:payment-cryptography:us-east-2:111122223333:key/
ov6icy4ryas4zcza"}} ' \
    --export-key-identifier arn:aws:payment-cryptography:us-east-2:111122223333:key/
tqv5yij6wtxx64pi \
    --export-attributes 'ExportDukptInitialKey={KeySerialNumber=12345678901234560001}'
```

```
{
"WrappedKey": {
    "KeyCheckValue": "73C263",
    "KeyCheckValueAlgorithm": "ANSI_X9_24",
    "KeyMaterial":
 "B0096B1TX00S000038A8A06588B9011F0D5EEF1CCAECFA6962647A89195B7A98BDA65DDE7C57FEA507559AF2A5D60
    "WrappedKeyMaterialFormat": "TR31_KEY_BLOCK"
}
}
```

Specify key block headers for export

You can modify or append key block information when exporting in ASC TR-31 or TR-34 formats. The following table describes the TR-31 key block format and which elements you can modify during export.

Key Block Attribute	Purpose	Can you modify during export?	Notes
Version ID	Defines the method used to protect the key material. The standard includes: • Version A and C (key variant - deprecated) • Version B (derivati on using TDES) • Version D (key derivation using AES)	No	We use version B for TDES wrapping keys and version D for AES wrapping keys. We support versions A and C only for import operations.
Key Block Length	Specifies the length of the remaining message	No	We calculate this value automatically. The length might appear incorrect before decrypting the payload because we may add key padding as required by the specification.
Key Usage	Defines the permitted purposes for the key, such as:	No	

Key Block Attribute	Purpose	Can you modify during export?	Notes
	• C0 (Card Verification) • B0 (Base Derivation Key)		
Algorithm	Specifies the algorithm of the underlying key. We support: • T (TDES) • H (HMAC) • A (AES)	No	We export this value as-is.
Key Usage	Defines allowed operations, such as: • Generate and Verify (C) • Encrypt/Decrypt/Wrap/Unwrap (B)	Yes*	
Key Version	Indicates the version number for key replacement/rotation. Defaults to 00 if not specified.	Yes - Can append	

Key Block Attribute	Purpose	Can you modify during export?	Notes
Key Exportability	Controls whether the key can be exported: • N - No Exportability • E - Export according to X9.24 (key blocks) • S - Export under key block or non-key block formats	Yes*	
Optional Key Blocks	Yes - Can append	Optional key blocks are name/value pairs cryptographically bound to the key. For example, KeySetID for DUKPT keys. We automatically calculate the number of blocks, length of each block, and padding block (PB) based on your name/value pair input.	

**When modifying values, your new value must be more restrictive than the current value in AWS Payment Cryptography.* For example:

- If the current key mode of use is Generate=True,Verify=True, you can change it to Generate=True,Verify=False
- If the key is already set to not exportable, you can't change it to exportable

When you export keys, we automatically apply the current values from the key being exported. However, you might want to modify or append those values before sending to the receiving system. Here are some common scenarios:

- When exporting a key to a payment terminal, set its exportability to `Not Exportable` because terminals typically only import keys and shouldn't export them.
- When you need to pass associated key metadata to the receiving system, use TR-31 optional headers to cryptographically bind the metadata to the key instead of creating a custom payload.
- Set the Key Version using the `KeyVersion` field to track key rotation.

TR-31/X9.143 defines common headers, but you can use other headers as long as they meet AWS Payment Cryptography parameters and your receiving system can accept them. For more information about key block headers during export, see Key Block Headers in the API Guide.

Here's an example of exporting a BDK key (for instance, to a KIF) with these specifications:

- Key version: 02
- KeyExportability: NON_EXPORTABLE
- KeySetID: 00ABCDEFAB (00 indicates TDES key, ABCDEFABCD is the initial key)

Because we don't specify key modes of use, this key inherits the mode of use from arn:aws:payment-cryptography:us-east-2:111122223333:key/5rplquuwozodpwsp (DeriveKey = true).

> **Note**
>
> Even when you set exportability to Not Exportable in this example, the KIF can still:
>
> - Derive keys such as IPEK/IK used in DUKPT
> - Export these derived keys to install on devices
>
> This is specifically allowed by the standards.

```
$ aws payment-cryptography export-key \
    --key-material='{"Tr31KeyBlock": { \
```

```
    "WrappingKeyIdentifier": "arn:aws:payment-cryptography:us-east-2:111122223333:key/
ov6icy4ryas4zcza", \
    "KeyBlockHeaders": { \
    "KeyModesOfUse": { \
    "Derive": true}, \
    "KeyExportability": "NON_EXPORTABLE", \
    "KeyVersion": "02", \
    "OptionalBlocks": { \
    "BI": "00ABCDEFABCD"}}} \
    }' \
    --export-key-identifier arn:aws:payment-cryptography:us-
east-2:111122223333:key/5rplquuwozodpwsp
```

```
{
"WrappedKey": {
    "WrappedKeyMaterialFormat": "TR31_KEY_BLOCK",
    "KeyMaterial": "EXAMPLE_KEY_MATERIAL_TR31",
    "KeyCheckValue": "A4C9B3",
    "KeyCheckValueAlgorithm": "ANSI_X9_24"
    }
}
```

Common Headers

X9.143 defines certain headers for common use cases. With the exception of the HM(HMAC Hash) header, AWS Payment Cryptography does not parse or utilize these headers.

Header Name	Purpose	Typical Validation	Notes
BI	Base Derivation Key Identifier for DUKPT	2 hex characters (00 for TDES, 11 for AES) then 10 hex character s for TDES KSI or 8 hex characters for BDK ID (AES DUKPT).	Contains the (BDK ID, for AES DUKPT) or the Key Set Identifie r (KSI, for TDES DUKPT). Can be used when exchanging the BDK ID or the KSI, but do not need to exchange the other data contained in the IK and KS blocks.

Header Name	Purpose	Typical Validation	Notes
			Typically BI is used when transmitting to a KIF whereas IK or KS are used when injecting into the terminal itself.
HM	Specifies the hash type for HMAC operations	• 10 – SHA-1 • 20 – SHA-224 • 21 – SHA-256 • 22 – SHA-384 • 23 – SHA-512 • 24 – SHA-512/224 • 25 – SHA-512/256 • 30 – SHA3-224 • 31 – SHA3-256 • 32 – SHA3-384 • 33 – SHA3-512 • 40 – SHAKE128 • 41 – SHAKE256	The service automatically populates this field on export and will parse it on import. Hash types not supported by the service such as SHAKE128 can be imported but may not be usuable for cryptographic functions.
IK	Initial Key Serial Number for AES DUKPT	16 hex characters	This value is used to instantiate the use of the Initial DUKPT key on the receiving device and it identifies the Initial Key derived from a BDK. This field typically contains the derivation data but no counter. Use KS for TDES DUKPT.

Header Name	Purpose	Typical Validation	Notes
KS	Initial Key Serial Number for TDES DUKPT	20 hex characters	This value is used to instantiate the use of the Initial DUKPT key on the receiving device and it identifies the Initial Key derived from a BDK. This field typically contains the derivation data + a zeroized counter value. Use IK for AES DUKPT.
KP	KCV of the wrapping key	2 hex characters represent the KCV method (00 for X9.24 method and 01 for CMAC method). Followed by KCV value which is typically 6 hex characters. For example 010FA329 represents KCV of 0FA329 calculated using 01(CMAC) method.	This value is used to instantiate the use of the Initial DUKPT key on the receiving device and it identifies the Initial Key derived from a BDK. This field typically contains the derivation data + a zeroized counter value. Use IK for AES DUKPT.
PB	Padding block	random printable ASCII characters	The service automatically populates this field on export to ensure optional headers are multiples of encryption block length

Export asymmetric (RSA) keys

To export a public key in certificate form, use the **get-public-key-certificate** command. This command returns:

- The certificate
- The root certificate

Both certificates are in base64 encoding.

Note

This operation is not idempotent—subsequent calls might generate different certificates even when using the same underlying key.

Example

```
$ aws payment-cryptography get-public-key-certificate \
     --key-identifier arn:aws:payment-cryptography:us-
east-2:111122223333:key/5dza7xqd6soanjtb
```

```
{
"KeyCertificate": "LS0tLS1CRUdJTi...",
"KeyCertificateChain": "LS0tLS1CRUdJT..."
}
```

Advanced Topics

This section covers advanced key exchange scenarios and configurations.

Topics

- Bring Your Own Certificate Authority (BYOCA)

Bring Your Own Certificate Authority (BYOCA)

By default, when a public key certificate is needed for asymmetric(RSA,ECC) keys created within the service, these certificates are issued by a AWS Payment Cryptography and account-unique certificate authority(CA). This is intended to make it simple to use X.509 without the burden of identifying or setting up a CA or managing Certificate Signing Requests(CSR).

AWS Payment Cryptography also provides the ability to use your own CA when it is required for policy or compliance reasons.

Overview

The BYOCA feature allows you to use your own Certificate Authority anywhere that certificates are used, including TR-34 import/export, RSA Unwrap, and ECDH-based key transfers. This is useful when you need to maintain a consistent certificate chain across your organization or when working with partners who require specific CA certificates. The following example demonstrates the BYOCA workflow using TR-34 key export.

The three key differences compared to the standard TR-34 export flow are:

1. The signing RSA key is explicitly created using CreateKey. Previously, it was implicitly created via GetParametersForExport.
2. A new API GetCertificateSigningRequest creates a Certificate Signing Request (CSR) that can be signed by your external CA.
3. The ExportKey API is extended to allow a certificate to be provided at runtime. Previously, this was implicitly provided by `import-token`, which becomes an optional field.

> **Important Considerations**
>
> - These examples use RSA-2048 keys and wrap a TDES-2KEY key. When exporting AES-128, make sure that all keys are RSA-3072 or RSA-4096.
> - The most common error is that the key represented by `SigningKeyIdentifier` and `SigningKeyCertificate` do not match.

BYOCA Workflow

The following steps demonstrate the complete BYOCA workflow for TR-34 export.

Steps

- Step 1: Create RSA Key
- Step 2: Generate Certificate Signing Request
- Step 3: Review CSR (Optional)
- Step 4: Sign the CSR with a Certificate Authority
- Step 5: Import CA Certificate
- Step 6: Get KRD Encryption Certificate
- Step 7: Export Key with BYOCA

Step 1: Create RSA Key

First, create an RSA Key Pair that will ultimately be the KDH Signing Certificate. You can add tags to identify the key's purpose.

Example Create RSA Key for Signing

```
$ aws payment-cryptography create-key --exportable \
    --key-attributes
 KeyAlgorithm=RSA_2048,KeyUsage=TR31_S0_ASYMMETRIC_KEY_FOR_DIGITAL_SIGNATURE,KeyClass=ASYMMETRI
```

```
{
    "Key": {
        "KeyArn": "arn:aws:payment-cryptography:us-east-1:111122223333:key/
xgmq6fs6uow736uc",
        "KeyAttributes": {
            "KeyUsage": "TR31_S0_ASYMMETRIC_KEY_FOR_DIGITAL_SIGNATURE",
            "KeyClass": "ASYMMETRIC_KEY_PAIR",
            "KeyAlgorithm": "RSA_2048",
            "KeyModesOfUse": {
                "Sign": true
            }
        },
        "KeyCheckValue": "41E3723C",
        "KeyCheckValueAlgorithm": "SHA_1",
        "Enabled": true,
        "Exportable": true,
        "KeyState": "CREATE_COMPLETE",
        "KeyOrigin": "AWS_PAYMENT_CRYPTOGRAPHY"
    }
```

```
}
```

Take note of the `KeyArn` as you'll need it in the next step.

Step 2: Generate Certificate Signing Request

Generate a Certificate Signing Request (CSR) to be signed by your external CA using the GetCertificateSigningRequest API. The output is a base64-encoded PEM file. If you base64 decode the contents and save them, you will have a valid CSR in PEM format.

Example Generate CSR

```
$ aws payment-cryptography-data get-certificate-signing-request \
    --key-identifier arn:aws:payment-cryptography:us-east-1:111122223333:key/
xgmq6fs6uow736uc \
    --signing-algorithm SHA512 \
    --certificate-subject '{
        "CommonName": "MyCertificateAWSUSEAST",
        "Organization": "Amazon",
        "OrganizationUnit": "PaymentCryptography",
        "Country": "US",
        "StateOrProvince": "Virginia",
        "City": "Arlington"
    }'
```

```
{
    "CertificateSigningRequest": "LS0tLS1CRUdJTiBDRVJUSUZJQ0FURSBSRVFVRVNULS0tLS0..."
}
```

The `CertificateSigningRequest` field contains the base64-encoded CSR that you'll send to your CA for signing.

Step 3: Review CSR (Optional)

You can optionally use OpenSSL to review the CSR contents and ensure it's valid and as expected.

Example Review CSR with OpenSSL

```
$ echo "LS0tLS1CRUdJTiBDRVJUSUZJQ0FURSBSRVFVRVNULS0tLS0..." | base64 -d | openssl req -
text
```

Step 4: Sign the CSR with a Certificate Authority

After generating the CSR, you need to have it signed by a Certificate Authority (CA). In production environments, you would typically use AWS Private CA or your organization's established CA infrastructure. For testing purposes, you can use OpenSSL to create a self-signed certificate.

Using AWS Private CA

To sign the CSR using AWS Private CA, first decode the base64-encoded CSR and save it to a file, then use the IssueCertificate API.

Example Sign CSR with AWS Private CA

```
$ echo "LS0tLS1CRUdJTiBDRVJUSUZJQ0FURSBSRVFVRVNULS0tLS0..." | base64 -d > csr.pem

$ aws acm-pca issue-certificate \
    --certificate-authority-arn arn:aws:acm-pca:us-east-1:111122223333:certificate-
authority/12345678-1234-1234-1234-123456789012 \
    --csr fileb://csr.pem \
    --signing-algorithm SHA256WITHRSA \
    --validity Value=365,Type=DAYS
```

```
{
    "CertificateArn": "arn:aws:acm-pca:us-east-1:111122223333:certificate-
authority/12345678-1234-1234-1234-123456789012/certificate/abcdef1234567890"
}
```

Then retrieve the signed certificate:

Example Retrieve Signed Certificate

```
$ aws acm-pca get-certificate \
    --certificate-authority-arn arn:aws:acm-pca:us-east-1:111122223333:certificate-
authority/12345678-1234-1234-1234-123456789012 \
    --certificate-arn arn:aws:acm-pca:us-east-1:111122223333:certificate-
authority/12345678-1234-1234-1234-123456789012/certificate/abcdef1234567890
```

```
{
    "Certificate": "-----BEGIN CERTIFICATE-----\nMIID...\n-----END CERTIFICATE-----",
    "CertificateChain": "-----BEGIN CERTIFICATE-----\nMIID...\n-----END
 CERTIFICATE-----"
```

```
}
```

Save the certificate content for use in the export step. You'll need to base64-encode it when providing it to the `ExportKey` API.

Using OpenSSL for Testing

For testing purposes, you can use OpenSSL to create a self-signed CA and sign the CSR. First, create a CA private key and self-signed certificate:

Example Create Test CA with OpenSSL

```
$ # Generate CA private key
openssl genrsa -out ca-key.pem 4096

$ # Create self-signed CA certificate
openssl req -new -x509 -days 3650 -key ca-key.pem -out ca-cert.pem \
    -subj "/C=US/ST=Virginia/L=Arlington/O=TestOrg/CN=Test CA"
```

Then decode the CSR from the previous step and sign it with your test CA:

Example Sign CSR with OpenSSL

```
$ # Decode the base64-encoded CSR
echo "LS0tLS1CRUdJTiBDRVJUSUZJQ0FURSBSRVFVRVNULS0tLS0..." | base64 -d > csr.pem

$ # Sign the CSR with the CA
openssl x509 -req -in csr.pem -CA ca-cert.pem -CAkey ca-key.pem \
    -CAcreateserial -out signed-cert.pem -days 365 -sha512
```

```
Certificate request self-signature ok
subject=C=US, ST=Virginia, L=Arlington, O=Amazon, OU=PaymentCryptography,
 CN=MyCertificateAWSUSEAST
```

The signed certificate is now in `signed-cert.pem`. You'll need to base64-encode this certificate when providing it to the `ExportKey` API:

Example Base64 Encode the Signed Certificate

```
$ cat signed-cert.pem | base64 -w 0
```

Step 5: Import CA Certificate

Any CA being used needs to be trusted first to prevent arbitrary certificates from being used. Import your external CA's root certificate using the ImportKey API. If using an intermediate CA, call `import-key` again but specify `TrustedPublicKey` instead of `RootCertificatePublicKey` and specify the root CA ARN.

Example Import Root CA Certificate

```
$ aws payment-cryptography import-key --key-material='{
    "RootCertificatePublicKey": {
        "KeyAttributes": {
            "KeyAlgorithm": "RSA_4096",
            "KeyClass": "PUBLIC_KEY",
            "KeyModesOfUse": {
                "Verify": true
            },
            "KeyUsage": "TR31_S0_ASYMMETRIC_KEY_FOR_DIGITAL_SIGNATURE"
        },
        "PublicKeyCertificate": "LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0t..."
    }
}'
```

```
{
    "Key": {
        "KeyArn": "arn:aws:payment-cryptography:us-east-1:111122223333:key/
xivpaqy7qbbm7cdw",
        "KeyAttributes": {
            "KeyUsage": "TR31_S0_ASYMMETRIC_KEY_FOR_DIGITAL_SIGNATURE",
            "KeyClass": "PUBLIC_KEY",
            "KeyAlgorithm": "RSA_4096",
            "KeyModesOfUse": {
                "Verify": true
            }
        },
        "Enabled": true,
        "KeyState": "CREATE_COMPLETE",
        "KeyOrigin": "EXTERNAL"
    }
}
```

Take note of the CA's `KeyArn` for use in the export step.

Step 6: Get KRD Encryption Certificate

In this example, we're importing back into AWS Payment Cryptography, so we call the service to receive a KRD public key certificate using the GetParametersForImport API. In a real scenario, this would be provided by the other system, like an HSM, an ATM, a payment terminal or payment terminal management system.

Example Get Parameters for Import

```
$ aws payment-cryptography-data get-parameters-for-import \
    --key-material-type "TR34_KEY_BLOCK" \
    --wrapping-key-algorithm RSA_2048
```

```
{
    "WrappingKeyCertificate": "LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0t...",
    "WrappingKeyCertificateChain": "LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0t...",
    "WrappingKeyAlgorithm": "RSA_2048",
    "ImportToken": "import-token-v2rxpl6drxeptn7w",
    "ParametersValidUntilTimestamp": "2025-11-01T18:45:31.271000-07:00"
}
```

Step 7: Export Key with BYOCA

Finally, export the key using TR-34 with your own CA-signed certificate using the ExportKey API. Provide the signing certificate that was signed by your external CA.

Example TR-34 Export with BYOCA

```
$ aws payment-cryptography-data export-key \
    --export-key-identifier arn:aws:payment-cryptography:us-east-1:111122223333:key/
iox73p5f4c4yjiod \
    --key-material '{
        "Tr34KeyBlock": {
            "CertificateAuthorityPublicKeyIdentifier": "arn:aws:payment-
cryptography:us-east-1:111122223333:key/j625deyfqlwctu57",
            "SigningKeyIdentifier": "arn:aws:payment-cryptography:us-
east-1:111122223333:key/xgmq6fs6uow736uc",
            "SigningKeyCertificate": "LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0t...",
            "KeyBlockFormat": "X9_TR34_2012",
            "WrappingKeyCertificate": "LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0t..."
        }
```

```
    }'
```

```
{
    "WrappedKey": {
        "WrappedKeyMaterialFormat": "TR34_KEY_BLOCK",
        "KeyMaterial": "3082055A06092A864886F70D010702A082054B30820547...",
        "KeyCheckValue": "3DCA31",
        "KeyCheckValueAlgorithm": "ANSI_X9_24"
    }
}
```

The exported key block can now be imported by the receiving system using the standard TR-34 import process.

Additional Notes

- These examples are shown using the AWS CLI. The same functionality is available in all AWS SDKs including Java, Python, Go, and Rust.
- If you're testing with a self-signed CA, you can use OpenSSL to create a test CA and sign the CSR. In production, use your organization's established CA infrastructure.

Using aliases

An *alias* is a friendly name for an AWS Payment Cryptography key. For example, an alias lets you refer to a key as `alias/test-key` instead of `arn:aws:payment-cryptography:us-east-2:111122223333:key/kwapwa6qaifllw2h`.

You can use an alias to identify a key in most key management (control plane) operations, and in cryptographic (data plane) operations.

You can also allow and deny access to AWS Payment Cryptography key based on their aliases without editing policies or managing grants. This feature is part of the service's support for attribute-based access control (ABAC).

Much of the power of aliases comes from your ability to change the key associated with an alias at any time. Aliases can make your code easier to write and maintain. For example, suppose you use an alias to refer to a particular AWS Payment Cryptography key and you want to change the AWS Payment Cryptography key. In that case, just associate the alias with a different key. You don't need to change your code or application configuration.

Aliases also make it easier to reuse the same code in different AWS Regions. Create aliases with the same name in multiple Regions and associate each alias with an AWS Payment Cryptography key in its Region. When the code runs in each Region, the alias refers to the associated AWS Payment Cryptography key in that Region.

You can create an alias for an AWS Payment Cryptography key by using the `CreateAlias` API.

The AWS Payment Cryptography API provides full control of aliases in each account and Region. The API includes operations to create an alias (CreateAlias), view alias names and the linked keyARN (**list-aliases**), change the AWS Payment Cryptography key associated with an alias (**update-alias**), and delete an alias (**delete-alias**).

Topics

- About aliases
- Using aliases in your applications
- Related APIs

About aliases

Learn how aliases work in AWS Payment Cryptography.

An alias is an independent AWS resource

An alias is not a property of an AWS Payment Cryptography key. The actions that you take on the alias don't affect its associated key. You can create an alias for an AWS Payment Cryptography key and then update the alias so it's associated with a different AWS Payment Cryptography key. You can even delete the alias without any effect on the associated AWS Payment Cryptography key. If you delete a AWS Payment Cryptography key, all aliases associated with that key will become unassigned.

If you specify an alias as the resource in an IAM policy, the policy refers to the alias, not to the associated AWS Payment Cryptography key.

Each alias has a friendly name

When you create an alias, you specify the alias name prefixed by `alias/`. For instance `alias/test_1234`

Each alias is associated with one AWS Payment Cryptography key at a time

The alias and its AWS Payment Cryptography key must be in the same account and Region.

An AWS Payment Cryptography key can be associated with more than one alias concurrently, but each alias can only be mapped to a single key

For example, this `list-aliases` output shows that the `alias/sampleAlias1` alias is associated with exactly one target AWS Payment Cryptography key, which is represented by the `KeyArn` property.

```
$ aws payment-cryptography list-aliases
```

```
  {
    "Aliases": [
        {
            "AliasName": "alias/sampleAlias1",
            "KeyArn": "arn:aws:payment-cryptography:us-east-2:111122223333:key/
kwapwa6qaifllw2h"
        }
    ]
}
```

Multiple aliases can be associated with the same AWS Payment Cryptography key

For example, you can associate the `alias/sampleAlias1;` and `alias/sampleAlias2` aliases with the same key.

```
$ aws payment-cryptography list-aliases
```

```
{
        "Aliases": [
            {
                "AliasName": "alias/sampleAlias1",
                "KeyArn": "arn:aws:payment-cryptography:us-east-2:111122223333:key/
kwapwa6qaifllw2h"
            },
            {
                "AliasName": "alias/sampleAlias2",
```

```
                "KeyArn": "arn:aws:payment-cryptography:us-east-2:111122223333:key/
kwapwa6qaifllw2h"
            }
        ]
    }
```

An alias must be unique for a given account and Region

For example, you can have only one `alias/sampleAlias1` alias in each account and Region. Aliases are case-sensitive, but we recommend against using aliases that only differ in capitalization as they can be prone to error. You cannot change an alias name. However, you can delete the alias and create a new alias with the desired name.

You can create an alias with the same name in different Regions

For example, you can have alias `alias/sampleAlias2` in US East (N. Virginia) and alias `alias/sampleAlias2` in US West (Oregon). Each alias would be associated with an AWS Payment Cryptography key in its Region. If your code refers to an alias name like `alias/finance-key`, you can run it in multiple Regions. In each Region, it uses a different alias/sampleAlias2. For details, see Using aliases in your applications.

You can change the AWS Payment Cryptography key associated with an alias

You can use the `UpdateAlias` operation to associate an alias with a different AWS Payment Cryptography key. For example, if the `alias/sampleAlias2` alias is associated with the `arn:aws:payment-cryptography:us-east-2:111122223333:key/kwapwa6qaifllw2h` AWS Payment Cryptography key, you can update it so it is associated with the `arn:aws:payment-cryptography:us-east-2:111122223333:key/tqv5yij6wtxx64pi` key.

> **Warning**
>
> AWS Payment Cryptography doesn't validate that the old and new keys have all the same attributes such as key usage. Updating with a different key type may result in problems in your application.

Some keys don't have aliases

An alias is an optional feature and not all keys will have aliases unless you choose to operate your environment in this way. Keys can be associated with Aliases using the `create-`

`alias` command. Also, you can use the update-alias operation to change the AWS Payment Cryptography key associated with an alias and the delete-alias operation to delete an alias. As a result, some AWS Payment Cryptography keys might have several aliases, and some might have none.

Mapping a key to an alias

You can map a key (represented by an ARN) to one or more aliases using the `create-alias` command. This command is not idempotent - to update an alias, use the update-alias command.

```
$ aws payment-cryptography create-alias --alias-name alias/sampleAlias1 \
             --key-arn arn:aws:payment-cryptography:us-east-2:111122223333:key/
kwapwa6qaifllw2h
```

```
{
    "Alias": {
        "AliasName": "alias/alias/sampleAlias1",
        "KeyArn": "arn:aws:payment-cryptography:us-east-2:111122223333:key/
kwapwa6qaifllw2h"
    }
}
```

Using aliases in your applications

You can use an alias to represent an AWS Payment Cryptography key in your application code. The `key-identifier` parameter in AWS Payment Cryptography data operations as well as other operations like List Keys accepts an alias name or alias ARN.

```
$ aws payment-cryptography-data generate-card-validation-data --key-identifier alias/
BIN_123456_CVK --primary-account-number=171234567890123 --generation-attributes
 CardVerificationValue2={CardExpiryDate=0123}
```

When using an alias ARN, remember that the alias mapping to an AWS Payment Cryptography key is defined in the account that owns the AWS Payment Cryptography key and might differ in each Region.

One of the most powerful uses of aliases is in applications that run in multiple AWS Regions.

You could create a different version of your application in each Region or use a dictionary, configuration or switch statement to select the right AWS Payment Cryptography key for each Region. But it might be easier to create an alias with the same alias name in each Region. Remember that the alias name is case-sensitive.

Related APIs

Tags

Tags are key and value pairs that act as metadata for organizing your AWS Payment Cryptography keys. They can be used to flexibly identify keys or group one or more keys together.

Get keys

An AWS Payment Cryptography key represents a single unit of cryptographic material and can only be used for cryptographic operations for this service. The GetKeys API takes a KeyIdentifier as input and returns key metadata including attributes, state, and timestamps, but does not return actual cryptographic key material.

Example

```
$ aws payment-cryptography get-key --key-identifier arn:aws:payment-cryptography:us-east-2:111122223333:key/kwapwa6qaifllw2h
```


```
{
  "Key": {
      "KeyArn": "arn:aws:payment-cryptography:us-east-2:111122223333:key/kwapwa6qaifllw2h",
      "KeyAttributes": {
          "KeyUsage": "TR31_D0_SYMMETRIC_DATA_ENCRYPTION_KEY",
          "KeyClass": "SYMMETRIC_KEY",
          "KeyAlgorithm": "AES_128",
          "KeyModesOfUse": {
              "Encrypt": true,
              "Decrypt": true,
              "Wrap": true,
              "Unwrap": true,
              "Generate": false,
              "Sign": false,
              "Verify": false,
              "DeriveKey": false,
              "NoRestrictions": false
          }
      },
      "KeyCheckValue": "0A3674",
      "KeyCheckValueAlgorithm": "CMAC",
      "Enabled": true,
      "Exportable": true,
      "KeyState": "CREATE_COMPLETE",
      "KeyOrigin": "AWS_PAYMENT_CRYPTOGRAPHY",
      "CreateTimestamp": "2023-06-02T07:38:14.913000-07:00",
      "UsageStartTimestamp": "2023-06-02T07:38:14.857000-07:00"
  }
}
```

Get the public key/certificate associated with a key pair

Get Public Key/Certificate returns the public key indicated by the `KeyArn`. This can be the public key portion of a key pair generated on AWS Payment Cryptography or a previously imported public key. The most common use case is to provide the public key to an outside service that will encrypt data. That data can then be passed to an application leveraging AWS Payment Cryptography and the data can be decrypted using the private key secured within AWS Payment Cryptography.

The service returns public keys as a public certificate. The API result contains the CA and the public key certificate. Both data elements are base64 encoded.

Note

The public certificate returned is intended to be short lived and is not intended to be idempotent. You may receive a different certificate on each API call even the public key itself is unchanged.

Example

```
$ aws payment-cryptography get-public-key-certificate  --key-identifier
 arn:aws:payment-cryptography:us-east-2:111122223333:key/nsq2i3mbg6sn775f
```


```
{
  "KeyCertificate":
 "LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0tCk1JSUV2VENDQXFXZ0F3SUJBZ0lSQUo1OWd2VkpDd3dlYldMNldYZEpYY
  "KeyCertificateChain":
 "LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0tCk1JSUY0VENDQThtZ0F3SUJBZ0lSQUtlN2piaHFKZjJPd3FGUWI5c3VuC
}
```

Tagging keys

In AWS Payment Cryptography, you can add tags to a AWS Payment Cryptography key when you create a key, and tag or untag existing keys unless they are pending deletion. Tags are optional, but they can be very useful.

For general information about tags, including best practices, tagging strategies, and the format and syntax of tags, see Tagging AWS resources in the *Amazon Web Services General Reference*.

Topics

- About tags in AWS Payment Cryptography
- Viewing key tags in the console
- Managing key tags with API operations
- Controlling access to tags
- Using tags to control access to keys

About tags in AWS Payment Cryptography

A *tag* is an optional metadata label that you can assign (or AWS can assign) to an AWS resource. Each tag consists of a *tag key* and a *tag value*, both of which are case-sensitive strings. The tag value can be an empty (null) string. Each tag on a resource must have a different tag key, but you can add the same tag to multiple AWS resources. Each resource can have up to 50 user-created tags.

Do not include confidential or sensitive information in the tag key or tag value. Tags are accessible to many AWS services, including billing.

In AWS Payment Cryptography, you can add tags to a key when you create the key, and tag or untag existing keys unless they are pending deletion. You cannot tag aliases. Tags are optional, but they can be very useful.

For example, you can add a `"Project"="Alpha"` tag to all AWS Payment Cryptography keys and Amazon S3 buckets that you use for the Alpha project. Another example is to add `"BIN"="20130622"` tag to all keys associated to a specific bank identification number(BIN).

```
    [
    {
      "Key": "Project",
      "Value": "Alpha"
    },
    {
      "Key": "BIN",
      "Value": "20130622"
    }
  ]
```

For general information about tags, including the format and syntax, see Tagging AWS resources in the *Amazon Web Services General Reference*.

Tags help you do the following:

- Identify and organize your AWS resources. Many AWS services support tagging, so you can assign the same tag to resources from different services to indicate that the resources are related. For example, you can assign the same tag to an AWS Payment Cryptography keys and an Amazon Elastic Block Store (Amazon EBS) volume or AWS Secrets Manager secret. You can also use tags to identify keys for automation.
- Track your AWS costs. When you add tags to your AWS resources, AWS generates a cost allocation report with usage and costs aggregated by tags. You can use this feature to track AWS Payment Cryptography costs for a project, application, or cost center.

  For more information about using tags for cost allocation, see Using Cost Allocation Tags in the *AWS Billing User Guide*. For information about the rules for tag keys and tag values, see User-Defined Tag Restrictions in the *AWS Billing User Guide*.
- Control access to your AWS resources. Allowing and denying access to keys based on their tags is part of AWS Payment Cryptography support for attribute-based access control (ABAC). For information about controlling access to AWS Payment Cryptography based on their tags, see Authorization based on AWS Payment Cryptography tags. For more general information about using tags to control access to AWS resources, see Controlling Access to AWS Resources Using Resource Tags in the *IAM User Guide*.

AWS Payment Cryptography writes an entry to your AWS CloudTrail log when you use the TagResource, UntagResource, or ListTagsForResource operations.

Viewing key tags in the console

To view tags in the console, you need tagging permission on the key from an IAM policy that includes the key. You need these permissions in addition to the permissions to view keys in the console.

Managing key tags with API operations

You can use the AWS Payment Cryptography API to add, delete, and list tags for the keys that you manage. These examples use the AWS Command Line Interface (AWS CLI), but you can use any supported programming language. You cannot tag AWS managed keys.

To add, edit, view, and delete tags for a key, you must have the required permissions. For details, see Controlling access to tags.

Topics

- CreateKey: Add tags to a new key
- TagResource: Add or change tags for a key
- ListResourceTags: Get the tags for a key
- UntagResource: Delete tags from a key

CreateKey: Add tags to a new key

You can add tags when you create a key. To specify the tags, use the `Tags` parameter of the CreateKey operation.

To add tags when creating a key, the caller must have `payment-cryptography:TagResource` permission in an IAM policy. At a minimum, the permission must cover all keys in the account and Region. For details, see Controlling access to tags.

The value of the `Tags` parameter of `CreateKey` is a collection of case-sensitive tag key and tag value pairs. Each tag on a key must have a different tag name. The tag value can be a null or empty string.

For example, the following AWS CLI command creates a symmetric encryption key with a `Project:Alpha` tag. When specifying more than one key-value pair, use a space to separate each pair.

```
$ aws payment-cryptography create-key --exportable --key-attributes
 KeyAlgorithm=TDES_2KEY, \
        KeyUsage=TR31_C0_CARD_VERIFICATION_KEY,KeyClass=SYMMETRIC_KEY, \
        KeyModesOfUse='{Generate=true,Verify=true}' \
        --tags '[{"Key":"Project","Value":"Alpha"},{"Key":"BIN","Value":"123456"}]'
```

When this command is successful, it returns a `Key` object with information about the new key. However, the `Key` does not include tags. To get the tags, use the ListResourceTags operation.

TagResource: Add or change tags for a key

The TagResource operation adds one or more tags to a key. You cannot use this operation to add or edit tags in a different AWS account.

To add a tag, specify a new tag key and a tag value. To edit a tag, specify an existing tag key and a new tag value. Each tag on a key must have a different tag key. The tag value can be a null or empty string.

For example, the following command adds **`UseCase`** and **`BIN`** tags to an example key.

```
$ aws payment-cryptography tag-resource --resource-arn arn:aws:payment-
cryptography:us-east-2:111122223333:key/kwapwa6qaifllw2h --tags
 '[{"Key":"UseCase","Value":"Acquiring"},{"Key":"BIN","Value":"123456"}]'
```

When this command is successful, it does not return any output. To view the tags on a key, use the ListResourceTags operation.

You can also use **TagResource** to change the tag value of an existing tag. To replace a tag value, specify the same tag key with a different value. Tags not listed in a modify command are not changed or removed.

For example, this command changes the value of the `Project` tag from `Alpha` to `Noe`.

The command will return http/200 with no content. To see your changes, use `ListTagsForResource`

```
$ aws payment-cryptography tag-resource --resource-arn arn:aws:payment-cryptography:us-
east-2:111122223333:key/kwapwa6qaifllw2h \
        --tags '[{"Key":"Project","Value":"Noe"}]'
```

ListResourceTags: Get the tags for a key

The ListResourceTags operation gets the tags for a key. The `ResourceArn` (keyArn or keyAlias) parameter is required. You cannot use this operation to view the tags on keys in a different AWS account.

For example, the following command gets the tags for an example key.

```
$ aws payment-cryptography list-tags-for-resource --resource-arn arn:aws:payment-cryptography:us-east-2:111122223333:key/kwapwa6qaifllw2h

  {
    "Tags": [
        {
            "Key": "BIN",
            "Value": "20151120"
        },
        {
            "Key": "Project",
            "Value": "Production"
        }
    ]
}
```

UntagResource: Delete tags from a key

The UntagResource operation deletes tags from a key. To identify the tags to delete, specify the tag keys. You cannot use this operation to delete tags from keys a different AWS account.

When it succeeds, the `UntagResource` operation doesn't return any output. Also, if the specified tag key isn't found on the key, it doesn't throw an exception or return a response. To confirm that the operation worked, use the ListResourceTags operation.

For example, this command deletes the **`Purpose`** tag and its value from the specified key.

```
$ aws payment-cryptography untag-resource \
        --resource-arn arn:aws:payment-cryptography:us-east-2:111122223333:key/
kwapwa6qaifllw2h --tag-keys Project
```

Controlling access to tags

To add, view, and delete tags by using the API, principals need tagging permissions in IAM policies.

You can also limit these permissions by using AWS global condition keys for tags. In AWS Payment Cryptography, these conditions can control access to tagging operations, such as TagResource and UntagResource.

For example policies and more information, see Controlling Access Based on Tag Keys in the *IAM User Guide*.

Permissions to create and manage tags work as follows.

payment-cryptography:TagResource

Allows principals to add or edit tags. To add tags while creating a key, the principal must have permission in an IAM policy that isn't restricted to particular keys.

payment-cryptography:ListTagsForResource

Allows principals to view tags on keys.

payment-cryptography:UntagResource

Allows principals to delete tags from keys.

Tag permissions in policies

You can provide tagging permissions in a key policy or IAM policy. For example, the following example key policy gives select users tagging permission on the key. It gives all users who can assume the example Administrator or Developer roles permission to view tags.

JSON

```
{
  "Version":"2012-10-17",
  "Id": "example-key-policy",
  "Statement": [
    {
      "Sid": "EnableIAMUserPermissions",
      "Effect": "Allow",
      "Principal": {"AWS": "arn:aws:iam::111122223333:root"},
      "Action": "payment-cryptography:*",
      "Resource": "*"
    },
```

```
    {
      "Sid": "AllowAllTaggingPermissions",
      "Effect": "Allow",
      "Principal": {"AWS": [
        "arn:aws:iam::111122223333:user/LeadAdmin",
        "arn:aws:iam::111122223333:user/SupportLead"
      ]},
      "Action": [
        "payment-cryptography:TagResource",
        "payment-cryptography:ListTagsForResource",
        "payment-cryptography:UntagResource"
      ],
      "Resource": "*"
    },
    {
      "Sid": "Allow roles to view tags",
      "Effect": "Allow",
      "Principal": {
        "AWS": [
          "arn:aws:iam::111122223333:role/Administrator",
          "arn:aws:iam::111122223333:role/Developer"
        ]
      },
      "Action": "payment-cryptography:ListTagsForResource",
      "Resource": "*"
    }
  ]
}
```

To give principals tagging permission on multiple keys, you can use an IAM policy. For this policy to be effective, the key policy for each key must allow the account to use IAM policies to control access to the key.

For example, the following IAM policy allows the principals to create keys. It also allows them to create and manage tags on all keys in the specified account. This combination allows the principals to use the tags parameter of the CreateKey operation to add tags to a key while they are creating it.

JSON

```
{
```

```
  "Version":"2012-10-17",
  "Statement": [
    {
      "Sid": "IAMPolicyCreateKeys",
      "Effect": "Allow",
      "Action": "payment-cryptography:CreateKey",
      "Resource": "*"
    },
    {
      "Sid": "IAMPolicyTags",
      "Effect": "Allow",
      "Action": [
        "payment-cryptography:TagResource",
        "payment-cryptography:UntagResource",
        "payment-cryptography:ListTagsForResource"
      ],
      "Resource": "arn:aws:payment-cryptography:*:111122223333:key/*"
    }
  ]
}
```

Limiting tag permissions

You can limit tagging permissions by using policy conditions. The following policy conditions can be applied to the `payment-cryptography:TagResource` and `payment-cryptography:UntagResource` permissions. For example, you can use the `aws:RequestTag/tag-key` condition to allow a principal to add only particular tags, or prevent a principal from adding tags with particular tag keys.

- aws:RequestTag
- aws:ResourceTag/*tag-key* (IAM policies only)
- aws:TagKeys

As a best practice when you use tags to control access to keys, use the `aws:RequestTag/tag-key` or `aws:TagKeys` condition key to determine which tags (or tag keys) are allowed.

For example, the following IAM policy is similar to the previous one. However, this policy allows the principals to create tags (`TagResource`) and delete tags `UntagResource` only for tags with a `Project` tag key.

Because `TagResource` and `UntagResource` requests can include multiple tags, you must specify a `ForAllValues` or `ForAnyValue` set operator with the aws:TagKeys condition. The `ForAnyValue` operator requires that at least one of the tag keys in the request matches one of the tag keys in the policy. The `ForAllValues` operator requires that all of the tag keys in the request match one of the tag keys in the policy. The `ForAllValues` operator also returns `true` if there are no tags in the request, but TagResource and UntagResource fail when no tags are specified. For details about the set operators, see Use multiple keys and values in the *IAM User Guide*.

JSON

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Sid": "IAMPolicyCreateKey",
      "Effect": "Allow",
      "Action": "payment-cryptography:CreateKey",
      "Resource": "*"
    },
    {
      "Sid": "IAMPolicyViewAllTags",
      "Effect": "Allow",
      "Action": "payment-cryptography:ListTagsForResource",
      "Resource": "arn:aws:payment-cryptography:*:111122223333:key/*"
    },
    {
      "Sid": "IAMPolicyManageTags",
      "Effect": "Allow",
      "Action": [
        "payment-cryptography:TagResource",
        "payment-cryptography:UntagResource"
      ],
      "Resource": "arn:aws:payment-cryptography:*:111122223333:key/*",
      "Condition": {
          "ForAllValues:StringEquals": {"aws:TagKeys": "Project"}
      }
    }
  ]
}
```

Using tags to control access to keys

You can control access to AWS Payment Cryptography based on the tags on the key. For example, you can write an IAM policy that allows principals to enable and disable only the keys that have a particular tag. Or you can use an IAM policy to prevent principals from using keys in cryptographic operations unless the key has a particular tag.

This feature is part of AWS Payment Cryptography support for attribute-based access control(ABAC). For information about using tags to control access to AWS resources, see What is ABAC for AWS? and Controlling Access to AWS Resources Using Resource Tags in the *IAM User Guide*.

AWS Payment Cryptography supports the aws:ResourceTag/*tag-key* global condition context key, which lets you control access to keys based on the tags on the key. Because multiple keys can have the same tag, this feature lets you apply the permission to a select set of keys. You can also easily change the keys in the set by changing their tags.

In AWS Payment Cryptography, the `aws:ResourceTag/tag-key` condition key is supported only in IAM policies. It isn't supported in key policies, which apply only to one key, or on operations that don't use a particular key, such as the ListKeys or ListAliases operations.

Controlling access with tags provides a simple, scalable, and flexible way to manage permissions. However, if not properly designed and managed, it can allow or deny access to your keys inadvertently. If you are using tags to control access, consider the following practices.

- Use tags to reinforce the best practice of least privileged access. Give IAM principals only the permissions they need on only the keys they must use or manage. For example, use tags to label the keys used for a project. Then give the project team permission to use only keys with the project tag.
- Be cautious about giving principals the `payment-cryptography:TagResource` and `payment-cryptography:UntagResource` permissions that let them add, edit, and delete tags. When you use tags to control access to keys, changing a tag can give principals permission to use keys that they didn't otherwise have permission to use. It can also deny access to keys that other principals require to do their jobs. Key administrators who don't have permission to change key policies or create grants can control access to keys if they have permission to manage tags.

  Whenever possible, use a policy condition, such as `aws:RequestTag/tag-key` or `aws:TagKeys` to limit a principal's tagging permissions to particular tags or tag patterns on particular keys.

- Review the principals in your AWS account that currently have tagging and untagging permissions and adjust them, if necessary. IAM policies might allow tag and untag permissions on all keys. For example, the *Admin* managed policy allows principals to tag, untag, and list tags on all keys.
- Before setting a policy that depends on a tag, review the tags on the keys in your AWS account. Make sure that your policy applies only to the tags you intend to include. Use CloudTrail logs and CloudWatch alarms to alert you to tag changes that might affect access to your keys.
- The tag-based policy conditions use pattern matching; they aren't tied to a particular instance of a tag. A policy that uses tag-based condition keys affects all new and existing tags that match the pattern. If you delete and recreate a tag that matches a policy condition, the condition applies to the new tag, just as it did to the old one.

For example, consider the following IAM policy. It allows the principals to call the Decrypt operations only on keys in your account that are the US East (N. Virginia) Region and have a `"Project"="Alpha"` tag. You might attach this policy to roles in the example Alpha project.

JSON

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Sid": "IAMPolicyWithResourceTag",
      "Effect": "Allow",
      "Action": [
        "payment-cryptography:DecryptData"
      ],
      "Resource": "arn:aws:payment-cryptography:us-east-1:111122223333:key/*",
      "Condition": {
        "StringEquals": {
          "aws:ResourceTag/Project": "Alpha"
        }
      }
    }
  ]
}
```

The following example IAM policy allows the principals to use any key in the account for certain cryptographic operations. But it prohibits the principals from using these cryptographic operations on keys with a `"Type"="Reserved"` tag or no `"Type"` tag.

JSON

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Sid": "IAMAllowCryptographicOperations",
      "Effect": "Allow",
      "Action": [
        "payment-cryptography:EncryptData",
        "payment-cryptography:DecryptData",
        "payment-cryptography:ReEncrypt*"
      ],
      "Resource": "arn:aws:payment-cryptography:*:111122223333:key/*"
    },
    {
      "Sid": "IAMDenyOnTag",
      "Effect": "Deny",
      "Action": [
        "payment-cryptography:EncryptData",
        "payment-cryptography:DecryptData",
        "payment-cryptography:ReEncrypt*"
      ],
      "Resource": "arn:aws:payment-cryptography:*:111122223333:key/*",
      "Condition": {
        "StringEquals": {
          "aws:ResourceTag/Type": "Reserved"
        }
      }
    },
    {
      "Sid": "IAMDenyNoTag",
      "Effect": "Deny",
      "Action": [
       "payment-cryptography:EncryptData",
       "payment-cryptography:DecryptData",
       "payment-cryptography:ReEncrypt*"
      ],
```

```
        "Resource": "arn:aws:kms:*:111122223333:key/*",
        "Condition": {
          "Null": {
            "aws:ResourceTag/Type": "true"
          }
        }
      }
    ]
}
```

Understanding key attributes for AWS Payment Cryptography key

A tenet of proper key management is that keys are appropriately scoped and can only be used for permitted operations. As such, certain keys can only be created with certain key modes of use. Whenever possible, this aligns with the available modes of use as defined by TR-31.

Although AWS Payment Cryptography will prevent you from creating invalid keys, valid combinations are provided here for your convenience.

Symmetric Keys

- TR31_B0_BASE_DERIVATION_KEY
  - **Allowed Key Algorithms**: TDES_2KEY ,TDES_3KEY ,AES_128 ,AES_192 ,AES_256
  - **Allowed combination of key modes of use**: { DeriveKey = true },{ NoRestrictions = true }
- TR31_C0_CARD_VERIFICATION_KEY
  - **Allowed Key Algorithms**: TDES_2KEY ,TDES_3KEY ,AES_128* ,AES_192* ,AES_256*
  - **Allowed combination of key modes of use**: { Generate = true } ,{ Verify = true } ,{ Generate = true, Verify= true } ,{ NoRestrictions = true }
- TR31_D0_SYMMETRIC_DATA_ENCRYPTION_KEY
  - **Allowed Key Algorithms**: TDES_2KEY ,TDES_3KEY ,AES_128 ,AES_192 ,AES_256
  - **Allowed combination of key modes of use**: { Encrypt = true, Decrypt = true, Wrap = true, Unwrap = true } , { Encrypt = true, Wrap = true } ,{ Decrypt = true, Unwrap = true } , { NoRestrictions = true }
- TR31_E0_EMV_MKEY_APP_CRYPTOGRAMS
  - **Allowed Key Algorithms**: TDES_2KEY ,TDES_3KEY*, AES_128* ,AES_192* ,AES_256*

  - **Allowed combination of key modes of use**: { DeriveKey = true }, { NoRestrictions = true }
- TR31_E1_EMV_MKEY_CONFIDENTIALITY
  - **Allowed Key Algorithms**: TDES_2KEY ,TDES_3KEY, AES_128*,AES_192*,AES_256*
  - **Allowed combination of key modes of use**: { DeriveKey = true }, { NoRestrictions = true }
- TR31_E2_EMV_MKEY_INTEGRITY
  - **Allowed Key Algorithms**: TDES_2KEY ,TDES_3KEY ,AES_128* ,AES_192* ,AES_256*
  - **Allowed combination of key modes of use**: { DeriveKey = true }, { NoRestrictions = true }
- TR31_E4_EMV_MKEY_DYNAMIC_NUMBERS
  - **Allowed Key Algorithms**: TDES_2KEY ,TDES_3KEY ,AES_128* ,AES_192* ,AES_256*
  - **Allowed combination of key modes of use**: { DeriveKey = true }, { NoRestrictions = true }
- TR31_E5_EMV_MKEY_CARD_PERSONALIZATION
  - **Allowed Key Algorithms**: TDES_2KEY ,TDES_3KEY ,AES_128* ,AES_192* ,AES_256*
  - **Allowed combination of key modes of use**: { DeriveKey = true }, { NoRestrictions = true }
- TR31_E6_EMV_MKEY_OTHER
  - **Allowed Key Algorithms**: TDES_2KEY ,TDES_3KEY ,AES_128* ,AES_192* ,AES_256*
  - **Allowed combination of key modes of use**: { DeriveKey = true }, { NoRestrictions = true }
- TR31_K0_KEY_ENCRYPTION_KEY
  - Recommended to use TR31_K1_KEY_BLOCK_PROTECTION_KEY. **Allowed Key Algorithms**: TDES_2KEY ,TDES_3KEY ,AES_128 ,AES_192 ,AES_256
  - **Allowed combination of key modes of use**: { Encrypt = true, Decrypt = true, Wrap = true, Unwrap = true } , { Encrypt = true, Wrap = true } ,{ Decrypt = true, Unwrap = true } , { NoRestrictions = true }
- TR31_K1_KEY_BLOCK_PROTECTION_KEY
  - **Allowed Key Algorithms**: TDES_2KEY ,TDES_3KEY ,AES_128 ,AES_192 ,AES_256
  - **Allowed combination of key modes of use**: { Encrypt = true, Decrypt = true, Wrap = true, Unwrap = true } , { Encrypt = true, Wrap = true } ,{ Decrypt = true, Unwrap = true } , { NoRestrictions = true }
- TR31_M1_ISO_9797_1_MAC_KEY
  - **Allowed Key Algorithms**: TDES_2KEY ,TDES_3KEY
  - **Allowed combination of key modes of use**: { Generate = true } ,{ Verify = true } ,{ Generate = true, Verify= true } ,{ NoRestrictions = true }

- TR31_M3_ISO_9797_3_MAC_KEY
  - **Allowed Key Algorithms**: TDES_2KEY ,TDES_3KEY
  - **Allowed combination of key modes of use**: { Generate = true } ,{ Verify = true } ,{ Generate = true, Verify= true } ,{ NoRestrictions = true }
- TR31_M6_ISO_9797_5_CMAC_KEY
  - **Allowed Key Algorithms**: TDES_2KEY ,TDES_3KEY ,AES_128 ,AES_192 ,AES_256
  - **Allowed combination of key modes of use**: { Generate = true } ,{ Verify = true } ,{ Generate = true, Verify= true } ,{ NoRestrictions = true }
- TR31_M7_HMAC_KEY
  - **Allowed Key Algorithms**: TDES_2KEY ,TDES_3KEY ,AES_128 ,AES_192 ,AES_256
  - **Allowed combination of key modes of use**: { Generate = true } ,{ Verify = true } ,{ Generate = true, Verify= true } ,{ NoRestrictions = true }
- TR31_P0_PIN_ENCRYPTION_KEY
  - **Allowed Key Algorithms**: TDES_2KEY ,TDES_3KEY ,AES_128 ,AES_192 ,AES_256
  - **Allowed combination of key modes of use**: { Encrypt = true, Decrypt = true, Wrap = true, Unwrap = true } ,{ Encrypt = true, Wrap = true } ,{ Decrypt = true, Unwrap = true } , { NoRestrictions = true }
- TR31_V1_IBM3624_PIN_VERIFICATION_KEY
  - **Allowed Key Algorithms**: TDES_2KEY ,TDES_3KEY ,AES_128 ,AES_192 ,AES_256
  - **Allowed combination of key modes of use**: { Generate = true } ,{ Verify = true } ,{ Generate = true, Verify= true } ,{ NoRestrictions = true }
- TR31_V2_VISA_PIN_VERIFICATION_KEY
  - **Allowed Key Algorithms**: TDES_2KEY ,TDES_3KEY ,AES_128 ,AES_192 ,AES_256
  - **Allowed combination of key modes of use**: { Generate = true } ,{ Verify = true } ,{ Generate = true, Verify= true } ,{ NoRestrictions = true }

Asymmetric Keys

- TR31_D1_ASYMMETRIC_KEY_FOR_DATA_ENCRYPTION
  - **Allowed Key Algorithms**: RSA_2048 ,RSA_3072 ,RSA_4096
  - **Allowed combination of key modes of use**: { Encrypt = true, Decrypt = true, Wrap = true, Unwrap = true } ,{ Encrypt = true, Wrap = true } ,{ Decrypt = true, Unwrap = true }

  - **NOTE:**: { Encrypt = true, Wrap = true } is the only valid option when importing a public key that is intended for encrypting data or wrapping a key
- TR31_S0_ASYMMETRIC_KEY_FOR_DIGITAL_SIGNATURE
  - **Allowed Key Algorithms**: RSA_2048 ,RSA_3072 ,RSA_4096
  - **Allowed combination of key modes of use**: { Sign = true } ,{ Verify = true }
  - **NOTE:**: { Verify = true } is the only valid option when importing a key meant for signing, such as root certificate, intermediate certificate or signing certificates for TR-34.
- TR31_K3_ASYMMETRIC_KEY_FOR_KEY_AGREEMENT
  - Used for key agreement algorithms such as ECDH
  - **Allowed Key Algorithms**: ECC_NIST_P256,ECC_NIST_P384,ECC_NIST_P521
  - **Allowed combination of key modes of use**: { DeriveKey = true }.
  - **NOTE:**DeriveKeyUsage is used to specify what kind of key will be derived from this base key. This is fixed at key creation/import.
- TR31_K2_TR34_ASYMMETRIC_KEY
  - Asymmetric key used for X9.24 compatible key exchange mechanisms like TR-34
  - **Allowed Key Algorithms**: RSA_2048,RSA_3072,RSA_4096
  - **Allowed combination of key modes of use**: { DeriveKey = true }.
  - **Allowed combination of key modes of use**: { Encrypt = true, Decrypt = true, Wrap = true, Unwrap = true } ,{ Encrypt = true, Wrap = true } ,{ Decrypt = true, Unwrap = true }
  - **NOTE:**: { Encrypt = true, Wrap = true } is the only valid option when importing a public key that is intended for encrypting data or wrapping a key

* This algorithm/key type combination is not currently supported by any cryptographic operations

Data operations

After you have established an AWS Payment Cryptography key, it can be used to perform cryptographic operations. Different operations perform different types of activity ranging from encryption, hashing as well as domain specific algorithms such as CVV2 generation.

Encrypted data cannot be decrypted without the matching decryption key (the symmetric key or private key depending on the encryption type). Hashing and domain specific algorithms similarly cannot be verified without the symmetric key or public key.

For information on valid key types for specific operations please see Valid keys for cryptographic operations

Note

We recommend using test data when in a non-production environment. Using production keys and data (PAN, BDK ID, etc.) in a non-production environment may impact your compliance scope such as for PCI DSS and PCI P2PE.

Topics

- Encrypt, Decrypt and Re-encrypt data
- Generate and verify card data
- Generate, translate and verify PIN data
- Verify auth request (ARQC) cryptogram
- Generate and verify MAC
- Valid keys for cryptographic operations

Encrypt, Decrypt and Re-encrypt data

Encryption and Decryption methods can be used to encrypt or decrypt data using a variety of symmetric and asymmetric techniques including TDES, AES and RSA. These methods also support keys derived using DUKPT and EMV techniques. For use cases where you wish to secure data under a new key without exposing the underlying data, the ReEncrypt command can also be used.

Note

When using the encrypt/decrypt functions, all inputs are assumed to be in hexBinary - for instance a value of 1 will be input as 31 (hex) and a lower case t is represented as 74 (hex). All outputs are in hexBinary as well.

For details on all available options, please consult the API Guide for Encrypt, Decrypt, and Re-Encrypt.

Topics

- Encrypt data
- Decrypt data

Encrypt data

The `Encrypt Data` API is used to encrypt data using symmetric and asymmetric data encryption keys as well as DUKPT and EMV derived keys. Various algorithms and variations are supported including `TDES`, `RSA` and `AES`.

The primary inputs are the encryption key used to encrypt the data, the plaintext data in hexBinary format to be encrypted and encryption attributes such as initialization vector and mode for block ciphers such as TDES. The plaintext data needs to be in multiples of 8 bytes for `TDES`, 16 bytes for `AES` and the length of the key in the case of `RSA`. Symmetric key inputs (TDES, AES, DUKPT, EMV) should be padded in cases where the input data does not meet these requirements. The following table shows the maximum length of plaintext for each type of key and the padding type that you define in `EncryptionAttributes` for RSA keys.

Padding type	RSA_2048	RSA_3072	RSA_4096
`OAEP SHA1`	428	684	940
`OAEP SHA256`	380	636	892
`OAEP SHA512`	252	508	764
`PKCS1`	488	744	1000

Padding type	RSA_2048	RSA_3072	RSA_4096
None	488	744	1000

The primary outputs include the encrypted data as ciphertext in hexBinary format and the checksum value for the encryption key. For details on all available options, please consult the API Guide for Encrypt.

Examples

- Encrypt data using AES symmetric key
- Encrypt data using DUKPT key
- Encrypt data using EMV-derived symmetric key
- Encrypt data using an RSA key

Encrypt data using AES symmetric key

Note

All examples assume the relevant key already exists. Keys can be created using the CreateKey operation or imported using the ImportKey operation.

Example

In this example, we will encrypt plaintext data using a symmetric key which has been created using the CreateKey Operation or imported using the ImportKey Operation. For this operation, the key must have KeyModesOfUse set to `Encrypt` and KeyUsage set to `TR31_D0_SYMMETRIC_DATA_ENCRYPTION_KEY`. Please see Keys for Cryptographic Operations for more options.

```
$ aws payment-cryptography-data encrypt-data --key-identifier arn:aws:payment-cryptography:us-east-2:111122223333:key/tqv5yij6wtxx64pi --plain-text 31323334313233343132333431323334 --encryption-attributes 'Symmetric={Mode=CBC}'
```

```
{
    "KeyArn": "arn:aws:payment-cryptography:us-east-2:111122223333:key/tqv5yij6wtxx64pi",
    "KeyCheckValue": "71D7AE",
    "CipherText": "33612AB9D6929C3A828EB6030082B2BD"
}
```

Encrypt data using DUKPT key

Example

In this example, we will encrypt plaintext data using a DUKPT key. AWS Payment Cryptography supports TDES and AES DUKPT keys. For this operation, the key must have KeyModesOfUse set to `DeriveKey` and KeyUsage set to `TR31_B0_BASE_DERIVATION_KEY`. Please see Keys for Cryptographic Operations for more options.

```
$ aws payment-cryptography-data encrypt-data --key-identifier
 arn:aws:payment-cryptography:us-east-2:111122223333:key/tqv5yij6wtxx64pi
 --plain-text 31323334313233343132333431323334 --encryption-attributes
 'Dukpt={KeySerialNumber=FFFF9876543210E00001}'
```

```
{
    "KeyArn": "arn:aws:payment-cryptography:us-east-2:111122223333:key/
tqv5yij6wtxx64pi",
    "KeyCheckValue": "71D7AE",
    "CipherText": "33612AB9D6929C3A828EB6030082B2BD"
}
```

Encrypt data using EMV-derived symmetric key

Example

In this example, we will encrypt clear text data using an EMV-derived symmetric key which has already been created. You might use a command such as this to send data to an EMV card. For this operation, the key must have KeyModesOfUse set to `Derive` and KeyUsage set to `TR31_E1_EMV_MKEY_CONFIDENTIALITY` or `TR31_E6_EMV_MKEY_OTHER`. Please see Keys for Cryptographic Operations for more details.

```
$ aws payment-cryptography-data encrypt-data --key-identifier
 arn:aws:payment-cryptography:us-east-2:111122223333:key/tqv5yij6wtxx64pi
 --plain-text 33612AB9D6929C3A828EB6030082B2BD --encryption-attributes
 'Emv={MajorKeyDerivationMode=EMV_OPTION_A,PanSequenceNumber=27,PrimaryAccountNumber=1000000000
 InitializationVector=1500000000000999,Mode=CBC}'
```

```
{
    "KeyArn": "arn:aws:payment-cryptography:us-east-2:111122223333:key/
tqv5yij6wtxx64pi",
    "KeyCheckValue": "71D7AE",
    "CipherText": "33612AB9D6929C3A828EB6030082B2BD"
}
```

Encrypt data using an RSA key

Example

In this example, we will encrypt plaintext data using an RSA public key which has been imported using the ImportKey operation. For this operation, the key must have KeyModesOfUse set to `Encrypt` and KeyUsage set to `TR31_D1_ASYMMETRIC_KEY_FOR_DATA_ENCRYPTION`. Please see Keys for Cryptographic Operations for more options.

For PKCS #7 or other padding schemes not currently supported, please apply prior to calling the service and select no padding by omitting the padding indicator *'Asymmetric={}'*

```
$ aws payment-cryptography-data encrypt-data --key-identifier
 arn:aws:payment-cryptography:us-east-2:111122223333:key/thfezpmsalcfwmsg
 --plain-text 31323334313233343132333431323334 --encryption-attributes
 'Asymmetric={PaddingType=OAEP_SHA256}'
```

```
{
 "CipherText":
 "12DF6A2F64CC566D124900D68E8AFEAA794CA819876E258564D525001D00AC93047A83FB13 \
 E73F06329A100704FA484A15A49F06A7A2E55A241D276491AA91F6D2D8590C60CDE57A642BC64A897F4832A3930
 \
 0FAEC7981102CA0F7370BFBF757F271EF0BB2516007AB111060A9633D1736A9158042D30C5AE11F8C5473EC70F067
 \
 72590DEA1638E2B41FAE6FB1662258596072B13F8E2F62F5D9FAF92C12BB70F42F2ECDCF56AADF0E311D4118FE3591
 \
 FB672998CCE9D00FFFE05D2CD154E3120C5443C8CF9131C7A6A6C05F5723B8F5C07A4003A5A6173E1B425E2B5E42AD
 \
 7A2966734309387C9938B029AFB20828ACFC6D00CD1539234A4A8D9B94CDD4F23A",
 "KeyArn": "arn:aws:payment-cryptography:us-east-1:111122223333:key/5dza7xqd6soanjtb",
 "KeyCheckValue": "FF9DE9CE"
}
```

Decrypt data

The `Decrypt Data` API is used to decrypt data using symmetric and asymmetric data encryption keys as well as DUKPT and EMV derived keys. Various algorithms and variations are supported including `TDES`, `RSA` and `AES`.

The primary inputs are the decryption key used to decrypt the data, the ciphertext data in hexBinary format to be decrypted and decryption attributes such as initialization vector, mode as block ciphers etc. The primary outputs include the decrypted data as plaintext in hexBinary format and the checksum value for the decryption key. For details on all available options, please consult the API Guide for Decrypt.

Examples

- Decrypt data using AES symmetric key
- Decrypt data using DUKPT key
- Decrypt data using EMV-derived symmetric key
- Decrypt data using an RSA key

Decrypt data using AES symmetric key

Example

In this example, we will decrypt ciphertext data using a symmetric key. This example shows an AES key but TDES_2KEY and TDES_3KEY are also supported. For this operation, the key must have KeyModesOfUse set to Decrypt and KeyUsage set to TR31_D0_SYMMETRIC_DATA_ENCRYPTION_KEY. Please see Keys for Cryptographic Operations for more options.

```
$ aws payment-cryptography-data decrypt-data --key-identifier arn:aws:payment-cryptography:us-east-2:111122223333:key/tqv5yij6wtxx64pi --cipher-text 33612AB9D6929C3A828EB6030082B2BD --decryption-attributes 'Symmetric={Mode=CBC}'
```

```
{
    "KeyArn": "arn:aws:payment-cryptography:us-east-2:111122223333:key/tqv5yij6wtxx64pi",
    "KeyCheckValue": "71D7AE",
    "PlainText": "31323334313233343132333431323334"
}
```

Decrypt data using DUKPT key

Note

Using decrypt-data with DUKPT for P2PE transactions may return credit card PAN and other cardholder data to your application that will need to accounted for when determining its PCI DSS scope.

Example

In this example, we will decrypt ciphertext data using a DUKPT key which has been created using the CreateKey Operation or imported using the ImportKey Operation. For this operation, the key must have KeyModesOfUse set to `DeriveKey` and KeyUsage set to `TR31_B0_BASE_DERIVATION_KEY`. Please see Keys for Cryptographic Operations for more options. When you use `DUKPT`, for `TDES` algorithm, the ciphertext data length must be a multiple of 16 bytes. For `AES` algorithm, the ciphertext data length must be a multiple of 32 bytes.

```
$ aws payment-cryptography-data decrypt-data --key-identifier
 arn:aws:payment-cryptography:us-east-2:111122223333:key/tqv5yij6wtxx64pi
 --cipher-text 33612AB9D6929C3A828EB6030082B2BD --decryption-attributes
 'Dukpt={KeySerialNumber=FFFF9876543210E00001}'
```

```
{
    "KeyArn": "arn:aws:payment-cryptography:us-east-2:111122223333:key/
tqv5yij6wtxx64pi",
    "KeyCheckValue": "71D7AE",
    "PlainText": "31323334313233343132333431323334"
}
```

Decrypt data using EMV-derived symmetric key

Example

In this example, we will decrypt ciphertext data using an EMV-derived symmetric key which has been created using the CreateKey operation or imported using the ImportKey operation. For this operation, the key must have KeyModesOfUse set to `Derive` and KeyUsage set to `TR31_E1_EMV_MKEY_CONFIDENTIALITY` or `TR31_E6_EMV_MKEY_OTHER`. Please see Keys for Cryptographic Operations for more details.

```
$ aws payment-cryptography-data decrypt-data --key-identifier
 arn:aws:payment-cryptography:us-east-2:111122223333:key/tqv5yij6wtxx64pi
 --cipher-text 33612AB9D6929C3A828EB6030082B2BD --decryption-attributes
 'Emv={MajorKeyDerivationMode=EMV_OPTION_A,PanSequenceNumber=27,PrimaryAccountNumber=1000000000
 InitializationVector=1500000000000999,Mode=CBC}'
```

```
{
"KeyArn": "arn:aws:payment-cryptography:us-east-2:111122223333:key/tqv5yij6wtxx64pi",
"KeyCheckValue": "71D7AE",
"PlainText": "31323334313233343132333431323334"
}
```

Decrypt data using an RSA key

Example

In this example, we will decrypt ciphertext data using an RSA key pair which has been created using the CreateKey operation. For this operation, the key must have KeyModesOfUse set to enable `Decrypt` and KeyUsage set to `TR31_D1_ASYMMETRIC_KEY_FOR_DATA_ENCRYPTION`. Please see Keys for Cryptographic Operations for more options.

For PKCS #7 or other padding schemes not currently supported, please select no padding by omitting the padding indicator *'Asymmetric={}'* and remove padding subsequent to calling the service.

```
$ aws payment-cryptography-data decrypt-data \
        --key-identifier arn:aws:payment-cryptography:us-
east-2:111122223333:key/5dza7xqd6soanjtb --cipher-text
 8F4C1CAFE7A5DEF9A40BEDE7F2A264635C... \
        --decryption-attributes 'Asymmetric={PaddingType=OAEP_SHA256}'
```

```
{
    "KeyArn": "arn:aws:payment-cryptography:us-
east-1:111122223333:key/5dza7xqd6soanjtb",
    "KeyCheckValue": "FF9DE9CE",
    "PlainText": "31323334313233343132333431323334"
}
```

Generate and verify card data

Generate and verify card data incorporates data derived from card data, for instance CVV, CVV2, CVC and DCVV.

Topics

- Generate card data
- Verify card data

Generate card data

The `Generate Card Data` API is used to generate card data using algorithms such as CVV,CVV2 or Dynamic CVV2. To see what keys can be used for this command, please see Valid keys for cryptographic operations section.

Many cryptographic values such as CVV, CVV2, iCVV, CAVV V7 use the same cryptographic algorithm but vary the input values. For instance CardVerificationValue1 has inputs of ServiceCode, Card Number and Expiration Date. While CardVerificationValue2 only has two of these inputs, this is because for CVV2/CVC2, the ServiceCode is fixed at 000. Similarly, for iCVV the ServiceCode is fixed at 999. Some algorithms may repurpose the existing fields such as CAVV V8 in which case you will need to consult your provider manual for the correct input values.

Note

Expiration date must be entered in the same format (such as MMYY vs. YYMM) for generation and validation to produce correct results.

Generate CVV2

Example

In this example, we will generate a CVV2 for a given PAN with inputs of PAN and card expiration date. This assumes that you have a card verification key generated.

```
$ aws payment-cryptography-data generate-card-validation-data --key-
identifier arn:aws:payment-cryptography:us-east-2:111122223333:key/
tqv5yij6wtxx64pi --primary-account-number=171234567890123 --generation-attributes
 CardVerificationValue2={CardExpiryDate=0123}
```

```
  {
      "KeyArn": "arn:aws:payment-cryptography:us-east-2:111122223333:key/
tqv5yij6wtxx64pi",
      "KeyCheckValue": "CADDA1",
      "ValidationData": "801"
  }
```

Generate iCVV

Example

In this example, we will generate a iCVV for a given PAN with inputs of PAN,a service code of 999 and card expiration date. This assumes that you have a card verification key generated.

For all available parameters see CardVerificationValue1 in the API reference guide.

```
$ aws payment-cryptography-data generate-card-validation-data --key-
identifier arn:aws:payment-cryptography:us-east-2:111122223333:key/
tqv5yij6wtxx64pi --primary-account-number=171234567890123 --generation-attributes
 CardVerificationValue1='{CardExpiryDate=1127,ServiceCode=999}'
```

```

{
    "KeyArn": "arn:aws:payment-cryptography:us-east-2:111122223333:key/
tqv5yij6wtxx64pi",
    "KeyCheckValue": "CADDA1",
    "ValidationData": "801"
}
```

Verify card data

`Verify Card Data` is used to verify data that has been created using payment algorithms that rely on encryption principals such as `DISCOVER_DYNAMIC_CARD_VERIFICATION_CODE`.

The input values are typically provided as part of an inbound transaction to an issuer or supporting platform partner. To verify an ARQC cryptogram (used for EMV chips cards), please see Verify ARQC.

For more information, see VerifyCardValidationData in the API guide.

If the value is verified, then the api will return http/200. If the value is not verified, it will return http/400.

Verify CVV2

Example

In this example, we will validate a CVV/CVV2 for a given PAN. The CVV2 is typically provided by the cardholder or user during transaction time for validation. In order to validate their input, the following values will be provided at runtime - Key to Use for validation (CVK), PAN, card expiration date and CVV2 entered. Card expiration format must match that used in initial value generation.

For all available parameters see CardVerificationValue2 in the API reference guide.

```
$ aws payment-cryptography-data verify-card-validation-data --key-identifier
 arn:aws:payment-cryptography:us-east-2:111122223333:key/tqv5yij6wtxx64pi
 --primary-account-number=171234567890123 --verification-attributes
 CardVerificationValue2={CardExpiryDate=0123} --validation-data 801
```

```
  {
      "KeyArn": "arn:aws:payment-cryptography:us-east-2:111122223333:key/
tqv5yij6wtxx64pi",
      "KeyCheckValue": "CADDA1"
  }
```

Verify iCVV

Example

In this example, we will verify a iCVV for a given PAN with inputs of Key to Use for validation (CVK), PAN, a service code of 999, card expiration date and the iCVV provided by the transaction to validate.

iCVV is not a user entered value (like CVV2) but embedded on an EMV card. Consideration should be given to whether it should always validate when provided.

For all available parameters see, CardVerificationValue1 in the API reference guide.

```
$ aws payment-cryptography-data verify-card-validation-data --key-identifier
 arn:aws:payment-cryptography:us-east-2:111122223333:key/tqv5yij6wtxx64pi
 --primary-account-number=171234567890123 --verification-attributes
 CardVerificationValue1='{CardExpiryDate=1127,ServiceCode=999} --validation-data 801
```

```
{
    "KeyArn": "arn:aws:payment-cryptography:us-east-2:111122223333:key/
tqv5yij6wtxx64pi",
    "KeyCheckValue": "CADDA1",
    "ValidationData": "801"
}
```

Generate, translate and verify PIN data

PIN data functions allow you to generate random pins, pin verification values (PVV) and validate inbound encrypted pins against PVV or PIN Offsets.

Pin translation allows you to translate a pin from one working key to another without exposing the pin in clear text as specified by PCI PIN Requirement 1.

Note

As PIN generation and validation are typically issuer functions and PIN translation is a typical acquirer function, we recommend that you consider least priviledged access and set policies appropriately for your systems use case.

Topics

- Translate PIN data
- Generate PIN data
- Verify PIN data

Translate PIN data

Translate PIN data functions are used for translating encrypted PIN data from one set of keys to another without the encrypted data leaving the HSM. It is used for P2PE encryption where the working keys should change but the processing system either doesn't need to, or is not permitted to, decrypt the data. The primary inputs are the encrypted data, the encryption key used to encrypt the data, the parameters used to generate the input values. The other set of inputs are the requested output parameters such as the key to be used to encrypt the output and the parameters used to create that output. The primary outputs are a newly encrypted dataset as well as the parameters used to generate it.

Note

For PCI compliance, the incoming and outgoing PrimaryAccountNumber values must match. Translating a PIN from one PAN to another is not permitted.

Topics

- PIN from PEK to DUKPT
- PIN from PEK to PEK

PIN from PEK to DUKPT

Example

In this example, we will translate a PIN from an AES ISO 4 PIN Block using the DUKPT to PEK TDES encryption using ISO 0 PIN block. This is common where a payment terminal encrypts a pin in ISO 4 and then it may be translated back to TDES for downstream processing if the next connection doesn't yet support AES.

```
$ aws payment-cryptography-data translate-pin-data --encrypted-pin-block
 "AC17DC148BDA645E" --outgoing-translation-
attributes=IsoFormat0='{PrimaryAccountNumber=171234567890123}' --outgoing-
key-identifier arn:aws:payment-cryptography:us-east-2:111122223333:key/
ivi5ksfsuplneuyt  --incoming-key-identifier arn:aws:payment-cryptography:us-
east-2:111122223333:key/4pmyquwjs3yj4vwe --incoming-translation-attributes
 IsoFormat4="{PrimaryAccountNumber=171234567890123}" --incoming-dukpt-attributes
 KeySerialNumber="FFFF9876543210E00008"
```

```
    {
            "PinBlock": "1F4209C670E49F83E75CC72E81B787D9",
            "KeyArn": "arn:aws:payment-cryptography:us-east-2:111122223333:key/
ivi5ksfsuplneuyt",
            "KeyCheckValue": "7CC9E2"
        }
```

PIN from PEK to PEK

Example

In this example, we translate a PIN encrypted under one PEK (PIN Encryption Key) to another PEK. This is commonly used when routing transactions between different systems or partners that use different encryption keys, while maintaining PCI PIN compliance by keeping the PIN encrypted throughout the process. Both keys use TDES 3KEY encryption in this example, but a variety of options are available including AES ISO-4 to TDES ISO-0, DUKPT to PEK, or AS2805 to PEK.

```
$ aws payment-cryptography-data translate-pin-data --encrypted-pin-block
 "AC17DC148BDA645E" \
    --incoming-translation-attributes
 IsoFormat0='{PrimaryAccountNumber=171234567890123}' \
    --incoming-key-identifier arn:aws:payment-cryptography:us-east-2:111122223333:key/
ivi5ksfsuplneuyt \
    --outgoing-translation-attributes
 IsoFormat0='{PrimaryAccountNumber=171234567890123}' \
    --outgoing-key-identifier arn:aws:payment-cryptography:us-east-2:111122223333:key/
alsuwfxug3pgy6xh
```

```
{
    "PinBlock": "E8F2A6C4D1B93E7F",
    "KeyArn": "arn:aws:payment-cryptography:us-east-2:111122223333:key/
alsuwfxug3pgy6xh",
    "KeyCheckValue": "9A325B"
}
```

The output PIN block is now encrypted under the second PEK and can be safely transmitted to the downstream system that holds the corresponding key.

Generate PIN data

Generate PIN data functions are used for generating PIN-related values, such as PVV and pin block offsets used for validating pin entry by users during transaction or authorization time. This API can also generate a new random pin using various algorithms.

Generate a random pin and matching Visa PVV

Example

In this example, we will generate a new (random) pin where the outputs will be an encrypted `PIN block` (PinData.PinBlock) and a PVV (pinData.Offset). The key inputs are [PAN](), the [Pin Verification Key](), the [Pin Encryption Key]() and the `PIN block format`.

This command requires that the key is of type `TR31_V2_VISA_PIN_VERIFICATION_KEY`.

```
$ aws payment-cryptography-data generate-pin-data --generation-key-identifier
 arn:aws:payment-cryptography:us-east-2:111122223333:key/37y2tsl45p5zjbh2 --encryption-
key-identifier arn:aws:payment-cryptography:us-east-2:111122223333:key/ivi5ksfsuplneuyt
 --primary-account-number 171234567890123 --pin-block-format ISO_FORMAT_0 --generation-
attributes VisaPin={PinVerificationKeyIndex=1}
```

```
{
            "GenerationKeyArn": "arn:aws:payment-cryptography:us-
east-2:111122223333:key/37y2tsl45p5zjbh2",
            "GenerationKeyCheckValue": "7F2363",
            "EncryptionKeyArn": "arn:aws:payment-cryptography:us-
east-2:111122223333:key/ivi5ksfsuplneuyt",
            "EncryptionKeyCheckValue": "7CC9E2",
            "EncryptedPinBlock": "AC17DC148BDA645E",
            "PinData": {
                "VerificationValue": "5507"
            }
        }
```

Generate a Visa PVV for a known pin

Example

In this example, we will generate a PVV for a given (encrypted) pin. An encrypted pin may be received upstream such as from a payment terminal or from a cardholder using the user selectable pin flow. The key inputs are PAN, the `Pin Verification Key`, the `Pin Encryption Key`, the `Encrypted Pin Block` and the `PIN block format`.

```
$ aws payment-cryptography-data generate-pin-data --generation-key-identifier
 arn:aws:payment-cryptography:us-east-2:111122223333:key/37y2tsl45p5zjbh2
 --encryption-key-identifier arn:aws:payment-cryptography:us-
east-2:111122223333:key/ivi5ksfsuplneuyt --primary-account-number
 171234567890123 --pin-block-format ISO_FORMAT_0 --generation-attributes
 VisaPinVerificationValue={PinVerificationKeyIndex=1,EncryptedPinBlock=AA584CED31790F37}
```

```
{
            "GenerationKeyArn": "arn:aws:payment-cryptography:us-
east-2:111122223333:key/37y2tsl45p5zjbh2",
            "GenerationKeyCheckValue": "7F2363",
            "EncryptionKeyArn": "arn:aws:payment-cryptography:us-
east-2:111122223333:key/ivi5ksfsuplneuyt",
            "EncryptionKeyCheckValue": "7CC9E2",
            "EncryptedPinBlock": "AC17DC148BDA645E",
            "PinData": {
                "VerificationValue": "5507"
            }
        }
```

Generate IBM3624 pin offset for a pin

IBM 3624 PIN Offset also sometimes called the IBM method. This method generates a natural/ intermediate PIN using the validation data (typically the PAN) and a PIN Key (PVK). Natural pins are effectively a derived value and being deterministic are very efficient to handle for an issuer because no pin data needs to be stored at a cardholder level. The most obvious con is that this scheme doesn't account for cardholder selectable or random pins. To allow for those types of pins, an offset algorithm was added to the scheme. The offset represents the difference between the user selected(or random) pin and the natural key. The offset value is stored by the card issuer or card processor. At transaction time, the AWS Payment Cryptography service internally recalculates the

natural pin and applies the offset to find the pin. It then compares this against the value provided by the transaction authorization.

Several options exist for IBM3624:

- `Ibm3624NaturalPin` will output the natural pin and an encrypted pin block
- `Ibm3624PinFromOffset` will generate an encrypted pin block given an offset
- `Ibm3624RandomPin` will generate a random pin and then the matching offset and encrypted pin block.
- `Ibm3624PinOffset` generates the pin offset given a user selected pin.

Internal to AWS Payment Cryptography, the following steps are performed:

- Pad the provided pan to 16 characters. If <16 are provided, pad on the right hand side using the provided padding character.
- Encrypts the validation data using the PIN generation key.
- Decimalize the encrypted data using the decimalization table. This maps hexidecimal digits to decimal digits for instance 'A' may map to 9 and 1 may map to 1.
- Get the first 4 digits from a hexidecimal representation of the output. This is the natural pin.
- If a user selected or random pin was generated, modulo subtract the natural pin with customer pin. The result is the pin offset.

Examples

- Example: Generate IBM3624 pin offset for a pin

Example: Generate IBM3624 pin offset for a pin

In this example, we will generate a new (random) pin where the outputs will be an encrypted `PIN block` (PinData.PinBlock) and an `IBM3624` offset value (pinData.Offset). The inputs are PAN, validation data (typically the pan), padding character, the `Pin Verification Key`, the `Pin Encryption Key` and the `PIN block format`.

This command requires that the pin generation key is of type `TR31_V1_IBM3624_PIN_VERIFICATION_KEY` and the encryption key is of type `TR31_P0_PIN_ENCRYPTION_KEY`

Example

The following example shows generating a random pin then outputting the encrypted pin block and IBM3624 offset value using Ibm3624RandomPin

```
$ aws payment-cryptography-data generate-pin-data --generation-key-identifier
 arn:aws:payment-cryptography:us-east-2:111122223333:key/37y2tsl45p5zjbh2
 --encryption-key-identifier arn:aws:payment-cryptography:us-
east-2:111122223333:key/ivi5ksfsuplneuyt --primary-account-number
 171234567890123 --pin-block-format ISO_FORMAT_0 --generation-attributes
 Ibm3624RandomPin="{DecimalizationTable=9876543210654321,PinValidationDataPadCharacter=D,PinVal
```

```
{
              "GenerationKeyArn": "arn:aws:payment-cryptography:us-
east-2:111122223333:key/37y2tsl45p5zjbh2",
              "GenerationKeyCheckValue": "7F2363",
              "EncryptionKeyArn": "arn:aws:payment-cryptography:us-
east-2:111122223333:key/ivi5ksfsuplneuyt",
              "EncryptionKeyCheckValue": "7CC9E2",
              "EncryptedPinBlock": "AC17DC148BDA645E",
              "PinData": {
                  "PinOffset": "5507"
              }
          }
```

Verify PIN data

Verify PIN data functions are used for verifying whether a pin is correct. This typically involves comparing the pin value previously stored against what was entered by the cardholder at a POI. These functions compare two values without exposing the underlying value of either source.

Validate encrypted PIN using PVV method

Example

In this example, we will validate a PIN for a given PAN. The PIN is typically provided by the cardholder or user during transaction time for validation and is compared against the value on file (the input from the cardholder is provided as an encrypted value from the terminal or other upstream provider). In order to validate this input, the following values will also be provided at runtime: The key used to encrypt the input pin (this is often an `IWK`), PAN and the value to verify against (either a `PVV` or `PIN offset`).

If AWS Payment Cryptography is able to validate the pin, an http/200 is returned. If the pin is not validated, it will return an http/400.

```
$ aws payment-cryptography-data verify-pin-data --verification-key-identifier
 arn:aws:payment-cryptography:us-east-2:111122223333:key/37y2tsl45p5zjbh2 --encryption-
key-identifier arn:aws:payment-cryptography:us-east-2:111122223333:key/ivi5ksfsuplneuyt
 --primary-account-number 171234567890123 --pin-block-format ISO_FORMAT_0  --
verification-attributes VisaPin="{PinVerificationKeyIndex=1,VerificationValue=5507}" --
encrypted-pin-block AC17DC148BDA645E
```

```
        {
            "VerificationKeyArn": "arn:aws:payment-cryptography:us-
east-2:111122223333:key/37y2tsl45p5zjbh2",
            "VerificationKeyCheckValue": "7F2363",
            "EncryptionKeyArn": "arn:aws:payment-cryptography:us-
east-2:111122223333:key/ivi5ksfsuplneuyt",
            "EncryptionKeyCheckValue": "7CC9E2",

        }
```

Validate encrypted PIN using PVV method - error bad pin

Example

In this example, we will attempt to validate a PIN for a given PAN but it will fail due to the pin being incorrect.

When using SDKs, this appears as {"Message":"Pin block verification failed.","Reason":"INVALID_PIN"}

```
$ aws payment-cryptography-data verify-pin-data --verification-key-identifier
 arn:aws:payment-cryptography:us-east-2:111122223333:key/37y2tsl45p5zjbh2 --encryption-
key-identifier arn:aws:payment-cryptography:us-east-2:111122223333:key/ivi5ksfsuplneuyt
 --primary-account-number 171234567890123 --pin-block-format ISO_FORMAT_0  --
verification-attributes VisaPin="{PinVerificationKeyIndex=1,VerificationValue=9999}" --
encrypted-pin-block AC17DC148BDA645E
```

```
       An error occurred (VerificationFailedException) when calling the VerifyPinData
 operation: Pin block verification failed.
```

Validate encrypted PIN using PVV method - error bad inputs

Example

In this example, we will attempt to validate a PIN for a given PAN but it will fail due to bad inputs and the incoming data was not a valid pin. Common causes are: 1/wrong key being used 2/input parameters such as pan or pin block format are incorrect 3/pin block is corrupted.

```
$ aws payment-cryptography-data verify-pin-data --verification-key-identifier
 arn:aws:payment-cryptography:us-east-2:111122223333:key/37y2tsl45p5zjbh2
 --encryption-key-identifier  --primary-account-number 171234567890123
 --pin-block-format ISO_FORMAT_0  --verification-attributes
 VisaPin="{PinVerificationKeyIndex=1,VerificationValue=9999}" --encrypted-pin-block
 AC17DC148BDA645E
```

```
        An error occurred (ValidationException) when calling the VerifyPinData
 operation: Pin block provided is invalid. Please check your input to ensure all field
 values are correct.
```

Validate a PIN against previously stored IBM3624 pin offset

In this example, we will validate a cardholder provided PIN against the pin offset stored on file with the card issuer/processor. The inputs are similar to ??? with the additional of the encrypted pin provided by the payment terminal (or other upstream provider such as card network). If the pin matches, the api will return http 200. where the outputs will be an encrypted `PIN block` (PinData.PinBlock) and an `IBM3624` offset value (pinData.Offset).

This command requires that the pin generation key is of type `TR31_V1_IBM3624_PIN_VERIFICATION_KEY` and the encryption key is of type `TR31_P0_PIN_ENCRYPTION_KEY`

Example

```
$ aws payment-cryptography-data generate-pin-data --generation-key-identifier
 arn:aws:payment-cryptography:us-east-2:111122223333:key/37y2tsl45p5zjbh2
 --encryption-key-identifier arn:aws:payment-cryptography:us-
east-2:111122223333:key/ivi5ksfsuplneuyt --primary-account-number
 171234567890123 --pin-block-format ISO_FORMAT_0 --generation-attributes
 Ibm3624RandomPin="{DecimalizationTable=9876543210654321,PinValidationDataPadCharacter=D,PinVal
```

```
{
"GenerationKeyArn": "arn:aws:payment-cryptography:us-
east-2:111122223333:key/37y2tsl45p5zjbh2",
"GenerationKeyCheckValue": "7F2363",
"EncryptionKeyArn": "arn:aws:payment-cryptography:us-east-2:111122223333:key/
ivi5ksfsuplneuyt",
"EncryptionKeyCheckValue": "7CC9E2",
"EncryptedPinBlock": "AC17DC148BDA645E",
"PinData": {
  "PinOffset": "5507"
}
}
```

Verify auth request (ARQC) cryptogram

The verify auth request cryptogram API is used for verifying ARQC. The generation of the ARQC is outside of the scope of the AWS Payment Cryptography and is typically performed on an EMV Chip Card (or digital equivalent such as mobile wallet) during transaction authorization time. An ARQC is unique to each transactions and is intended to cryptographically show both the validity of the card as well as to ensure that the transaction data exactly matches the current (expected) transaction.

AWS Payment Cryptography provides a variety of options for validating ARQC and generating optional ARPC values including those defined in EMV 4.4 Book 2 and other schemes used by Visa and Mastercard. For a full list of all available options, please see the VerifyCardValidationData section in the API Guide.

ARQC cryptograms typically require the following inputs (although this may vary by implementation):

- PAN - Specified in the PrimaryAccountNumber field
- PAN Sequence Number (PSN) - specified in the PanSequenceNumber field

- Key Derivation Method such as Common Session Key (CSK) - Specified in the SessionKeyDerivationAttributes
- Master Key Derivation Mode (such as EMV Option A) - Specified in the MajorKeyDerivationMode
- Transaction data - a string of various transaction, terminal and card data such as Amount and Date - specified in the TransactionData field
- Issuer Master Key - the master key used to derive the cryptogram (AC) key used to protect individual transactions and specified in the KeyIdentifier field

Topics

- Building transaction data
- Transaction data padding
- Examples

Building transaction data

The exact content (and order) of the transaction data field varies by implementation and network scheme but the minimum recommended fields (and concatenation sequence) is defined in EMV 4.4 Book 2 Section 8.1.1 - Data Selection. If the first three fields are amount (17.00), other amount (0.00) and country of purchase, that would result in the transaction data beginning as follows:

- 000000001700 - amount - 12 positions implied two digit decimal
- 000000000000 - other amount - 12 positions implied two digit decimal
- 0124 - four digit country code
- Output (partial) Transaction Data - 0000000017000000000000000124

Transaction data padding

Transaction data should be padded prior to sending to the service. Most schemes use ISO 9797 Method 2 padding, where a hex string is appended by hex 80 followed by 00 until the field is a multiple of the encryption block size; 8 bytes or 16 characters for TDES and 16 bytes or 32 characters for AES. The alternative (method 1) is not as common but uses only 00 as the padding characters.

ISO 9797 Method 1 Padding

Unpadded:
00000000170000000000000008400080008000084016051700000000938000000B03011203
(74 characters or 37 bytes)

Padded:
00000000170000000000000008400080008000084016051700000000938000000B03011203**000000**
(80 characters or 40 bytes)

ISO 9797 Method 2 Padding

Unpadded:
00000000170000000000000008400080008000084016051700000000938000000B1F220103000000
(80 characters or 40 bytes)

Padded:
00000000170000000000000008400080008000084016051700000000938000000B1F220103000000**80**
(88 characters or 44 bytes)

Examples

Visa CVN10

Example

In this example, we will validate an ARQC generated using Visa CVN10.

If AWS Payment Cryptography is able to validate the ARQC, an http/200 is returned. If then ARCQ (Authorization Request Cryptogram) is not validated, it will return a http/400 response.

```
$ aws payment-cryptography-data verify-auth-request-cryptogram --auth-request-
cryptogram D791093C8A921769 \
--key-identifier arn:aws:payment-cryptography:us-east-2:111122223333:key/
pw3s6nl62t5ushfk \
--major-key-derivation-mode EMV_OPTION_A \
--transaction-data
 00000000170000000000000008400080008000084016051700000000093800000B03011203000000 \
--session-key-derivation-attributes='{"Visa":{"PanSequenceNumber":"01" \
,"PrimaryAccountNumber":"9137631040001422"}}'
```

```
{
  "KeyArn": "arn:aws:payment-cryptography:us-east-2:111122223333:key/pw3s6nl62t5ushfk",
  "KeyCheckValue": "08D7B4"
}
```

Visa CVN18 and Visa CVN22

Example

In this example, we will validate an ARQC generated using Visa CVN18 or CVN22. The cryptographic operations are the same between CVN18 and CVN22 but the data contained within transaction data varies. Compared to CVN10, a completely different cryptogram is generated even with the same inputs.

If AWS Payment Cryptography is able to validate the ARQC, an http/200 is returned. If the ARCQ is not validated, it will return an http/400.

```
$ aws payment-cryptography-data verify-auth-request-cryptogram \
--auth-request-cryptogram 61EDCC708B4C97B4
--key-identifier arn:aws:payment-cryptography:us-east-2:111122223333:key/
pw3s6nl62t5ushfk \
--major-key-derivation-mode EMV_OPTION_A
--transaction-data
 0000000017000000000000000084000800080000840160517000000000938000000B1F22010300000000000
 \
00000000000000000000000000000000000000000080000000000000000
--session-key-derivation-attributes='{"EmvCommon":
{"ApplicationTransactionCounter":"000B", \
"PanSequenceNumber":"01","PrimaryAccountNumber":"9137631040001422"}}'
```

```
{
  "KeyArn": "arn:aws:payment-cryptography:us-east-2:111122223333:key/pw3s6nl62t5ushfk",
  "KeyCheckValue": "08D7B4"
}
```

Generate and verify MAC

Message Authentication Codes (MAC) are typically used to authenticate the integrity of a message (whether it's been modified). Cryptographic hashes such as HMAC (Hash-Based Message Authentication Code), CBC-MAC and CMAC (Cipher-based Message Authentication Code) provide additional assurance of the sender of the MAC by utilizing cryptography. HMAC is based on hash functions while CMAC is based on block ciphers. The service also supports ISO9797 Algorithms 1 and 3 which are types of CBC-MACs.

All MAC algorithms of this service combine a cryptographic hash function and a shared secret key. They take a message and a secret key, such as the key material in a key, and return a unique tag or mac. If even one character of the message changes, or if the secret key changes, the resulting tag is entirely different. By requiring a secret key, cryptographic MACs also provides authenticity; it is impossible to generate an identical mac without the secret key. Cryptographic MACs are sometimes called symmetric signatures, because they work like digital signatures, but use a single key for both signing and verification.

AWS Payment Cryptography supports several types of MACs:

ISO9797 ALGORITHM 1

Denoted by `KeyUsage` of ISO9797_ALGORITHM1. If the field isn't a multiple of block size (8 bytes/16 hex characters for TDES, 16 bytes/32 characters for AES, AWS Payment Cryptography automatically applies ISO9797 Padding Method 1. If other padding methods are needed, you can apply them prior to calling the service.

ISO9797 ALGORITHM 3 (Retail MAC)

Denoted by `KeyUsage` of ISO9797_ALGORITHM3. The same padding rules apply as Algorithm 1

ISO9797 ALGORITHM 5 (CMAC)

Denoted by `KeyUsage` of TR31_M6_ISO_9797_5_CMAC_KEY

HMAC

Denoted by `KeyUsage` of TR31_M7_HMAC_KEY including HMAC_SHA224, HMAC_SHA256, HMAC_SHA384 and HMAC_SHA512

AS2805.4.1 MAC

Denoted by `KeyUsage` of TR31_M0_ISO_16609_MAC_KEY. For more details on AS2805, see ???

DUKPT MAC

DUKPT MAC is typically used to confirm the source and payload of messages to/from payment terminals. It derives a key using DUKPT derivation techniques and then performs the MAC. Keys used with this option are denoted by a `KeyUsage` of TR31_B0_BASE_DERIVATION_KEY.

EMV MAC

EMV MAC is typically referred to as an integrity key in EMV documentation. It derives a key using EMV derivation techniques and then utilizes ISO9797_ALGORITHM3 internally. It is

typically used to send issuer scripts to a chip card for reprogramming. Keys used with this option are denoted by a `KeyUsage` of TR31_E2_EMV_MKEY_INTEGRITY. If you are both sending a script and update an offline pin, see GenerateMacEmvPinChange that performs both of these operations.

Topics

- Generate MAC
- Verify MAC

Generate MAC

Generate MAC API is used to authenticate card-related data, such as track data from a card magnetic stripe, by using known cryptographic keys to generate a MAC (Message Authentication Code) for data validation between sending and receiving parties. The data used to generate MAC includes message data, secret MAC encryption key and MAC algorithm to generate a unique MAC value for transmission. The receiving party of the MAC will use the same MAC message data, MAC encryption key, and algorithm to reproduce another MAC value for comparison and data authentication. Even if one character of the message changes or the MAC key used for verification is not identical, the resulting MAC value is different. The API supports ISO 9797-1 Algorithm 1 and ISO 9797-1 Algorithm 3 MAC (using a static MAC key and a derived DUKPT key), HMAC and EMV MAC encryption keys for this operation.

The input value for `message-data` must be hexBinary data.

For more information on all options for this API, see GenerateMac and VerifyMac.

The optional parameter mac-length allows you to truncate the output value (although this can also be done within your code). A length of 8 refers to 8 bytes or 16 hex characters.

MAC keys can either be created with AWS Payment Cryptography by calling CreateKey or imported by calling ImportKey.

Note

CMAC and HMAC algorithms don't require padding. All others require that the data be padded to the block size of the algorithm, which is multiples of 8 bytes (16 hex characters) for TDES and 16 bytes (32 hex characters) for AES.

Examples

- Generate HMAC
- Generate MAC using ISO 9797-1 Algorithm 3
- Generate MAC using CMAC
- Generate MAC using DUKPT CMAC

Generate HMAC

In this example, we will generate a HMAC (Hash-Based Message Authentication Code) for card data authentication using HMAC algorithm HMAC_SHA256 and HMAC encryption key. The key must have KeyUsage set to TR31_M7_HMAC_KEY and KeyModesOfUse to Generate. The hash length (e.g. 256) is defined when the key is created and cannot be modified.

The optional mac-length parameter will trim the output MAC, although this can be performed outside the service as well. This value is in bytes, so a value of 16 will expect a hex string of length 32.

Example

```
$ aws payment-cryptography-data generate-mac \
    --key-identifier arn:aws:payment-cryptography:us-east-2:111122223333:key/
qnobl5lghrzunce6 \
    --message-data
 "3b313038383439303031303733393431353d32343038323236303030373030303f33" \
    --generation-attributes Algorithm=HMAC
```

```
{
    "KeyArn": "arn:aws:payment-cryptography:us-east-2:111122223333:key/
qnobl5lghrzunce6",
    "KeyCheckValue": "2976E7",
    "Mac": "ED87F26E961C6D0DDB78DA5038AA2BDDEA0DCE03E5B5E96BDDD494F4A7AA470C"
}
```

Generate MAC using ISO 9797-1 Algorithm 3

In this example, we will generate a MAC using ISO 9797-1 Algorithm 3 (Retail MAC) for card data authentication. The key must have KeyUsage set to TR31_M3_ISO_9797_3_MAC_KEY and KeyModesOfUse to Generate.

Example

```
$ aws payment-cryptography-data generate-mac \
    --key-identifier arn:aws:payment-cryptography:us-east-2:111122223333:key/
kwapwa6qaifllw2h \
    --message-data
 "3b313038383439303031303733393431353d32343038323236303030373030303f33" \
    --generation-attributes="Algorithm=ISO9797_ALGORITHM3"
```

```
{
    "KeyArn": "arn:aws:payment-cryptography:us-east-2:111122223333:key/
kwapwa6qaifllw2h",
    "KeyCheckValue": "2976EA",
    "Mac": "A8F7A73DAF87B6D0"
}
```

Generate MAC using CMAC

CMAC is most commonly used when the keys are AES but it also supports TDES. In this example, we will generate a MAC using CMAC (ISO 9797-1 Algorithm 5) for card data authentication with an AES key. The key must have KeyUsage set to `TR31_M6_ISO_9797_5_CMAC_KEY` and KeyModesOfUse to `Generate`.

Example

```
$ aws payment-cryptography-data generate-mac \
    --key-identifier arn:aws:payment-cryptography:us-east-2:111122223333:key/
tqv5yij6wtxx64pi \
    --message-data
 "3b313038383439303031303733393431353d32343038323236303030373030303f33" \
    --generation-attributes Algorithm="CMAC"
```

```
{
    "KeyArn": "arn:aws:payment-cryptography:us-east-2:111122223333:key/
tqv5yij6wtxx64pi",
    "KeyCheckValue": "C1EB8F",
    "Mac": "1F8C36E63F91E4E93DF7842BF5E2E5F7"
}
```

Generate MAC using DUKPT CMAC

In this example, we will generate a MAC using DUKPT (Derived Unique Key Per Transaction) with CMAC for card data authentication. The key must have KeyUsage set to `TR31_B0_BASE_DERIVATION_KEY` and KeyModesOfUse `DeriveKey` set to true. DUKPT keys derive a unique key for each transaction using a Base Derivation Key (BDK) and a Key Serial Number (KSN).

Example

```
$ aws payment-cryptography-data generate-mac --key-identifier arn:aws:payment-cryptography:us-east-2:111122223333:key/qnobl5lghrzunce6 --message-data
 "3b313038383439303031303733393431353d32343038323236303030373030303f33" --generation-attributes="DukptCmac={KeySerialNumber="932A6E954ABB32DD00000001",Direction=BIDIRECTIONAL}"
```

```
{
    "KeyArn": "arn:aws:payment-cryptography:us-east-2:111122223333:key/
qnobl5lghrzunce6",
    "KeyCheckValue": "C1EB8F"
}
```

Verify MAC

Verify MAC API is used to verify MAC (Message Authentication Code) for card-related data authentication. It must use the same encryption key used during generate MAC to re-produce MAC value for authentication. The MAC encryption key can either be created with AWS Payment Cryptography by calling CreateKey or imported by calling ImportKey. The API supports DUKPT MAC, HMAC and EMV MAC encryption keys for this operation.

If the value is verified, then response parameter `MacDataVerificationSuccessful` will return `Http/200`, otherwise `Http/400` with a message indicating that `Mac verification failed`.

Examples

- Verify HMAC
- Verify MAC using DUKPT CMAC

Verify HMAC

In this example, we will verify a HMAC (Hash-Based Message Authentication Code) for card data authentication using HMAC algorithm `HMAC_SHA256` and HMAC encryption key. The key must have KeyUsage set to `TR31_M7_HMAC_KEY` and KeyModesOfUse `Verify` set to true.

Example

```
$ aws payment-cryptography-data verify-mac \
     --key-identifier arn:aws:payment-cryptography:us-east-2:111122223333:key/
qnobl5lghrzunce6 \
     --message-data
 "3b343038383439303031303733393431353d3234303832323630303037303030303f33" \
     --mac ED87F26E961C6D0DDB78DA5038AA2BDDEA0DCE03E5B5E96BDDD494F4A7AA470C \
     --verification-attributes Algorithm=HMAC_SHA256
```

```
{
    "KeyArn": "arn:aws:payment-cryptography:us-east-2:111122223333:key/
qnobl5lghrzunce6",
    "KeyCheckValue": "2976E7"
}
```

Verify MAC using DUKPT CMAC

In this example, we will verify a MAC using DUKPT (Derived Unique Key Per Transaction) with CMAC for card data authentication. The key must have KeyUsage set to `TR31_B0_BASE_DERIVATION_KEY` and KeyModesOfUse `DeriveKey` set to true. DUKPT keys derive a unique key for each transaction using a Base Derivation Key (BDK) and a Key Serial Number (KSN). The value of DukptKeyVariant must match between sender and receiver. REQUEST will typically be used from terminal to backend, VERIFY from backend to terminal and BIDIRECTIONAL when a single key is used in both directions.

Example

```
$ aws payment-cryptography-data verify-mac \
     --key-identifier arn:aws:payment-cryptography:us-east-2:111122223333:key/
tqv5yij6wtxx64pi \
     --message-data
 "3b343038383439303031303733393431353d32343038323236303030373030303f33" \
     --mac D8E804EE74BF1D909A2C01C0BDE8EF34 \
     --verification-attributes
 DukptCmac='{"KeySerialNumber":"932A6E954ABB32DD00000001","DukptKeyVariant":"BIDIRECTIONAL"}'
```

```
{
    "KeyArn": "arn:aws:payment-cryptography:us-east-2:111122223333:key/
tqv5yij6wtxx64pi",
    "KeyCheckValue": "C1EB8F"
}
```

Valid keys for cryptographic operations

Certain keys can only be used for certain operations. Additionally, some operations may limit the key modes of use for keys. Please see the following table for allowed combinations.

Note

Certain combinations, although permitted, may create unusable situations such as generating CVV codes (`generate`) but then unable to verify them (`verify`).

Topics

- GenerateCardData
- VerifyCardData
- GeneratePinData (for VISA/ABA schemes)
- GeneratePinData (for IBM3624)
- VerifyPinData (for VISA/ABA schemes)
- VerifyPinData (for IBM3624)

- Decrypt Data
- Encrypt Data
- Translate Pin Data
- Generate/Verify MAC
- GenerateMacEmvPinChange
- VerifyAuthRequestCryptogram
- Import/Export Key
- Unused key types

GenerateCardData

API Endpoint	Cryptographic Operation or Algorithm	Allowed Key Usage	Allowed Key Algorithm	Allowed combination of key modes of use
GenerateCardData	• AMEX_CARD_SECURITY_CODE_VERSION_1 • AMEX_CARD_SECURITY_CODE_VERSION_2	TR31_C0_CARD_VERIFICATION_KEY	• TDES_2KEY • TDES_3KEY	{ Generate = true },{ Generate = true, Verify = true }
GenerateCardData	• CARD_VERIFICATION_VALUE_1 • CARD_VERIFICATION_VALUE_2	TR31_C0_CARD_VERIFICATION_KEY	• TDES_2KEY	{ Generate = true },{ Generate = true, Verify = true }
GenerateCardData	• CARDHOLDER_AUTHENTICATION_V	TR31_E6_EMV_MKEY_OTHER	• TDES_2KEY	{ DeriveKey = true }

API Endpoint	Cryptographic Operation or Algorithm	Allowed Key Usage	Allowed Key Algorithm	Allowed combination of key modes of use
	ERIFICATION_VALUE			
GenerateCardData	• DYNAMIC_CARD_VERIFICATION_CODE	TR31_E4_EMV_MKEY_DYNAMIC_NUMBERS	• TDES_2KEY	{ DeriveKey = true }
GenerateCardData	• DYNAMIC_CARD_VERIFICATION_VALUE	TR31_E6_EMV_MKEY_OTHER	• TDES_2KEY	{ DeriveKey = true }

VerifyCardData

Cryptographic Operation or Algorithm	Allowed Key Usage	Allowed Key Algorithm	Allowed combination of key modes of use
• AMEX_CARD_SECURITY_CODE_VERSION_1 • AMEX_CARD_SECURITY_CODE_VERSION_2	TR31_C0_CARD_VERIFICATION_KEY	• TDES_2KEY • TDES_3KEY	{ Generate = true }, { Generate = true, Verify = true }
• CARD_VERIFICATION_VALUE_1 • CARD_VERIFICATION_VALUE_2	TR31_C0_CARD_VERIFICATION_KEY	• TDES_2KEY	{ Generate = true }, { Generate = true, Verify = true }

Cryptographic Operation or Algorithm	Allowed Key Usage	Allowed Key Algorithm	Allowed combination of key modes of use
• CARDHOLDER_AUTHENTICATION_VERIFICATION_VALUE	TR31_E6_EMV_MKEY_OTHER	• TDES_2KEY	{ DeriveKey = true }
• DYNAMIC_CARD_VERIFICATION_CODE	TR31_E4_EMV_MKEY_DYNAMIC_NUMBERS	• TDES_2KEY	{ DeriveKey = true }
• DYNAMIC_CARD_VERIFICATION_VALUE	TR31_E6_EMV_MKEY_OTHER	• TDES_2KEY	{ DeriveKey = true }

GeneratePinData (for VISA/ABA schemes)

`VISA_PIN or VISA_PIN_VERIFICATION_VALUE`

Key Type	Allowed Key Usage	Allowed Key Algorithm	Allowed combination of key modes of use
PIN Encryption Key	TR31_P0_PIN_ENCRYPTION_KEY	• TDES_2KEY • TDES_3KEY	• { Encrypt = true, Wrap = true } • { Encrypt = true, Decrypt = true, Wrap = true, Unwrap = true } • { NoRestrictions = true }
PIN Generation Key	TR31_V2_VISA_PIN_VERIFICATION_KEY	• TDES_3KEY	• { Generate = true }

Key Type	Allowed Key Usage	Allowed Key Algorithm	Allowed combination of key modes of use
			• { Generate = true, Verify = true }

GeneratePinData (for IBM3624)

IBM3624_PIN_OFFSET,IBM3624_NATURAL_PIN,IBM3624_RANDOM_PIN,
IBM3624_PIN_FROM_OFFSET)

Key Type	Allowed Key Usage	Allowed Key Algorithm	Allowed combination of key modes of use
PIN Encryption Key	TR31_P0_PIN_ENCRYPTION_KEY	• TDES_2KEY • TDES_3KEY	For IBM3624_NATURAL_PIN, IBM3624_RANDOM_PIN, IBM3624_PIN_FROM_OFFSET • { Encrypt = true, Wrap = true } • { Encrypt = true, Decrypt = true, Wrap = true, Unwrap = true } • { NoRestrictions = true } For IBM3624_PIN_OFFSET

Key Type	Allowed Key Usage	Allowed Key Algorithm	Allowed combination of key modes of use
			• { Encrypt = true, Unwrap = true } • { Encrypt = true, Decrypt = true, Wrap = true, Unwrap = true } • { NoRestrictions = true }
PIN Generation Key	TR31_V1_IBM3624_PIN_VERIFICATION_KEY	• TDES_3KEY	• { Generate = true } • { Generate = true, Verify = true }

VerifyPinData (for VISA/ABA schemes)

`VISA_PIN`

Key Type	Allowed Key Usage	Allowed Key Algorithm	Allowed combination of key modes of use
PIN Encryption Key	TR31_P0_PIN_ENCRYPTION_KEY	• TDES_2KEY • TDES_3KEY	• { Decrypt = true, Unwrap = true } • { Encrypt = true, Decrypt = true, Wrap = true, Unwrap = true } • { NoRestrictions = true }
PIN Generation Key	TR31_V2_VISA_PIN_VERIFICATION_KEY	• TDES_3KEY	• { Verify = true }

Key Type	Allowed Key Usage	Allowed Key Algorithm	Allowed combination of key modes of use
			• { Generate = true, Verify = true }

VerifyPinData (for IBM3624)

IBM3624_PIN_OFFSET,IBM3624_NATURAL_PIN,IBM3624_RANDOM_PIN, IBM3624_PIN_FROM_OFFSET)

Key Type	Allowed Key Usage	Allowed Key Algorithm	Allowed combination of key modes of use
PIN Encryption Key	TR31_P0_PIN_ENCRYPTION_KEY	• TDES_2KEY • TDES_3KEY	For IBM3624_NATURAL_PIN, IBM3624_RANDOM_PIN, IBM3624_PIN_FROM_OFFSET • { Decrypt = true, Unwrap = true } • { Encrypt = true, Decrypt = true, Wrap = true, Unwrap = true } • { NoRestrictions = true }
PIN Verification Key	TR31_V1_IBM3624_PIN_VERIFICATION_KEY	• TDES_3KEY	• { Verify = true } • { Generate = true, Verify = true }

Decrypt Data

Key Type	Allowed Key Usage	Allowed Key Algorithm	Allowed combination of key modes of use
DUKPT	TR31_B0_BASE_DERIVATION_KEY	• TDES_2KEY • AES_128 • AES_192 • AES_256	• { DeriveKey = true } • { NoRestrictions = true }
EMV	TR31_E1_EMV_MKEY_CONFIDENTIALITY TR31_E6_EMV_MKEY_OTHER	• TDES_2KEY	• { DeriveKey = true }
RSA	TR31_D1_ASYMMETRIC_KEY_FOR_DATA_ENCRYPTION	• RSA_2048 • RSA_3072 • RSA_4096	• { Decrypt = true, Unwrap=true} • {Encrypt=true, Wrap=true,Decrypt = true, Unwrap=true}
Symmetric keys	TR31_D0_SYMMETRIC_DATA_ENCRYPTION_KEY	• TDES_2KEY • TDES_3KEY • AES_128 • AES_192 • AES_256	• {Decrypt = true, Unwrap=true} • {Encrypt=true, Wrap=true,Decrypt = true, Unwrap=true} • { NoRestrictions = true }

Encrypt Data

Key Type	Allowed Key Usage	Allowed Key Algorithm	Allowed combination of key modes of use
DUKPT	TR31_B0_BASE_DERIVATION_KEY	• TDES_2KEY • AES_128 • AES_192 • AES_256	• { DeriveKey = true } • { NoRestrictions = true }
EMV	TR31_E1_EMV_MKEY_CONFIDENTIALITY TR31_E6_EMV_MKEY_OTHER	• TDES_2KEY	• { DeriveKey = true }
RSA	TR31_D1_ASYMMETRIC_KEY_FOR_DATA_ENCRYPTION	• RSA_2048 • RSA_3072 • RSA_4096	• { Encrypt = true, Wrap=true} • {Encrypt=true, Wrap=true,Decrypt = true, Unwrap=true}
Symmetric keys	TR31_D0_SYMMETRIC_DATA_ENCRYPTION_KEY	• TDES_2KEY • TDES_3KEY • AES_128 • AES_192 • AES_256	• {Encrypt = true, Wrap=true} • {Encrypt=true, Wrap=true,Decrypt = true, Unwrap=true} • { NoRestrictions = true }

Translate Pin Data

Direction	Key Type	Allowed Key Usage	Allowed Key Algorithm	Allowed combination of key modes of use
Inbound Data Source	DUKPT	TR31_B0_BASE_DERIVATION_KEY	• TDES_2KEY • AES_128 • AES_192 • AES_256	• { DeriveKey = true } • { NoRestrictions = true }
Inbound Data Source	non-DUKPT (PEK, AWK, IWK, etc)	TR31_P0_PIN_ENCRYPTION_KEY	• TDES_2KEY • TDES_3KEY • AES_128 • AES_192 • AES_256	• { Decrypt = true, Unwrap = true } • { Encrypt = true, Decrypt = true, Wrap = true, Unwrap = true } • { NoRestrictions = true }
Outbound Data Target	DUKPT	TR31_B0_BASE_DERIVATION_KEY	• TDES_2KEY • AES_128 • AES_192 • AES_256	• { DeriveKey = true } • { NoRestrictions = true }
Outbound Data Target	non-DUKPT (PEK, IWK, AWK, etc)	TR31_P0_PIN_ENCRYPTION_KEY	• TDES_2KEY • TDES_3KEY • AES_128 • AES_192 • AES_256	• { Encrypt = true, Wrap = true } • { Encrypt = true, Decrypt = true, Wrap =

Direction	Key Type	Allowed Key Usage	Allowed Key Algorithm	Allowed combination of key modes of use
				true, Unwrap = true } • { NoRestrictions = true }

Generate/Verify MAC

MAC keys are used for creating cryptographic hashes of a message/body of data. It is not recommended to create a key with limited key modes of use as you will be unable to perform the matching operation. However, you may import/export a key with only one operation if the other system is intended to perform the other half of the operation pair.

Allowed Key Usage	Allowed Key Usage	Allowed Key Algorithm	Allowed combination of key modes of use
MAC Key	TR31_M1_ISO_9797_1_MAC_KEY	• TDES_2KEY • TDES_3KEY	• { Generate = true } • { Generate = true, Verify = true } • { Verify = true } • { Generate = true }
MAC Key (Retail MAC)	TR31_M1_ISO_9797_3_MAC_KEY	• TDES_2KEY • TDES_3KEY	• { Generate = true } • { Generate = true, Verify = true } • { Verify = true } • { Generate = true }
MAC Key (CMAC)	TR31_M6_ISO_9797_5_CMAC_KEY	• TDES_2KEY • TDES_3KEY	• { Generate = true }

Allowed Key Usage	Allowed Key Usage	Allowed Key Algorithm	Allowed combination of key modes of use
		• AES_128 • AES_192 • AES_256	• { Generate = true, Verify = true } • { Verify = true } • { Generate = true }
MAC Key (HMAC)	TR31_M7_HMAC_KEY	• TDES_2KEY • TDES_3KEY • AES_128 • AES_192 • AES_256	• { Generate = true } • { Generate = true, Verify = true } • { Verify = true }
MAC Key (AS2805)	TR31_M0_ISO_16609_MAC_KEY	• TDES_2KEY • TDES_3KEY	• { Generate = true } • { Generate = true, Verify = true } • { Verify = true }

GenerateMacEmvPinChange

GenerateMacEmvPinChange combines MAC generation and PIN encryption for EMV offline PIN change operations. This operation requires two different key types: an integrity key for MAC generation and a confidentiality key for PIN encryption.

Key Type	Allowed Key Usage	Allowed Key Algorithm	Allowed combination of key modes of use
Secure Messaging Integrity Key	TR31_E2_EMV_MKEY_INTEGRITY	• TDES_2KEY	• { NoRestrictions = true }

Key Type	Allowed Key Usage	Allowed Key Algorithm	Allowed combination of key modes of use
Secure Messaging Confidentiality Key	TR31_E1_EMV_MKEY_CONFIDENTIALITY	• TDES_2KEY	• { DeriveKey = true }
Current PIN PEK (PIN Encryption Key)	TR31_P0_PIN_ENCRYPTION_KEY	• TDES_2KEY • TDES_3KEY • AES_128 • AES_192 • AES_256	• { Decrypt = true, Unwrap = true } • { Encrypt = true, Decrypt = true, Wrap = true, Unwrap = true } • { NoRestrictions = true }
New PIN PEK (PIN Encryption Key)	TR31_P0_PIN_ENCRYPTION_KEY	• TDES_2KEY • TDES_3KEY • AES_128 • AES_192 • AES_256	• { Decrypt = true, Unwrap = true } • { Encrypt = true, Decrypt = true, Wrap = true, Unwrap = true } • { NoRestrictions = true }
ARQC Key **Note** Only applies for Visa and Amex derivation schemes.	TR31_E0_EMV_MKEY_APP_CRYPTOGRAMS	• TDES_2KEY	• { DeriveKey = true }

VerifyAuthRequestCryptogram

Allowed Key Usage	EMV Option	Allowed Key Algorithm	Allowed combination of key modes of use
• OPTION A • OPTION B	TR31_E0_EMV_MKEY_APP_CRYPTOGRAMS	• TDES_2KEY	• { DeriveKey = true }

Import/Export Key

Operation Type	Allowed Key Usage	Allowed Key Algorithm	Allowed combination of key modes of use
TR-31 Wrapping Key	TR31_K1_KEY_BLOCK_PROTECTION_KEY TR31_K0_KEY_ENCRYPTION_KEY	• TDES_2KEY • TDES_3KEY • AES_128 • AES_192 • AES_256	• { Encrypt = true, Wrap = true } (export only) • { Decrypt = true, Unwrap = true } (import only) • { Encrypt = true, Decrypt = true, Wrap = true, Unwrap = true }
Import of trusted CA	TR31_S0_ASYMMETRIC_KEY_FOR_DIGITAL_SIGNATURE	• RSA_2048 • RSA_3072 • RSA_4096	• { Verify = true }
Import of public key certificate for	TR31_D1_ASYMMETRIC	• RSA_2048 • RSA_3072 • RSA_4096	• { Encrypt=true,Wrap=true }

Operation Type	Allowed Key Usage	Allowed Key Algorithm	Allowed combination of key modes of use
asymmetric encryption	_KEY_FOR_ DATA_ENCRYPTION		
Key used to key agreement algorithms such as ECDH	TR31_K3_ASYMMETRIC_KEY_FOR_ KEY_AGREEMENT	• ECC_NIST_P256 • ECC_NIST_P384 • ECC_NIST_P521	• { DeriveKey = true }

Unused key types

The following key types are not currently used by AWS Payment Cryptography

- TR31_P1_PIN_GENERATION_KEY

Common use cases

AWS Payment Cryptography supports many typical payment cryptographic operations. The following topics act as a guide on how to use these operations for typical common use cases. For a list of all commands, please review the AWS Payment Cryptography API.

Topics

- Issuers and issuer processors
- Acquiring and payment facilitators

Issuers and issuer processors

Issuer use cases typically consist of a few parts. This section is organized by function (such as working with pins). In a production system, the keys are typically scoped to a given card bin and are created during bin setup rather than inline as shown here.

Topics

- General Functions
- Network specific functions

General Functions

Topics

- Generate a random pin and the associated PVV and then verify the value
- Generate or verify a CVV for a given card
- Generate or verify a CVV2 for a specific card
- Generate or verify a iCVV for a specific card
- Verify an EMV ARQC and generate an ARPC
- Generate and Verify an EMV MAC
- Generate EMV MAC for PIN Change

Generate a random pin and the associated PVV and then verify the value

Topics

- Create the key(s)
- Generate a random pin, generate PVV and return the encrypted PIN and PVV
- Validate encrypted PIN using PVV method

Create the key(s)

In order to generate a random pin and the PVV, you'll need two keys, a Pin Verification Key(PVK) for generating the PVV and a Pin Encryption Key for encrypting the pin. The pin itself is randomly generated securely inside the service and is not related to either key cryptographically.

The PGK must be a key of algorithm TDES_2KEY based on the PVV algorithm itself. A PEK can be TDES_2KEY, TDES_3KEY or AES_128. In this case, since the PEK is intended for internal use within your system, AES_128 would be a good choice. If a PEK is used for interchange with other systems (e.g. card networks, acquirers, ATMs) or are being moved as part of a migration, TDES_2KEY may be the more appropriate choice for compatibility reasons.

Create the PEK

```
$ aws payment-cryptography create-key \
               --exportable
               --key-attributes
 KeyAlgorithm=AES_128,KeyUsage=TR31_P0_PIN_ENCRYPTION_KEY,\
                   KeyClass=SYMMETRIC_KEY,\
                   KeyModesOfUse='{Encrypt=true,Decrypt=true,Wrap=true,Unwrap=true}' --
tags='[{"Key":"CARD_BIN","Value":"12345678"}]'
```

The response echoes back the request parameters, including an ARN for subsequent calls as well as a Key Check Value (KCV).

```
{
               "Key": {
                   "KeyArn": "arn:aws:payment-cryptography:us-east-2:111122223333:key/
ivi5ksfsuplneuyt",
                   "KeyAttributes": {
                       "KeyUsage": "TR31_P0_PIN_ENCRYPTION_KEY",
                       "KeyClass": "SYMMETRIC_KEY",
```

```
                        "KeyAlgorithm": "AES_128",
                        "KeyModesOfUse": {
                            "Encrypt": false,
                            "Decrypt": false,
                            "Wrap": false,
                            "Unwrap": false,
                            "Generate": true,
                            "Sign": false,
                            "Verify": true,
                            "DeriveKey": false,
                            "NoRestrictions": false
                        }
                    },
                    "KeyCheckValue": "7CC9E2",
                    "KeyCheckValueAlgorithm": "CMAC",
                    "Enabled": true,
                    "Exportable": true,
                    "KeyState": "CREATE_COMPLETE",
                    "KeyOrigin": "AWS_PAYMENT_CRYPTOGRAPHY",
                    "CreateTimestamp": "2023-06-05T06:41:46.648000-07:00",
                    "UsageStartTimestamp": "2023-06-05T06:41:46.626000-07:00"
                }
            }
```

Take note of the `KeyArn` that represents the key, for example *arn:aws:payment-cryptography:us-east-2:111122223333:key/ivi5ksfsuplneuyt*. You need that in the next step.

Create the PVK

```
$ aws payment-cryptography create-key --exportable --key-attributes
 KeyAlgorithm=TDES_2KEY,KeyUsage=TR31_V2_VISA_PIN_VERIFICATION_KEY,KeyClass=SYMMETRIC_KEY,KeyMo
  --tags='[{"Key":"CARD_BIN","Value":"12345678"}]'
```

The response echoes back the request parameters, including an ARN for subsequent calls as well as a Key Check Value (KCV).

```
{
                  "Key": {
                      "KeyArn": "arn:aws:payment-cryptography:us-
east-2:111122223333:key/ov6icy4ryas4zcza",
                      "KeyAttributes": {
                          "KeyUsage": "TR31_V2_VISA_PIN_VERIFICATION_KEY",
```

```
                    "KeyClass": "SYMMETRIC_KEY",
                    "KeyAlgorithm": "TDES_2KEY",
                    "KeyModesOfUse": {
                        "Encrypt": false,
                        "Decrypt": false,
                        "Wrap": false,
                        "Unwrap": false,
                        "Generate": true,
                        "Sign": false,
                        "Verify": true,
                        "DeriveKey": false,
                        "NoRestrictions": false
                    }
                },
                "KeyCheckValue": "51A200",
                "KeyCheckValueAlgorithm": "ANSI_X9_24",
                "Enabled": true,
                "Exportable": true,
                "KeyState": "CREATE_COMPLETE",
                "KeyOrigin": "AWS_PAYMENT_CRYPTOGRAPHY",
                "CreateTimestamp": "2023-06-05T06:41:46.648000-07:00",
                "UsageStartTimestamp": "2023-06-05T06:41:46.626000-07:00"
            }
        }
```


Take note of the `KeyArn` that represents the key, for example *arn:aws:payment-cryptography:us-east-2:111122223333:key/ov6icy4ryas4zcza*. You need that in the next step.

Generate a random pin, generate PVV and return the encrypted PIN and PVV

Example

In this example, we will generate a new (random) 4 digit pin where the outputs will be an encrypted `PIN block` (PinData.PinBlock) and a `PVV` (pinData.VerificationValue). The key inputs are PAN, the `Pin Verification Key`(also known as the pin generation key), the `Pin Encryption Key` and the PIN Block format.

This command requires that the key is of type `TR31_V2_VISA_PIN_VERIFICATION_KEY`.

```
$ aws payment-cryptography-data generate-pin-data --generation-key-identifier
 arn:aws:payment-cryptography:us-east-2:111122223333:key/37y2tsl45p5zjbh2 --encryption-
key-identifier arn:aws:payment-cryptography:us-east-2:111122223333:key/ivi5ksfsuplneuyt
```

```
 --primary-account-number 171234567890123 --pin-block-format ISO_FORMAT_0 --generation-attributes VisaPin={PinVerificationKeyIndex=1}
```

```
{
            "GenerationKeyArn": "arn:aws:payment-cryptography:us-east-2:111122223333:key/37y2tsl45p5zjbh2",
            "GenerationKeyCheckValue": "7F2363",
            "EncryptionKeyArn": "arn:aws:payment-cryptography:us-east-2:111122223333:key/ivi5ksfsuplneuyt",
            "EncryptionKeyCheckValue": "7CC9E2",
            "EncryptedPinBlock": "AC17DC148BDA645E",
            "PinData": {
                "VerificationValue": "5507"
            }
        }
```

Validate encrypted PIN using PVV method

Example

In this example, we will validate a PIN for a given PAN. The PIN is typically provided by the cardholder or user during transaction time for validation and is compared against the value on file (the input from the cardholder is provided as an encrypted value from the terminal or other upstream provider). In order to validate this input, the following values will also be provided at runtime - The encrypted pin, the key used to encrypt the input pin (often referred to as an IWK), PAN and the value to verify against (either a PVV or `PIN offset`).

If AWS Payment Cryptography is able to validate the pin, an http/200 is returned. If the pin is not validated, it will return an http/400.

```
$ aws payment-cryptography-data verify-pin-data --verification-key-identifier
 arn:aws:payment-cryptography:us-east-2:111122223333:key/37y2tsl45p5zjbh2 --encryption-key-identifier arn:aws:payment-cryptography:us-east-2:111122223333:key/ivi5ksfsuplneuyt
 --primary-account-number 171234567890123 --pin-block-format ISO_FORMAT_0  --verification-attributes VisaPin="{PinVerificationKeyIndex=1,VerificationValue=5507}" --encrypted-pin-block AC17DC148BDA645E
```

```
{
        "VerificationKeyArn": "arn:aws:payment-cryptography:us-east-2:111122223333:key/37y2tsl45p5zjbh2",
```

```
        "VerificationKeyCheckValue": "7F2363",
        "EncryptionKeyArn": "arn:aws:payment-cryptography:us-east-2:111122223333:key/
ivi5ksfsuplneuyt",
        "EncryptionKeyCheckValue": "7CC9E2",
}
```

Generate or verify a CVV for a given card

CVV or CVV1 is a value that is traditionally embedded in a cards magnetic stripe. It is not the same as CVV2 (visible to the cardholder and for use for online purchases).

The first step is to create a key. For this tutorial, you create a CVK double-length 3DES (2KEY TDES) key.

Note

CVV, CVV2 and iCVV all use similar if not identical algorithms but vary the input data. All use the same key type TR31_C0_CARD_VERIFICATION_KEY but it is recommended to use separate keys for each purpose. These can be distinguished using aliases and/or tags as in the example below.

Create the key

```
$ aws payment-cryptography create-key --exportable --key-attributes
 KeyAlgorithm=TDES_2KEY,KeyUsage=TR31_C0_CARD_VERIFICATION_KEY,KeyClass=SYMMETRIC_KEY,KeyModesO
 --tags='[{"Key":"KEY_PURPOSE","Value":"CVV"},{"Key":"CARD_BIN","Value":"12345678"}]'
```

The response echoes back the request parameters, including an ARN for subsequent calls as well as a Key Check Value (KCV).

```
{
            "Key": {
                "KeyArn": "arn:aws:payment-cryptography:us-east-2:111122223333:key/
r52o3wbqxyf6qlqr",
                "KeyAttributes": {
                    "KeyUsage": "TR31_C0_CARD_VERIFICATION_KEY",
                    "KeyClass": "SYMMETRIC_KEY",
                    "KeyAlgorithm": "TDES_2KEY",
                    "KeyModesOfUse": {
```

```
                    "Encrypt": false,
                    "Decrypt": false,
                    "Wrap": false,
                    "Unwrap": false,
                    "Generate": true,
                    "Sign": false,
                    "Verify": true,
                    "DeriveKey": false,
                    "NoRestrictions": false
                }
            },
            "KeyCheckValue": "DE89F9",
            "KeyCheckValueAlgorithm": "ANSI_X9_24",
            "Enabled": true,
            "Exportable": true,
            "KeyState": "CREATE_COMPLETE",
            "KeyOrigin": "AWS_PAYMENT_CRYPTOGRAPHY",
            "CreateTimestamp": "2023-06-05T06:41:46.648000-07:00",
            "UsageStartTimestamp": "2023-06-05T06:41:46.626000-07:00"
        }
    }
```

Take note of the `KeyArn` that represents the key, for example *arn:aws:payment-cryptography:us-east-2:111122223333:key/r52o3wbqxyf6qlqr*. You need that in the next step.

Generate a CVV

Example

In this example, we will generate a CVV for a given PAN with inputs of PAN,a service code(as defined by ISO/IEC 7813) of 121 and card expiration date.

For all available parameters see CardVerificationValue1 in the API reference guide.

```
$ aws payment-cryptography-data generate-card-validation-data --key-
identifier arn:aws:payment-cryptography:us-east-2:111122223333:key/
r52o3wbqxyf6qlqr --primary-account-number=171234567890123 --generation-attributes
 CardVerificationValue1='{CardExpiryDate=1127,ServiceCode=121}'
```

```
                  {
```

```
                      "KeyArn": "arn:aws:payment-cryptography:us-
east-2:111122223333:key/r52o3wbqxyf6qlqr",
                      "KeyCheckValue": "DE89F9",
                      "ValidationData": "801"
                  }
```

Validate CVV

Example

In this example, we will verify a CVV for a given PAN with inputs of an CVK, PAN, a service code of 121, card expiration date and the CVV provided during the transaction to validate.

For all available parameters see, CardVerificationValue1 in the API reference guide.

Note

CVV is not a user entered value (like CVV2) but is typically embedded on a magstripe. Consideration should be given to whether it should always validate when provided.

```
$ aws payment-cryptography-data verify-card-validation-data --key-identifier
 arn:aws:payment-cryptography:us-east-2:111122223333:key/r52o3wbqxyf6qlqr
 --primary-account-number=171234567890123 --verification-attributes
 CardVerificationValue1='{CardExpiryDate=1127,ServiceCode=121} --validation-data 801
```

```
{
                    "KeyArn": "arn:aws:payment-cryptography:us-east-2:111122223333:key/
r52o3wbqxyf6qlqr",
                    "KeyCheckValue": "DE89F9",
                    "ValidationData": "801"
}
```

Generate or verify a CVV2 for a specific card

CVV2 is a value that is traditionally provided on the back of a card and is used for online purchases. For virtual cards, it might also be displayed on an app or a screen. Cryptographically, it is the same as CVV1 but with a different service code value.

Create the key

```
$ aws payment-cryptography create-key --exportable --key-attributes
 KeyAlgorithm=TDES_2KEY,KeyUsage=TR31_C0_CARD_VERIFICATION_KEY,KeyClass=SYMMETRIC_KEY,KeyModesO
 --tags='[{"Key":"KEY_PURPOSE","Value":"CVV2"},{"Key":"CARD_BIN","Value":"12345678"}]'
```

The response echoes back the request parameters, including an ARN for subsequent calls as well as a Key Check Value (KCV).

```
{
               "Key": {
                   "KeyArn": "arn:aws:payment-cryptography:us-
east-2:111122223333:key/7f7g4spf3xcklhzu",
                   "KeyAttributes": {
                       "KeyUsage": "TR31_C0_CARD_VERIFICATION_KEY",
                       "KeyClass": "SYMMETRIC_KEY",
                       "KeyAlgorithm": "TDES_2KEY",
                       "KeyModesOfUse": {
                           "Encrypt": false,
                           "Decrypt": false,
                           "Wrap": false,
                           "Unwrap": false,
                           "Generate": true,
                           "Sign": false,
                           "Verify": true,
                           "DeriveKey": false,
                           "NoRestrictions": false
                       }
                   },
                   "KeyCheckValue": "AEA5CD",
                   "KeyCheckValueAlgorithm": "ANSI_X9_24",
                   "Enabled": true,
                   "Exportable": true,
                   "KeyState": "CREATE_COMPLETE",
                   "KeyOrigin": "AWS_PAYMENT_CRYPTOGRAPHY",
                   "CreateTimestamp": "2023-06-05T06:41:46.648000-07:00",
                   "UsageStartTimestamp": "2023-06-05T06:41:46.626000-07:00"
               }
           }
```

Take note of the `KeyArn` that represents the key, for example *arn:aws:payment-cryptography:us-east-2:111122223333:key/7f7g4spf3xcklhzu*. You need that in the next step.

Generate a CVV2

Example

In this example, we will generate a CVV2 for a given PAN with inputs of PAN and card expiration date.

For all available parameters see CardVerificationValue2 in the API reference guide.

```
$ aws payment-cryptography-data generate-card-validation-data --key-identifier
 arn:aws:payment-cryptography:us-east-2:111122223333:key/7f7g4spf3xcklhzu
 --primary-account-number=171234567890123 --generation-attributes
 CardVerificationValue2='{CardExpiryDate=1127}'
```

```
                      {
                          "KeyArn": "arn:aws:payment-cryptography:us-
east-2:111122223333:key/7f7g4spf3xcklhzu",
                          "KeyCheckValue": "AEA5CD",
                          "ValidationData": "321"
                      }
```

Validate a CVV2

Example

In this example, we will verify a CVV2 for a given PAN with inputs of an CVK, PANand card expiration date and the CVV provided during the transaction to validate.

For all available parameters see, CardVerificationValue2 in the API reference guide.

Note

CVV2 and the other inputs are user entered values. As such, it is not necessarily a sign of an issue that this periodically fails to validate.

```
$ aws payment-cryptography-data verify-card-validation-data --key-identifier
 arn:aws:payment-cryptography:us-east-2:111122223333:key/7f7g4spf3xcklhzu
```

```
 --primary-account-number=171234567890123 --verification-attributes
 CardVerificationValue2='{CardExpiryDate=1127} --validation-data 321
```

```
{
                       "KeyArn": "arn:aws:payment-cryptography:us-
east-2:111122223333:key/7f7g4spf3xcklhzu",
                       "KeyCheckValue": "AEA5CD",
                       "ValidationData": "801"
   }
```

Generate or verify a iCVV for a specific card

iCVV uses the same algorithm as CVV/CVV2 but iCVV is embedded inside a chip card. Its service code is 999.

Create the key

```
$ aws payment-cryptography create-key --exportable --key-attributes
 KeyAlgorithm=TDES_2KEY,KeyUsage=TR31_C0_CARD_VERIFICATION_KEY,KeyClass=SYMMETRIC_KEY,KeyModesO
 --tags='[{"Key":"KEY_PURPOSE","Value":"ICVV"},{"Key":"CARD_BIN","Value":"12345678"}]'
```

The response echoes back the request parameters, including an ARN for subsequent calls as well as a Key Check Value (KCV).

```
{
               "Key": {
                   "KeyArn": "arn:aws:payment-cryptography:us-east-2:111122223333:key/
c7dsi763r6s7lfp3",
                   "KeyAttributes": {
                       "KeyUsage": "TR31_C0_CARD_VERIFICATION_KEY",
                       "KeyClass": "SYMMETRIC_KEY",
                       "KeyAlgorithm": "TDES_2KEY",
                       "KeyModesOfUse": {
                           "Encrypt": false,
                           "Decrypt": false,
                           "Wrap": false,
                           "Unwrap": false,
                           "Generate": true,
                           "Sign": false,
```

```
                    "Verify": true,
                    "DeriveKey": false,
                    "NoRestrictions": false
                }
            },
            "KeyCheckValue": "1201FB",
            "KeyCheckValueAlgorithm": "ANSI_X9_24",
            "Enabled": true,
            "Exportable": true,
            "KeyState": "CREATE_COMPLETE",
            "KeyOrigin": "AWS_PAYMENT_CRYPTOGRAPHY",
            "CreateTimestamp": "2023-06-05T06:41:46.648000-07:00",
            "UsageStartTimestamp": "2023-06-05T06:41:46.626000-07:00"
        }
    }
```

Take note of the `KeyArn` that represents the key, for example *arn:aws:payment-cryptography:us-east-2:111122223333:key/c7dsi763r6s7lfp3*. You need that in the next step.

Generate a iCVV

Example

In this example, we will generate a iCVV for a given PAN with inputs of PAN,a service code(as defined by ISO/IEC 7813) of 999 and card expiration date.

For all available parameters see CardVerificationValue1 in the API reference guide.

```
$ aws payment-cryptography-data generate-card-validation-data --key-
identifier arn:aws:payment-cryptography:us-east-2:111122223333:key/
c7dsi763r6s7lfp3 --primary-account-number=171234567890123 --generation-attributes
 CardVerificationValue1='{CardExpiryDate=1127,ServiceCode=999}'
```

```
                      {
                          "KeyArn": "arn:aws:payment-cryptography:us-
east-2:111122223333:key/c7dsi763r6s7lfp3",
                          "KeyCheckValue": "1201FB",
                          "ValidationData": "532"
                      }
```

Validate iCVV

Example

For validation, the inputs are CVK, PAN, a service code of 999, card expiration date and the iCVV provided during the transaction to validate.

For all available parameters see, CardVerificationValue1 in the API reference guide.

Note

iCVV is not a user entered value (like CVV2) but is typically embedded on an EMV/chip card. Consideration should be given to whether it should always validate when provided.

```
$ aws payment-cryptography-data verify-card-validation-data --key-identifier
 arn:aws:payment-cryptography:us-east-2:111122223333:key/c7dsi763r6s7lfp3
 --primary-account-number=171234567890123 --verification-attributes
 CardVerificationValue1='{CardExpiryDate=1127,ServiceCode=999} --validation-data 532
```

```
{
                       "KeyArn": "arn:aws:payment-cryptography:us-
east-2:111122223333:key/c7dsi763r6s7lfp3",
                       "KeyCheckValue": "1201FB",
                       "ValidationData": "532"
   }
```

Verify an EMV ARQC and generate an ARPC

ARQC (Authorization Request Cryptogram) is a cryptogram generated by an EMV (chip) card and used to validate the transaction details as well as the use of an authorized card. It incorporates data from the card, terminal and the transaction itself.

At validation time on the backend, the same inputs are provided to AWS Payment Cryptography, the cryptogram is internally re-created and this is compared against the value provided with the transaction. In this sense, it is similar to a MAC. EMV 4.4 Book 2 defines three aspects of this function - key derivation methods (known as common session key - CSK) to generate one-time transaction keys, a minimum payload and methods for generating a response (ARPC).

Individual card schemes may specify additional transactional fields to incorporate or the order those fields appear. Other (generally deprecated) scheme specific derivation schemes exist as well and are covered elsewhere in this documentation.

For more information, see VerifyCardValidationData in the API guide.

Create the key

```
$ aws payment-cryptography create-key --exportable --key-attributes
 KeyAlgorithm=TDES_2KEY,KeyUsage=TR31_E0_EMV_MKEY_APP_CRYPTOGRAMS,KeyClass=SYMMETRIC_KEY,KeyMod
 --tags='[{"Key":"KEY_PURPOSE","Value":"CVN18"},{"Key":"CARD_BIN","Value":"12345678"}]'
```

The response echoes back the request parameters, including an ARN for subsequent calls as well as a Key Check Value (KCV).

```
{
                "Key": {
                    "KeyArn": "arn:aws:payment-cryptography:us-east-2:111122223333:key/
pw3s6nl62t5ushfk",
                    "KeyAttributes": {
                        "KeyUsage": "TR31_E0_EMV_MKEY_APP_CRYPTOGRAMS",
                        "KeyClass": "SYMMETRIC_KEY",
                        "KeyAlgorithm": "TDES_2KEY",
                        "KeyModesOfUse": {
                            "Encrypt": false,
                            "Decrypt": false,
                            "Wrap": false,
                            "Unwrap": false,
                            "Generate": false,
                            "Sign": false,
                            "Verify": false,
                            "DeriveKey": true,
                            "NoRestrictions": false
                        }
                    },
                    "KeyCheckValue": "08D7B4",
                    "KeyCheckValueAlgorithm": "ANSI_X9_24",
                    "Enabled": true,
                    "Exportable": true,
                    "KeyState": "CREATE_COMPLETE",
                    "KeyOrigin": "AWS_PAYMENT_CRYPTOGRAPHY",
                    "CreateTimestamp": "2024-03-07T06:41:46.648000-07:00",
                    "UsageStartTimestamp": "2024-03-07T06:41:46.626000-07:00"
```

```
                }
            }
```

Take note of the `KeyArn` that represents the key, for example *arn:aws:payment-cryptography:us-east-2:111122223333:key/pw3s6nl62t5ushfk*. You need that in the next step.

Generate an ARQC

The ARQC is generated exclusively by an EMV card. As such, AWS Payment Cryptography has no facility for generating such a payload. For test purposes, a number of libraries are available online that can generate an appropriate payload as well as known values that are generally provided by the various schemes.

Validate an ARQC

Example

If AWS Payment Cryptography is able to validate the ARQC, an http/200 is returned. An ARPC (response) can optionally be provided and in included in the response after the ARQC is validated.

```
$ aws payment-cryptography-data verify-auth-request-cryptogram
 --auth-request-cryptogram 61EDCC708B4C97B4 --key-identifier
 arn:aws:payment-cryptography:us-east-2:111122223333:key/pw3s6nl62t5ushfk
 --major-key-derivation-mode EMV_OPTION_A --transaction-data
 00000000170000000000000008400080008000084016051700000000093800000B1F22010300000000000000000000
 --session-key-derivation-attributes='{"EmvCommon":
{"ApplicationTransactionCounter":"000B",
 "PanSequenceNumber":"01","PrimaryAccountNumber":"9137631040001422"}}' --auth-response-
attributes='{"ArpcMethod2":{"CardStatusUpdate":"12345678"}}'
```

```
{
    "KeyArn": "arn:aws:payment-cryptography:us-east-2:111122223333:key/
pw3s6nl62t5ushfk",
    "KeyCheckValue": "08D7B4",
    "AuthResponseValue":"2263AC85"
}
```

Generate and Verify an EMV MAC

EMV MAC is MAC using an input of an EMV derived key and then performing a ISO9797-3 (Retail) MAC over the resulting data. EMV MAC is typically used to send commands to an EMV card such as unblock scripts.

Note

AWS Payment Cryptography does not validate the contents of the script. Please consult your scheme or card manual for details on specific commands to include.

For more information, see MacAlgorithmEmv in the API guide.

Topics

- Create the key
- Generate an EMV MAC

Create the key

```
$ aws payment-cryptography create-key --exportable --key-attributes
 KeyAlgorithm=TDES_2KEY,KeyUsage=TR31_E2_EMV_MKEY_INTEGRITY,KeyClass=SYMMETRIC_KEY,KeyModesOfUs
 --tags='[{"Key":"KEY_PURPOSE","Value":"CVN18"},{"Key":"CARD_BIN","Value":"12345678"}]'
```

The response echoes back the request parameters, including an ARN for subsequent calls as well as a Key Check Value (KCV).

```
{
        "Key": {
            "KeyArn": "arn:aws:payment-cryptography:us-east-2:111122223333:key/
pw3s6nl62t5ushfk",
            "KeyAttributes": {
                "KeyUsage": "TR31_E2_EMV_MKEY_INTEGRITY",
                "KeyClass": "SYMMETRIC_KEY",
                "KeyAlgorithm": "TDES_2KEY",
                "KeyModesOfUse": {
                    "Encrypt": false,
                    "Decrypt": false,
                    "Wrap": false,
                    "Unwrap": false,
                    "Generate": false,
                    "Sign": false,
                    "Verify": false,
                    "DeriveKey": true,
                    "NoRestrictions": false
                }
```

```
            },
            "KeyCheckValue": "08D7B4",
            "KeyCheckValueAlgorithm": "ANSI_X9_24",
            "Enabled": true,
            "Exportable": true,
            "KeyState": "CREATE_COMPLETE",
            "KeyOrigin": "AWS_PAYMENT_CRYPTOGRAPHY",
            "CreateTimestamp": "2024-03-07T06:41:46.648000-07:00",
            "UsageStartTimestamp": "2024-03-07T06:41:46.626000-07:00"
        }
    }
```

Take note of the `KeyArn` that represents the key, for example *arn:aws:payment-cryptography:us-east-2:111122223333:key/pw3s6nl62t5ushfk*. You need that in the next step.

Generate an EMV MAC

The typical flow is that a backend process will generate an EMV script (such as card unblock), sign it using this command (which derives a one-time key specific to one particular card) and then return the MAC. Then the command + MAC are sent to the card to be applied. Sending the command to the card is outside the scope of AWS Payment Cryptography.

Note

This command is meant for commands when no encrypted data (such as PIN) is sent. EMV Encrypt can be combined with this command to append encrypted data to the issuer script prior to calling this command

Message Data

Message data includes the APDU header and command. While this can vary by implementation, this example is the APDU header for unblock (84 24 00 00 08), following by ATC (0007) and then ARQC of the previous transaction (999E57FD0F47CACE). The service does not validate the contents of this field.

Session Key Derivation Mode

This field defines how the session key is generated. EMV_COMMON_SESSION_KEY is generally used for the new implementations, while EMV2000 | AMEX | MASTERCARD_SESSION_KEY | VISA may be used as well.

MajorKeyDerivationMode

EMV Defines Mode A, B or C. Mode A is the most common and AWS Payment Cryptography currently supports mode A or mode B.

PAN

The account number, typically available in chip field 5A or ISO8583 field 2 but may also be retrieved from the card system.

PSN

The card sequence number. If not used, enter 00.

SessionKeyDerivationValue

This is the per session derivation data. It can either be the last ARQC(ApplicationCryptogram) from field 9F26 or the last ATC from 9F36 depending on the derivation scheme.

Padding

Padding is automatically applied and uses ISO/IEC 9797-1 padding method 2.

Example

```
$ aws payment-cryptography-data generate-mac --message-data
 84240000080007999E57FD0F47CACE --key-identifier arn:aws:payment-
cryptography:us-east-2:111122223333:key/pw3s6nl62t5ushfk --message-
data 8424000008999E57FD0F47CACE0007 --generation-attributes
 EmvMac="{MajorKeyDerivationMode=EMV_OPTION_A,PanSequenceNumber='00',PrimaryAccountNumber='2235
```

```
{
"KeyArn": "arn:aws:payment-cryptography:us-east-2:111122223333:key/pw3s6nl62t5ushfk",
"KeyCheckValue": "08D7B4",
"Mac":"5652EEDF83EA0D84"
}
```

Generate EMV MAC for PIN Change

EMV PIN change combines two operations: generating a MAC for an issuer script and encrypting a new PIN for offline PIN change on an EMV chip card. This command is only needed in certain countries where the pin is stored on the chip card (this is common for European countries). This is

commonly used when a cardholder needs to change their PIN and the new PIN must be securely transmitted to the card along with a MAC to verify the command's authenticity.

Note

If you only need to send commands to the card but not change the PIN, consider using ARPC CSU or Generate EMV MAC commands instead.

For more information, see GenerateMacEmvPinChange in the API guide.

Generate EMV MAC and encrypted PIN for PIN change

This operation requires two keys: an EMV integrity key (KeyUsage: TR31_E2_EMV_MKEY_INTEGRITY) for MAC generation and an EMV confidentiality key (KeyUsage: TR31_E4_EMV_MKEY_CONFIDENTIALITY) for PIN encryption. The typical flow is that a backend process will generate an EMV PIN change script, which includes both the MAC for the issuer script and the encrypted new PIN. The command and encrypted PIN are then sent to the card to update the offline PIN. Sending the command to the card is outside the scope of AWS Payment Cryptography.

Message Data

Message data includes the APDU command for the issuer script. The service does not validate the contents of this field.

New Encrypted PIN Block

The new encrypted PIN block that will be sent to the card. This must be provided as an encrypted value using a PIN encryption key.

New PIN PEK Identifier

The key used to encrypt the new PIN before it's passed to this API.

Secure Messaging Integrity Key

The EMV integrity key (KeyUsage: TR31_E2_EMV_MKEY_INTEGRITY) used for MAC generation.

Secure Messaging Confidentiality Key

The EMV confidentiality key (KeyUsage: TR31_E4_EMV_MKEY_CONFIDENTIALITY) used for PIN encryption.

MajorKeyDerivationMode

EMV defines Mode A, B, or C. Mode A is the most common and AWS Payment Cryptography currently supports mode A or mode B.

Mode

The encryption mode, typically CBC for PIN change operations.

PAN

The account number, typically available in chip field 5A or ISO8583 field 2 but may also be retrieved from the card system.

PanSequenceNumber

The card sequence number. If not used, enter 00.

ApplicationCryptogram

This is the per session derivation data, typically the last ARQC from field 9F26.

PinBlockLengthPosition

Specifies where the PIN block length is encoded. Typically set to NONE. Check your card scheme specifications if you're not sure.

PinBlockPaddingType

Specifies the padding type for the PIN block. Typically set to NO_PADDING. Check your card scheme specifications if you're not sure.

Example

```
$ aws payment-cryptography-data generate-mac-emv-pin-change \
    --message-data 00A4040008A000000004101080D80500000001010A04000000000000 \
    --new-encrypted-pin-block 67FB27C75580EFE7 \
    --new-pin-pek-identifier arn:aws:payment-cryptography:us-east-2:111122223333:key/
ivi5ksfsuplneuyt \
    --pin-block-format ISO_FORMAT_0 \
    --secure-messaging-confidentiality-key-identifier arn:aws:payment-cryptography:us-
east-2:111122223333:key/tqv5yij6wtxx64pi \
    --secure-messaging-integrity-key-identifier arn:aws:payment-cryptography:us-
east-2:111122223333:key/pw3s6nl62t5ushfk \
```

```
    --derivation-method-attributes
 'EmvCommon={ApplicationCryptogram=1234567890123457,MajorKeyDerivationMode=EMV_OPTION_A,Mode=CB
```

```
{
    "SecureMessagingIntegrityKeyArn": "arn:aws:payment-cryptography:us-
east-2:111122223333:key/pw3s6nl62t5ushfk",
    "SecureMessagingIntegrityKeyCheckValue": "08D7B4",
    "SecureMessagingConfidentialityKeyArn": "arn:aws:payment-cryptography:us-
east-2:111122223333:key/tqv5yij6wtxx64pi",
    "SecureMessagingConfidentialityKeyCheckValue": "C1EB8F",
    "Mac": "5652EEDF83EA0D84",
    "EncryptedPinBlock": "F1A2B3C4D5E6F7A8"
}
```

Network specific functions

Topics

- Visa specific functions
- Mastercard specific functions
- American Express specific functions
- JCB specific functions

Visa specific functions

Topics

- ARQC - CVN18/CVN22
- ARQC - CVN10
- 3DS CAVV V7
- dCVV (Dynamic Card Verification Value) - CVN17

ARQC - CVN18/CVN22

CVN18 and CVN22 utilize the CSK method of key derivation. The exact transaction data varies between these two methods - please see the scheme documentation for details on constructing the transaction data field.

ARQC - CVN10

CVN10 is an older Visa method for EMV transactions that uses per card key derivation rather than session (per transaction) derivation and also uses a different payload. For information on the payload contents, please contact the scheme for details.

Create key

```
$ aws payment-cryptography create-key --exportable --key-attributes
 KeyAlgorithm=TDES_2KEY,KeyUsage=TR31_E0_EMV_MKEY_APP_CRYPTOGRAMS,KeyClass=SYMMETRIC_KEY,KeyMod
 --tags='[{"Key":"KEY_PURPOSE","Value":"CVN10"},{"Key":"CARD_BIN","Value":"12345678"}]'
```

The response echoes back the request parameters, including an ARN for subsequent calls as well as a Key Check Value (KCV).

```
{
            "Key": {
                "KeyArn": "arn:aws:payment-cryptography:us-east-2:111122223333:key/
pw3s6nl62t5ushfk",
                "KeyAttributes": {
                    "KeyUsage": "TR31_E0_EMV_MKEY_APP_CRYPTOGRAMS",
                    "KeyClass": "SYMMETRIC_KEY",
                    "KeyAlgorithm": "TDES_2KEY",
                    "KeyModesOfUse": {
                        "Encrypt": false,
                        "Decrypt": false,
                        "Wrap": false,
                        "Unwrap": false,
                        "Generate": false,
                        "Sign": false,
                        "Verify": false,
                        "DeriveKey": true,
                        "NoRestrictions": false
                    }
                },
                "KeyCheckValue": "08D7B4",
                "KeyCheckValueAlgorithm": "ANSI_X9_24",
                "Enabled": true,
                "Exportable": true,
                "KeyState": "CREATE_COMPLETE",
                "KeyOrigin": "AWS_PAYMENT_CRYPTOGRAPHY",
                "CreateTimestamp": "2024-03-07T06:41:46.648000-07:00",
```

```
                "UsageStartTimestamp": "2024-03-07T06:41:46.626000-07:00"
            }
        }
```

Take note of the `KeyArn` that represents the key, for example *arn:aws:payment-cryptography:us-east-2:111122223333:key/pw3s6nl62t5ushfk*. You need that in the next step.

Validate the ARQC

Example

In this example, we will validate an ARQC generated using Visa CVN10.

If AWS Payment Cryptography is able to validate the ARQC, an http/200 is returned. If the arqc is not validated, it will return a http/400 response.

```
$ aws payment-cryptography-data verify-auth-request-cryptogram --auth-request-cryptogram D791093C8A921769 \
  --key-identifier arn:aws:payment-cryptography:us-east-2:111122223333:key/pw3s6nl62t5ushfk \
  --major-key-derivation-mode EMV_OPTION_A \
  --transaction-data
 00000000170000000000000008400080008000084016051700000000093800000B03011203000000 \
  --session-key-derivation-attributes='{"Visa":{"PanSequenceNumber":"01" \
  ,"PrimaryAccountNumber":"9137631040001422"}}'
```

```
{
    "KeyArn": "arn:aws:payment-cryptography:us-east-2:111122223333:key/pw3s6nl62t5ushfk",
    "KeyCheckValue": "08D7B4"
  }
```

3DS CAVV V7

For Visa Secure (3DS) transactions, a CAVV (Cardholder Authentication Verification Value) is generated by the issuer Access Control Server (ACS). The CAVV is evidence that cardholder authentication occurred, is unique for each authentication transaction and is provided by the acquirer in the authorization message. CAVV v7 binds additional data about the transaction to the approval including elements such as merchant name, purchase amount and purchase date. In this way, it is effectively a cryptographic hash of the transaction payload.

Cryptographically, CAVV V7 utilizes the CVV algorithm but the inputs have all been changed/repurposed. Please consult appropriate third party/Visa documentation on how to produce the inputs to generate a CAVV V7 payload.

Create the key

```
$ aws payment-cryptography create-key --exportable --key-attributes
 KeyAlgorithm=TDES_2KEY,KeyUsage=TR31_C0_CARD_VERIFICATION_KEY,KeyClass=SYMMETRIC_KEY,KeyModesO
 --tags='[{"Key":"KEY_PURPOSE","Value":"CAVV-V7"},
{"Key":"CARD_BIN","Value":"12345678"}]'
```

The response echoes back the request parameters, including an ARN for subsequent calls as well as a Key Check Value (KCV).

```
{
            "Key": {
                "KeyArn": "arn:aws:payment-cryptography:us-east-2:111122223333:key/
dnaeyrjgdjjtw6dk",
                "KeyAttributes": {
                    "KeyUsage": "TR31_C0_CARD_VERIFICATION_KEY",
                    "KeyClass": "SYMMETRIC_KEY",
                    "KeyAlgorithm": "TDES_2KEY",
                    "KeyModesOfUse": {
                        "Encrypt": false,
                        "Decrypt": false,
                        "Wrap": false,
                        "Unwrap": false,
                        "Generate": true,
                        "Sign": false,
                        "Verify": true,
                        "DeriveKey": false,
                        "NoRestrictions": false
                    }
                },
                "KeyCheckValue": "F3FB13",
                "KeyCheckValueAlgorithm": "ANSI_X9_24",
                "Enabled": true,
                "Exportable": true,
                "KeyState": "CREATE_COMPLETE",
                "KeyOrigin": "AWS_PAYMENT_CRYPTOGRAPHY",
                "CreateTimestamp": "2023-06-05T06:41:46.648000-07:00",
                "UsageStartTimestamp": "2023-06-05T06:41:46.626000-07:00"
```

```
            }
        }
```

Take note of the `KeyArn` that represents the key, for example *arn:aws:payment-cryptography:us-east-2:111122223333:key/dnaeyrjgdjjtw6dk*. You need that in the next step.

Generate a CAVV V7

Example

In this example, we will generate a CAVV V7 for a given transactions with inputs as specified in the specifications. Note that for this algorithm, fields may be re-used/re-purposed, so it should not be assumed that the field labels match the inputs.

For all available parameters see CardVerificationValue1 in the API reference guide.

```
$ aws payment-cryptography-data generate-card-validation-data --key-identifier arn:aws:payment-cryptography:us-east-2:111122223333:key/dnaeyrjgdjjtw6dk --primary-account-number=171234567890123 --generation-attributes CardVerificationValue1='{CardExpiryDate=9431,ServiceCode=431}'
```

```
              {
                  "KeyArn": "arn:aws:payment-cryptography:us-east-2:111122223333:key/dnaeyrjgdjjtw6dk",
                  "KeyCheckValue": "F3FB13",
                  "ValidationData": "491"
              }
```

Validate CAVV V7

Example

For validation, the inputs are CVK, the computed input values and the CAVV provided during the transaction to validate.

For all available parameters see, CardVerificationValue1 in the API reference guide.

Note

CAVV is not a user entered value (like CVV2) but is calculated by the issuer ACS. Consideration should be given to whether it should always validate when provided.

```
$ aws payment-cryptography-data verify-card-validation-data --key-identifier
 arn:aws:payment-cryptography:us-east-2:111122223333:key/dnaeyrjgdjjtw6dk
 --primary-account-number=171234567890123 --verification-attributes
 CardVerificationValue1='{CardExpiryDate=9431,ServiceCode=431} --validation-data 491
```

```
{
                "KeyArn": "arn:aws:payment-cryptography:us-east-2:111122223333:key/
dnaeyrjgdjjtw6dk",
                "KeyCheckValue": "F3FB13",
                "ValidationData": "491"
}
```

dCVV (Dynamic Card Verification Value) - CVN17

dCVV (dynamic Card Verification Value) is a Visa-specific dynamic cryptogram used for contactless EMV transactions. It is known as early EMV and it provides enhanced security by generating a unique verification value for each transaction. The dCVV uses inputs including the Primary Account Number (PAN), PAN Sequence Number (PSN), Application Transaction Counter (ATC), unpredictable number, and track data. It is still used in some places, but has mostly been replaced by other algorithms like CVN18.

For all available parameters see DynamicCardVerificationValue in the API reference guide.

Create key

```
$ aws payment-cryptography create-key --exportable --key-attributes
 KeyAlgorithm=TDES_2KEY,KeyUsage=TR31_E4_EMV_MKEY_DYNAMIC_NUMBERS,KeyClass=SYMMETRIC_KEY,KeyMod
 --tags='[{"Key":"KEY_PURPOSE","Value":"DCVV"},{"Key":"CARD_BIN","Value":"12345678"}]'
```

The response echoes back the request parameters, including an ARN for subsequent calls as well as a Key Check Value (KCV).

```
{
```

```
            "Key": {
                "KeyArn": "arn:aws:payment-cryptography:us-east-2:111122223333:key/
mw7dn3qxvkfh8ztc",
                "KeyAttributes": {
                    "KeyUsage": "TR31_E4_EMV_MKEY_DYNAMIC_NUMBERS",
                    "KeyClass": "SYMMETRIC_KEY",
                    "KeyAlgorithm": "TDES_2KEY",
                    "KeyModesOfUse": {
                        "Encrypt": false,
                        "Decrypt": false,
                        "Wrap": false,
                        "Unwrap": false,
                        "Generate": true,
                        "Sign": false,
                        "Verify": true,
                        "DeriveKey": false,
                        "NoRestrictions": false
                    }
                },
                "KeyCheckValue": "A8E4D2",
                "KeyCheckValueAlgorithm": "ANSI_X9_24",
                "Enabled": true,
                "Exportable": true,
                "KeyState": "CREATE_COMPLETE",
                "KeyOrigin": "AWS_PAYMENT_CRYPTOGRAPHY",
                "CreateTimestamp": "2025-02-02T11:45:30.648000-08:00",
                "UsageStartTimestamp": "2025-02-02T11:45:30.626000-08:00"
            }
        }
```


Take note of the `KeyArn` that represents the key, for example *arn:aws:payment-cryptography:us-east-2:111122223333:key/mw7dn3qxvkfh8ztc*. You need that in the next step.

Generate a dCVV

Example

In this example, we will generate a dCVV for a contactless EMV transaction. The inputs include the PAN, PAN Sequence Number, Application Transaction Counter, unpredictable number, and track data.

```
$ aws payment-cryptography-data generate-card-validation-data --key-identifier
 arn:aws:payment-cryptography:us-east-2:111122223333:key/mw7dn3qxvkfh8ztc \
```

```
    --primary-account-number=5111112627662122 \
    --generation-attributes
 DynamicCardVerificationValue='{ApplicationTransactionCounter=01,PanSequenceNumber=00,TrackData
 \
    --validation-data-length 5
```

```
{
    "KeyArn": "arn:aws:payment-cryptography:us-east-2:111122223333:key/
mw7dn3qxvkfh8ztc",
    "KeyCheckValue": "A8E4D2",
    "ValidationData": "36667"
}
```

Validate dCVV

Example

In this example, we will validate a dCVV provided during a transaction. The same inputs used for generation must be provided for validation.

If AWS Payment Cryptography is able to validate, an http/200 is returned. If the value is not validated, it will return a http/400 response.

```
$ aws payment-cryptography-data verify-card-validation-data --key-identifier
 arn:aws:payment-cryptography:us-east-2:111122223333:key/mw7dn3qxvkfh8ztc \
    --primary-account-number=5111112627662122 \
    --validation-data=36667 \
    --verification-attributes
 DynamicCardVerificationValue='{ApplicationTransactionCounter=01,PanSequenceNumber=00,TrackData
```

```
{
    "KeyArn": "arn:aws:payment-cryptography:us-east-2:111122223333:key/
mw7dn3qxvkfh8ztc",
    "KeyCheckValue": "A8E4D2"
}
```

Mastercard specific functions

Topics

- DCVC3

- ARQC - CVN14/CVN15
- ARQC - CVN12/CVN13
- 3DS SPA2 AAV

DCVC3

DCVC3 predates EMV CSK and Mastercard CVN12 schemes and represents another approach for utilizing dynamic keys. It is sometimes repurposed for other use cases as well. In this scheme, the inputs are PAN, PSN, Track1/Track2 data, an unpredictable number and transaction counter (ATC).

Create key

```
$ aws payment-cryptography create-key --exportable --key-attributes
 KeyAlgorithm=TDES_2KEY,KeyUsage=TR31_E4_EMV_MKEY_DYNAMIC_NUMBERS,KeyClass=SYMMETRIC_KEY,KeyMod
 --tags='[{"Key":"KEY_PURPOSE","Value":"DCVC3"},{"Key":"CARD_BIN","Value":"12345678"}]'
```

The response echoes back the request parameters, including an ARN for subsequent calls as well as a Key Check Value (KCV).

```
{
            "Key": {
                "KeyArn": "arn:aws:payment-cryptography:us-east-2:111122223333:key/
hrh6qgbi3sk4y3wq",
                "KeyAttributes": {
                    "KeyUsage": "TR31_E4_EMV_MKEY_DYNAMIC_NUMBERS",
                    "KeyClass": "SYMMETRIC_KEY",
                    "KeyAlgorithm": "TDES_2KEY",
                    "KeyModesOfUse": {
                        "Encrypt": false,
                        "Decrypt": false,
                        "Wrap": false,
                        "Unwrap": false,
                        "Generate": false,
                        "Sign": false,
                        "Verify": false,
                        "DeriveKey": true,
                        "NoRestrictions": false
                    }
                },
                "KeyCheckValue": "08D7B4",
                "KeyCheckValueAlgorithm": "ANSI_X9_24",
```

```
                "Enabled": true,
                "Exportable": true,
                "KeyState": "CREATE_COMPLETE",
                "KeyOrigin": "AWS_PAYMENT_CRYPTOGRAPHY",
                "CreateTimestamp": "2024-03-07T06:41:46.648000-07:00",
                "UsageStartTimestamp": "2024-03-07T06:41:46.626000-07:00"
            }
        }
```

Take note of the `KeyArn` that represents the key, for example *arn:aws:payment-cryptography:us-east-2:111122223333:key/hrh6qgbi3sk4y3wq*. You need that in the next step.

Generate a DCVC3

Example

Although DCVC3 is typically generated by a chip card, it can also be manually generated such as in this example

```
$ aws payment-cryptography-data generate-card-validation-data --key-identifier
 arn:aws:payment-cryptography:us-east-2:111122223333:key/pw3s6nl62t5ushfk
 --primary-account-number=5413123456784808 --generation-attributes
 DynamicCardVerificationCode='{ApplicationTransactionCounter=0000,TrackData=5241060000000069D13
```


```
{
    "KeyArn": "arn:aws:payment-cryptography:us-east-2:111122223333:key/
pw3s6nl62t5ushfk",
    "KeyCheckValue": "08D7B4",
    "ValidationData": "865"
  }
```


Validate the DCVC3

Example

In this example, we will validate an DCVC3. Note that ATC should be provided as a hex number for instance a counter of 11 should be represented as 000B. The service expects a 3 digit DCVC3, so if you have stored a 4(or 5) digit value, simply truncate the left characters until you have 3 digits (for instance 15321 should result in validation-data value of 321).

If AWS Payment Cryptography is able to validate, an http/200 is returned. If the value is not validated, it will return a http/400 response.

```
$ aws payment-cryptography-data verify-card-validation-data  --key-identifier
 arn:aws:payment-cryptography:us-east-2:111122223333:key/pw3s6nl62t5ushfk
 --primary-account-number=5413123456784808 --verification-attributes
 DynamicCardVerificationCode='{ApplicationTransactionCounter=000B,TrackData=5241060000000069D13
 --validation-data 398
```

```
{
    "KeyArn": "arn:aws:payment-cryptography:us-east-2:111122223333:key/
pw3s6nl62t5ushfk",
    "KeyCheckValue": "08D7B4"
  }
```

ARQC - CVN14/CVN15

CVN14 and CVN15 utilize the EMV CSK method of key derivation. The exact transaction data varies between these two methods - please see the scheme documentation for details on constructing the transaction data field.

ARQC - CVN12/CVN13

CVN12 and CVN13 are older Mastercard-specific method for EMV transactions that incorporates an unpredictable number into the per-transaction derivation and also uses a different payload. For information on the payload contents, please contact the scheme.

Create key

```
$ aws payment-cryptography create-key --exportable --key-attributes
 KeyAlgorithm=TDES_2KEY,KeyUsage=TR31_E0_EMV_MKEY_APP_CRYPTOGRAMS,KeyClass=SYMMETRIC_KEY,KeyMod
 --tags='[{"Key":"KEY_PURPOSE","Value":"CVN12"},{"Key":"CARD_BIN","Value":"12345678"}]'
```

The response echoes back the request parameters, including an ARN for subsequent calls as well as a Key Check Value (KCV).

```
{
            "Key": {
                "KeyArn": "arn:aws:payment-cryptography:us-east-2:111122223333:key/
pw3s6nl62t5ushfk",
                "KeyAttributes": {
                    "KeyUsage": "TR31_E0_EMV_MKEY_APP_CRYPTOGRAMS",
                    "KeyClass": "SYMMETRIC_KEY",
                    "KeyAlgorithm": "TDES_2KEY",
```

```
                    "KeyModesOfUse": {
                        "Encrypt": false,
                        "Decrypt": false,
                        "Wrap": false,
                        "Unwrap": false,
                        "Generate": false,
                        "Sign": false,
                        "Verify": false,
                        "DeriveKey": true,
                        "NoRestrictions": false
                    }
                },
                "KeyCheckValue": "08D7B4",
                "KeyCheckValueAlgorithm": "ANSI_X9_24",
                "Enabled": true,
                "Exportable": true,
                "KeyState": "CREATE_COMPLETE",
                "KeyOrigin": "AWS_PAYMENT_CRYPTOGRAPHY",
                "CreateTimestamp": "2024-03-07T06:41:46.648000-07:00",
                "UsageStartTimestamp": "2024-03-07T06:41:46.626000-07:00"
            }
        }
```

Take note of the `KeyArn` that represents the key, for example *arn:aws:payment-cryptography:us-east-2:111122223333:key/pw3s6nl62t5ushfk*. You need that in the next step.

Validate the ARQC

Example

In this example, we will validate an ARQC generated using Mastercard CVN12.

If AWS Payment Cryptography is able to validate the ARQC, an http/200 is returned. If the arqc is not validated, it will return a http/400 response.

```
$ aws payment-cryptography-data verify-auth-request-cryptogram --auth-request-cryptogram 31BE5D49F14A5F01 \
  --key-identifier arn:aws:payment-cryptography:us-east-2:111122223333:key/pw3s6nl62t5ushfk \
  --major-key-derivation-mode EMV_OPTION_A \
  --transaction-data
 0000000017000000000000000840008000800008401605170000000009380000 0B1F22010300000000000000000000
 \
```

```
  --session-key-derivation-attributes='{"MastercardSessionKey":
{"ApplicationTransactionCounter":"000B","PanSequenceNumber":"01","PrimaryAccountNumber":"541312
```

```
{
    "KeyArn": "arn:aws:payment-cryptography:us-east-2:111122223333:key/
pw3s6nl62t5ushfk",
    "KeyCheckValue": "08D7B4"
  }
```

3DS SPA2 AAV

SPA2(Secure Payment Application) AAV (Account Authentication Value) is used for Mastercard 3DS transactions (also known as Mastercard Idenity Check). It provides cryptographic authentication for e-commerce transactions using HMAC-based MAC generation. The AAV is generated using transaction-specific data and a shared secret key.

Create key

```
$ aws payment-cryptography create-key --exportable --key-attributes
 KeyAlgorithm=HMAC_SHA256,KeyUsage=TR31_M7_HMAC_KEY,KeyClass=SYMMETRIC_KEY,KeyModesOfUse='{Gene
 --tags='[{"Key":"KEY_PURPOSE","Value":"SPA2_AAV"},
{"Key":"CARD_BIN","Value":"12345678"}]'
```

The response echoes back the request parameters, including an ARN for subsequent calls as well as a Key Check Value (KCV).

```
{
            "Key": {
                "KeyArn": "arn:aws:payment-cryptography:us-west-2:111122223333:key/
q5vjtshsg67cz5gn",
                "KeyAttributes": {
                    "KeyUsage": "TR31_M7_HMAC_KEY",
                    "KeyClass": "SYMMETRIC_KEY",
                    "KeyAlgorithm": "HMAC_SHA256",
                    "KeyModesOfUse": {
                        "Encrypt": false,
                        "Decrypt": false,
                        "Wrap": false,
                        "Unwrap": false,
                        "Generate": true,
                        "Sign": false,
                        "Verify": true,
```

```
                        "DeriveKey": false,
                        "NoRestrictions": false
                    }
                },
                "KeyCheckValue": "C661F9",
                "KeyCheckValueAlgorithm": "HMAC",
                "Enabled": true,
                "Exportable": true,
                "KeyState": "CREATE_COMPLETE",
                "KeyOrigin": "AWS_PAYMENT_CRYPTOGRAPHY",
                "CreateTimestamp": "2024-03-07T06:41:46.648000-07:00",
                "UsageStartTimestamp": "2024-03-07T06:41:46.626000-07:00"
            }
        }
```

Take note of the `KeyArn` that represents the key, for example *arn:aws:payment-cryptography:us-west-2:111122223333:key/q5vjtshsg67cz5gn*. You need that in the next step.

Generate SPA2 AAV

Example

In this example, we will generate the Issuer Authentication Value (IAV) component of the SPA2 AAV using HMAC MAC generation. The message data contains the transaction-specific information that will be authenticated. The format of the message data should follow Mastercard's SPA2 specifications and is not covered in this example.

Note

Please review your Mastercard specifications for the formatting to insert the IAV into the AAV value.

```
$ aws payment-cryptography-data generate-mac --key-identifier arn:aws:payment-cryptography:us-west-2:111122223333:key/q5vjtshsg67cz5gn --message-data
 "2226400099919520FFFFd8b448be65694fe7b42f836bad396e9d" --generation-attributes
 Algorithm=HMAC --region us-west-2
```

```
{
    "KeyArn": "arn:aws:payment-cryptography:us-west-2:111122223333:key/
q5vjtshsg67cz5gn",
```

```
    "KeyCheckValue": "C661F9",
    "Mac": "6FB2405E9D8A4C1F7B173F73ADD1A6DC358531CAB0E9994FC5B62012ADDE91FC"
}
```

Verify SPA2 AAV

Example

In this example, we will verify an SPA2 AAV. The same message data and MAC value are provided for verification.

If AWS Payment Cryptography is able to validate the MAC, an http/200 is returned. If the MAC is not validated, it will return a http/400 response.

```
$ aws payment-cryptography-data verify-mac --key-identifier arn:aws:payment-cryptography:us-west-2:111122223333:key/q5vjtshsg67cz5gn --message-data "2226400099919520FFFFd8b448be65694fe7b42f836bad396e9d" --mac "6FB2405E9D8A4C1F7B173F73ADD1A6DC358531CAB0E9994FC5B62012ADDE91FC" --verification-attributes Algorithm=HMAC --region us-west-2
```

```
{
    "KeyArn": "arn:aws:payment-cryptography:us-west-2:111122223333:key/q5vjtshsg67cz5gn",
    "KeyCheckValue": "C661F9"
}
```

American Express specific functions

Topics

- CSC1
- CSC2
- iCSC
- 3DS AEVV

CSC1

CSC Version 1 is also known as the Classic CSC Algorithm. The service can provide it as a 3,4 or 5 digit number.

For all available parameters see AmexCardSecurityCodeVersion1 in the API reference guide.

Create key

```
$ aws payment-cryptography create-key --exportable --key-attributes
 KeyAlgorithm=TDES_2KEY,KeyUsage=TR31_C0_CARD_VERIFICATION_KEY,KeyClass=SYMMETRIC_KEY,KeyModesO
 --tags='[{"Key":"KEY_PURPOSE","Value":"CSC1"},{"Key":"CARD_BIN","Value":"12345678"}]'
```

The response echoes back the request parameters, including an ARN for subsequent calls as well as a Key Check Value (KCV).

```
{
                "Key": {
                    "KeyArn": "arn:aws:payment-cryptography:us-east-2:111122223333:key/
esh6hn7pxdtttzgq",
                    "KeyAttributes": {
                        "KeyUsage": "TR31_C0_CARD_VERIFICATION_KEY",
                        "KeyClass": "SYMMETRIC_KEY",
                        "KeyAlgorithm": "TDES_2KEY",
                        "KeyModesOfUse": {
                            "Encrypt": false,
                            "Decrypt": false,
                            "Wrap": false,
                            "Unwrap": false,
                            "Generate": true,
                            "Sign": false,
                            "Verify": true,
                            "DeriveKey": false,
                            "NoRestrictions": false
                        }
                    },
                    "KeyCheckValue": "8B5077",
                    "KeyCheckValueAlgorithm": "ANSI_X9_24",
                    "Enabled": true,
                    "Exportable": true,
                    "KeyState": "CREATE_COMPLETE",
                    "KeyOrigin": "AWS_PAYMENT_CRYPTOGRAPHY",
                    "CreateTimestamp": "2023-06-05T06:41:46.648000-07:00",
                    "UsageStartTimestamp": "2023-06-05T06:41:46.626000-07:00"
                }
            }
```

Take note of the `KeyArn` that represents the key, for example *arn:aws:payment-cryptography:us-east-2:111122223333:key/esh6hn7pxdtttzgq*. You need that in the next step.

Generate a CSC1

Example

```
$ aws payment-cryptography-data generate-card-validation-data --key-
identifier arn:aws:payment-cryptography:us-east-2:111122223333:key/
esh6hn7pxdtttzgq --primary-account-number=344131234567848 --generation-attributes
 AmexCardSecurityCodeVersion1='{CardExpiryDate=1224}' --validation-data-length 4
```

```
{
    "KeyArn": "arn:aws:payment-cryptography:us-east-2:111122223333:key/
esh6hn7pxdtttzgq",
    "KeyCheckValue": "8B5077",
    "ValidationData": "3938"
  }
```

Validate the CSC1

Example

In this example, we will validate a CSC1.

If AWS Payment Cryptography is able to validate, an http/200 is returned. If the value is not validated, it will return a http/400 response.

```
$ aws payment-cryptography-data verify-card-validation-data  --key-identifier
 arn:aws:payment-cryptography:us-east-2:111122223333:key/esh6hn7pxdtttzgq
 --primary-account-number=344131234567848 --verification-attributes
 AmexCardSecurityCodeVersion1='{CardExpiryDate=1224}''  --validation-data 3938
```

```
{
    "KeyArn": "arn:aws:payment-cryptography:us-east-2:111122223333:key/
esh6hn7pxdtttzgq",
    "KeyCheckValue": "8B5077"
  }
```

CSC2

CSC Version 2 is also known as the Enhanced CSC Algorithm. The service can provide it as a 3,4 or 5 digit number. The service code for CSC2 is typically 000.

For all available parameters see AmexCardSecurityCodeVersion2 in the API reference guide.

Create key

```
$ aws payment-cryptography create-key --exportable --key-attributes
 KeyAlgorithm=TDES_2KEY,KeyUsage=TR31_C0_CARD_VERIFICATION_KEY,KeyClass=SYMMETRIC_KEY,KeyModesO
 --tags='[{"Key":"KEY_PURPOSE","Value":"CSC2"},{"Key":"CARD_BIN","Value":"12345678"}]'
```

The response echoes back the request parameters, including an ARN for subsequent calls as well as a Key Check Value (KCV).

```
{
            "Key": {
                "KeyArn": "arn:aws:payment-cryptography:us-east-2:111122223333:key/
erlm445qvunmvoda",
                "KeyAttributes": {
                    "KeyUsage": "TR31_C0_CARD_VERIFICATION_KEY",
                    "KeyClass": "SYMMETRIC_KEY",
                    "KeyAlgorithm": "TDES_2KEY",
                    "KeyModesOfUse": {
                        "Encrypt": false,
                        "Decrypt": false,
                        "Wrap": false,
                        "Unwrap": false,
                        "Generate": true,
                        "Sign": false,
                        "Verify": true,
                        "DeriveKey": false,
                        "NoRestrictions": false
                    }
                },
                "KeyCheckValue": "BF1077",
                "KeyCheckValueAlgorithm": "ANSI_X9_24",
                "Enabled": true,
                "Exportable": true,
                "KeyState": "CREATE_COMPLETE",
                "KeyOrigin": "AWS_PAYMENT_CRYPTOGRAPHY",
                "CreateTimestamp": "2023-06-05T06:41:46.648000-07:00",
                "UsageStartTimestamp": "2023-06-05T06:41:46.626000-07:00"
            }
        }
```

Take note of the `KeyArn` that represents the key, for example *arn:aws:payment-cryptography:us-east-2:111122223333:key/erlm445qvunmvoda*. You need that in the next step.

Generate a CSC2

In this example, we will generate a CSC2 with a length of 4. CSC can be generated with a length of 3,4 or 5. For American Express, PANs should be 15 digits and start with 34 or 37. Expiration date is typically formatted as YYMM. Service code may vary - review your manual but typical values are 000, 201 or 702

Example

```
$ aws payment-cryptography-data generate-card-validation-data --key-
identifier arn:aws:payment-cryptography:us-east-2:111122223333:key/
erlm445qvunmvoda --primary-account-number=344131234567848 --generation-attributes
 AmexCardSecurityCodeVersion2='{CardExpiryDate=2412,ServiceCode=000}' --validation-
data-length 4
```

```
{
"KeyArn": "arn:aws:payment-cryptography:us-east-2:111122223333:key/erlm445qvunmvoda",
"KeyCheckValue": "BF1077",
"ValidationData": "3982"
}
```

Validate the CSC2

Example

In this example, we will validate a CSC2.

If AWS Payment Cryptography is able to validate, an http/200 is returned. If the value is not validated, it will return a http/400 response.

```
$ aws payment-cryptography-data verify-card-validation-data  --key-identifier
 arn:aws:payment-cryptography:us-east-2:111122223333:key/erlm445qvunmvoda
 --primary-account-number=344131234567848 --verification-attributes
 AmexCardSecurityCodeVersion2='{CardExpiryDate=2412,ServiceCode=000}' --validation-data
 3982
```

```
{
"KeyArn": "arn:aws:payment-cryptography:us-east-2:111122223333:key/erlm445qvunmvoda",
"KeyCheckValue": "BF1077"
}
```

iCSC

iCSC is also known as a static CSC Algorithm and is calculated using CSC Version 2. The service can provide it as a 3,4 or 5 digit number.

Use service code 999 to calculate iCSC for a contact card. Use service code 702 to calculate iCSC for a contactless card.

For all available parameters see AmexCardSecurityCodeVersion2 in the API reference guide.

Create key

```
$ aws payment-cryptography create-key --exportable --key-attributes
 KeyAlgorithm=TDES_2KEY,KeyUsage=TR31_C0_CARD_VERIFICATION_KEY,KeyClass=SYMMETRIC_KEY,KeyModesO
 --tags='[{"Key":"KEY_PURPOSE","Value":"CSC1"},{"Key":"CARD_BIN","Value":"12345678"}]'
```

The response echoes back the request parameters, including an ARN for subsequent calls as well as a Key Check Value (KCV).

```
{
        "Key": {
        "KeyArn": "arn:aws:payment-cryptography:us-
east-1:111122223333:key/7vrybrbvjcvwtunv",
        "KeyAttributes": {
            "KeyUsage": "TR31_C0_CARD_VERIFICATION_KEY"
            "KeyAlgorithm": "TDES_2KEY",
            "KeyClass": "SYMMETRIC_KEY",
            "KeyModesOfUse": {
                "Decrypt": false,
                "DeriveKey": false,
                "Encrypt": false,
                "Generate": true,
                "NoRestrictions": false,
                "Sign": false,
                "Unwrap": false,
                "Verify": true,
                "Wrap": false
            },
        },
        "KeyCheckValue": "7121C7",
        "KeyCheckValueAlgorithm": "ANSI_X9_24",
        "Enabled": true,
        "Exportable": true,
```

```
        "KeyOrigin": "AWS_PAYMENT_CRYPTOGRAPHY",
        "KeyState": "CREATE_COMPLETE",
        "CreateTimestamp": "2025-01-29T09:19:21.209000-05:00",
        "UsageStartTimestamp": "2025-01-29T09:19:21.192000-05:00"
        }
    }
```

Take note of the `KeyArn` that represents the key, for example *arn:aws:payment-cryptography:us-east-1:111122223333:key/7vrybrbvjcvwtunv*. You need that in the next step.

Generate a iCSC

In this example, we will generate a iCSC with a length of 4, for a contactless card using service code 702. CSC can be generated with a length of 3,4 or 5. For American Express, PANs should be 15 digits and start with 34 or 37.

Example

```
$ aws payment-cryptography-data generate-card-validation-data --key-identifier
 arn:aws:payment-cryptography:us-east-1:111122223333:key/7vrybrbvjcvwtunv
 --primary-account-number=344131234567848 --generation-attributes
 AmexCardSecurityCodeVersion2='{CardExpiryDate=1224,ServiceCode=702}' --validation-
data-length 4
```

```
{
    "KeyArn": arn:aws:payment-cryptography:us-east-1:111122223333:key/7vrybrbvjcvwtunv,
    "KeyCheckValue": 7121C7,
    "ValidationData": "2365"
}
```

Validate the iCSC

Example

In this example, we will validate a iCSC.

If AWS Payment Cryptography is able to validate, an http/200 is returned. If the value is not validated, it will return a http/400 response.

```
$ aws payment-cryptography-data verify-card-validation-data  --key-identifier
 arn:aws:payment-cryptography:us-east-1:111122223333:key/7vrybrbvjcvwtunv
```

```
--primary-account-number=344131234567848 --verification-attributes
AmexCardSecurityCodeVersion2='{CardExpiryDate=1224,ServiceCode=702}' --validation-data
2365
```

```
{
    "KeyArn": arn:aws:payment-cryptography:us-east-1:111122223333:key/7vrybrbvjcvwtunv,
    "KeyCheckValue": 7121C7
}
```

3DS AEVV

3DS AEVV (3-D Secure Account Verification Value) is used for American Express 3-D Secure authentication. It uses the same algorithm as CSC2 but with different input parameters. The expiration date field should be populated with an unpredictable (random) number, and the service code consists of the AEVV Authentication Results Code (1 digit) plus the Second Factor Authentication Code (2 digits). The output length should be 3-digits.

For all available parameters see AmexCardSecurityCodeVersion2 in the API reference guide.

Create key

```
$ aws payment-cryptography create-key --exportable --key-attributes
 KeyAlgorithm=TDES_2KEY,KeyUsage=TR31_C0_CARD_VERIFICATION_KEY,KeyClass=SYMMETRIC_KEY,KeyModesO
 --tags='[{"Key":"KEY_PURPOSE","Value":"3DS_AEVV"},
{"Key":"CARD_BIN","Value":"12345678"}]'
```

The response echoes back the request parameters, including an ARN for subsequent calls as well as a Key Check Value (KCV).

```
{
        "Key": {
        "KeyArn": "arn:aws:payment-cryptography:us-east-2:111122223333:key/
kw8djn5qxvfh3ztm",
        "KeyAttributes": {
            "KeyUsage": "TR31_C0_CARD_VERIFICATION_KEY"
            "KeyAlgorithm": "TDES_2KEY",
            "KeyClass": "SYMMETRIC_KEY",
            "KeyModesOfUse": {
                "Decrypt": false,
                "DeriveKey": false,
```

```
                "Encrypt": false,
                "Generate": true,
                "NoRestrictions": false,
                "Sign": false,
                "Unwrap": false,
                "Verify": true,
                "Wrap": false
            },
        },
        "KeyCheckValue": "8F3A21",
        "KeyCheckValueAlgorithm": "ANSI_X9_24",
        "Enabled": true,
        "Exportable": true,
        "KeyOrigin": "AWS_PAYMENT_CRYPTOGRAPHY",
        "KeyState": "CREATE_COMPLETE",
        "CreateTimestamp": "2025-02-02T10:30:15.209000-05:00",
        "UsageStartTimestamp": "2025-02-02T10:30:15.192000-05:00"
        }
    }
```


Take note of the `KeyArn` that represents the key, for example *arn:aws:payment-cryptography:us-east-2:111122223333:key/kw8djn5qxvfh3ztm*. You need that in the next step.

Generate a 3DS AEVV

In this example, we will generate a 3DS AEVV with a length of 3. The expiration date field contains an unpredictable (random) number (e.g., 1234), and the service code consists of the AEVV Authentication Results Code (1 digit) plus the Second Factor Authentication Code (2 digits), for example 543 where 5 is the Authentication Results Code and 43 is the Second Factor Authentication Code. For American Express, PANs should be 15 digits and start with 34 or 37.

Example

```
$ aws payment-cryptography-data generate-card-validation-data --key-identifier arn:aws:payment-cryptography:us-east-2:111122223333:key/kw8djn5qxvfh3ztm --primary-account-number=344131234567848 --generation-attributes AmexCardSecurityCodeVersion2='{CardExpiryDate=1234,ServiceCode=543}' --validation-data-length 3
```


```
{
    "KeyArn": arn:aws:payment-cryptography:us-east-2:111122223333:key/kw8djn5qxvfh3ztm,
```

```
    "KeyCheckValue": 8F3A21,
    "ValidationData": "921"
}
```

Validate the 3DS AEVV

Example

In this example, we will validate a 3DS AEVV.

If AWS Payment Cryptography is able to validate, an http/200 is returned. If the value is not validated, it will return a http/400 response.

```
$ aws payment-cryptography-data verify-card-validation-data  --key-identifier
 arn:aws:payment-cryptography:us-east-2:111122223333:key/kw8djn5qxvfh3ztm
 --primary-account-number=344131234567848 --verification-attributes
 AmexCardSecurityCodeVersion2='{CardExpiryDate=1234,ServiceCode=543}' --validation-data
 921
```

```
{
    "KeyArn": arn:aws:payment-cryptography:us-east-2:111122223333:key/kw8djn5qxvfh3ztm,
    "KeyCheckValue": 8F3A21
}
```

JCB specific functions

Topics

- ARQC - CVN04
- ARQC - CVN01

ARQC - CVN04

JCB CVN04 utilizes the CSK method of key derivation. Please see the scheme documentation for details on constructing the transaction data field.

ARQC - CVN01

CVN01 is an older JCB method for EMV transactions that uses per card key derivation rather than session (per transaction) derivation and also uses a different payload. This message is also used by

Visa hence the element name has that name even though it's also used for JCB. For information on the payload contents, please contact the scheme documentation.

Create key

```
$ aws payment-cryptography create-key --exportable --key-attributes
 KeyAlgorithm=TDES_2KEY,KeyUsage=TR31_E0_EMV_MKEY_APP_CRYPTOGRAMS,KeyClass=SYMMETRIC_KEY,KeyMod
 --tags='[{"Key":"KEY_PURPOSE","Value":"CVN10"},{"Key":"CARD_BIN","Value":"12345678"}]'
```

The response echoes back the request parameters, including an ARN for subsequent calls as well as a Key Check Value (KCV).


```
{
                        "Key": {
                            "KeyArn": "arn:aws:payment-cryptography:us-
east-2:111122223333:key/pw3s6nl62t5ushfk",
                            "KeyAttributes": {
                                "KeyUsage": "TR31_E0_EMV_MKEY_APP_CRYPTOGRAMS",
                                "KeyClass": "SYMMETRIC_KEY",
                                "KeyAlgorithm": "TDES_2KEY",
                                "KeyModesOfUse": {
                                    "Encrypt": false,
                                    "Decrypt": false,
                                    "Wrap": false,
                                    "Unwrap": false,
                                    "Generate": false,
                                    "Sign": false,
                                    "Verify": false,
                                    "DeriveKey": true,
                                    "NoRestrictions": false
                                }
                            },
                            "KeyCheckValue": "08D7B4",
                            "KeyCheckValueAlgorithm": "ANSI_X9_24",
                            "Enabled": true,
                            "Exportable": true,
                            "KeyState": "CREATE_COMPLETE",
                            "KeyOrigin": "AWS_PAYMENT_CRYPTOGRAPHY",
                            "CreateTimestamp": "2024-03-07T06:41:46.648000-07:00",
                            "UsageStartTimestamp": "2024-03-07T06:41:46.626000-07:00"
                        }
                    }
```

Take note of the `KeyArn` that represents the key, for example *arn:aws:payment-cryptography:us-east-2:111122223333:key/pw3s6nl62t5ushfk*. You need that in the next step.

Validate the ARQC

Example

In this example, we will validate an ARQC generated using JCB CVN01. This uses the same options as the Visa method, hence the name of the parameter.

If AWS Payment Cryptography is able to validate the ARQC, an http/200 is returned. If the arqc is not validated, it will return a http/400 response.

```
$ aws payment-cryptography-data verify-auth-request-cryptogram --auth-request-cryptogram D791093C8A921769 \
              --key-identifier arn:aws:payment-cryptography:us-east-2:111122223333:key/pw3s6nl62t5ushfk \
              --major-key-derivation-mode EMV_OPTION_A \
              --transaction-data 00000000170000000000000008400080008000084016051700000000093800000B03011203000000 \
              --session-key-derivation-attributes='{"Visa":{"PanSequenceNumber":"01" \
              ,"PrimaryAccountNumber":"9137631040001422"}}'
```

```
{
                "KeyArn": "arn:aws:payment-cryptography:us-east-2:111122223333:key/pw3s6nl62t5ushfk",
                "KeyCheckValue": "08D7B4"
              }
```

Acquiring and payment facilitators

Acquirers, PSPs and Payment Facilators typically have a different set of cryptographic requirements than issuers. Common use cases include:

Data Decryption

Data (especially pan data) may be encrypted by a payment terminal and need to be decrypted by the backend. Decrypt Data and Encrypt Data support a variety of methods including TDES, AES and DUKPT derivation techniques. The AWS Payment Cryptography service itself is also PCI P2PE compliant and is registered as a PCI P2PE decryption component.

TranslatePin

To maintain PCI PIN compliance, acquiring systems shall not have cardholder pins in the clear after they have been entered on a secure device. Therefore, to pass the pin onward from terminal to a downstream system (such as a payment network or issuer), there is a need to re-encrypt it using a different key than the one that the payment terminal used. Translate Pin accomplishes that by converting an encrypted pin from one key to another securely with the servicebbb. Using this command, pins can be converted between various schemes such as TDES, AES and DUKPT derivation and pin block formats such as ISO-0, ISO-3 and ISO-4.

VerifyMac

Data from a payment terminal may be MAC'd to ensure that the data hasn't been modified in transit. Verify Mac and GenerateMac supports a variety of techniques using symmetric keys including TDES, AES and DUKPT derivation techniques for use with ISO-9797-1 algorithm 1, ISO-9797-1 algorithm 3 (Retail MAC) and CMAC techniques.

Additional Topics

- Using Dynamic Keys

Using Dynamic Keys

Dynamic Keys allows one-time or limited use keys to be used for cryptographic operations like `EncryptData`. This flow can be utilized when the key material frequently rotates (such as on every card transaction) and there is a desire to avoid importing the key material into the service. Short-lived keys may be utilized as part of softPOS/Mpoc or other solutions.

Note

This can be used in lieu of the typical flow using AWS Payment Cryptography, where cryptographic keys are either created or imported into the service and keys are specified using a key alias or key arn.

The following operations support Dynamic Keys:

- EncryptData
- DecryptData

- ReEncryptData
- TranslatePin

Decrypting Data

The following example shows using Dynamic Keys along with the decrypt command. The key identifier in this case is the wrapping key (KEK) that secures the decryption key (that is provided in the wrapped-key parameter in TR-31 format). The wrapped key shall be key purpose of D0 to be used with decrypt command along with a mode of use of B or D.

Example

```
$ aws payment-cryptography-data decrypt-data --key-identifier
 arn:aws:payment-cryptography:us-east-2:111122223333:key/ov6icy4ryas4zcza
 --cipher-text 1234123412341234123412341234123A --decryption-attributes
 'Symmetric={Mode=CBC,InitializationVector=1234123412341234}'   --wrapped-key
 WrappedKeyMaterial={"Tr31KeyBlock"="D0112D0TN00E0000B05A6E82D7FC68B95C84306634B0000DA4701BE9BC
```

```
{
   "KeyArn": "arn:aws:payment-cryptography:us-east-2:111122223333:key/
ov6icy4ryas4zcza",
   "KeyCheckValue": "0A3674",
   "PlainText": "2E138A746A0032023BEF5B85BA5060BA"
}
```

Translating a pin

The following example shows using Dynamic Keys along with the translate pin command to translate from a dynamic key to a semi-static acquirer working key (AWK). The incoming key identifier in this case is the wrapping key (KEK) that is protecting the dynamic pin encryption key (PEK) that is provided in the TR-31 format. The wrapped key shall be key purpose of `P0` along with a mode of use of B or D. The outgoing key identifier is a key of type `TR31_P0_PIN_ENCRYPTION_KEY` and a mode of use of Encrypt=true,Wrap=true

Example

```
$ aws payment-cryptography-data translate-pin-data --encrypted-pin-block
 "C7005A4C0FA23E02" --incoming-translation-
```

```
attributes=IsoFormat0='{PrimaryAccountNumber=171234567890123}'
 --incoming-key-identifier alias/PARTNER1_KEK  --outgoing-key-
identifier alias/ACQUIRER_AWK_PEK --outgoing-translation-attributes
 IsoFormat0="{PrimaryAccountNumber=171234567890123}"  --incoming-wrapped-key
 WrappedKeyMaterial={"Tr31KeyBlock"="D0112P0TB00S0000EB5D8E63076313162B04245C8CE351C956EA4A16CC
```

```
{
   "PinBlock": "2E66192BDA390C6F",
   "KeyArn": "arn:aws:payment-cryptography:us-east-2:111122223333:key/
ov6icy4ryas4zcza",
   "KeyCheckValue": "0A3674"
}
```

Region specific features for AWS Payment Cryptography

Certain features may be region specific and not otherwise used. Those features are described in more detail in this section.

AS2805

Australia Standard 2805 (AS2805) is a standard for electronic funds transfers used primarily for card-based payment transactions. It is maintained by Standards Australia. The standard consists of 6 books that covers numerous topics from message format to encryption standards.

Part 6 provides guidance on key management including host-to-host (node-to-node) communication and relevant cryptographic requirements while other aspects are covered in other parts. All cryptography in this standard is currently based on TDES.

Note

AS2805 is currently available in the ap-southeast-2 Region. It will be rolled out to additional Regions in the near future.

AS2805 has a number of differences compared to other implementations, which are summarized below.

Key Protection

Relies on key variants instead of keyblocks such as in TR-31/X9.143. AWS Payment Cryptography stores all keys as key blocks internally but permits import, export and calculation using AS2805 defined variants.

Unidirectional Keys

AS2805 mandates the use of unidirectional keys. If both nodes need to generate message authentication codes (MAC), they use two keys.

Pin Blocks

AS2805 defines a key derivation technique for unique pin encryption keys per transaction. This can be used in lieu of DUKPT. The AS2805 scheme relies on transaction data (trace number and transaction amount) as compared to DUKPT's use of transaction counter.

Key Exchange Validation

Defines a process to validate KEK before beginning to exchange working keys such as pin keys. In other schemes, KEK are exchanged infrequently and are validated using KCV.

AS2805 uses the concept of key variants rather than key blocks to ensure keys are only used for the intended (and sole) purpose. The following is how AWS Payment Cryptography maps between variants and keyblocks when importing, exporting or performing other cryptographic functions with keys.

AS2805 Key Type	AWS Payment Cryptography Key Type
TERMINAL_MAJOR_KEY_VARIANT_00	TR31_K0_KEY_ENCRYPTION_KEY
PIN_ENCRYPTION_KEY_VARIANT_28	TR31_P0_PIN_ENCRYPTION_KEY
MESSAGE_AUTHENTICATION_KEY_VARIANT_24	TR31_M0_ISO_16609_MAC_KEY
DATA_ENCRYPTION_KEY_VARIANT_22	TR31_D0_SYMMETRIC_DATA_ENCRYPTION_KEY
VARIANT_MASK_82,VARIANT_MASK_82C0	Options available as part of KEK validation process. These key types are ephemeral and are not stored by the service.

Given two nodes, node1 and node2, the following examples are from the perspective of node1. AWS Payment Cryptography supports APIs from both sides of the process.

Topics

- Initial Key(KEK) exchange
- Validation of KEK
- Creation and transmission of working keys
- Exporting working keys
- Pin Translation
- Mac Generation and Validation

Initial Key(KEK) exchange

In AS2805, each side has their own KEK. KEK(s) refers to the sending side key that will be used whenever the sending side needs to protect/wrap keys and send them to node2. KEK(r) is the key created by the opposite(node2) side.

Note

These terms are relative - one side creates a key (sending side) and the other side receives it. So given KEY1, it is referred to on node1 as KEK(s) and on node2 as KEK(r).

KEK for AS2805 are always key type = TR31_K0_KEY_ENCRYPTION_KEY as they are used to protect cryptograms and not key blocks. This maps to TERMINAL_MAJOR_KEY_VARIANT_00 as defined in AS2805 6.1

Steps:

1.Create a key

Create a key using the CreateKey api. You will create a key of type TR31_K0_KEY_ENCRYPTION_KEY

2.Determine method to exchange keys with node2

Determine how to exchange KEK with the counter party. For AS2805, the most common and interoperable method is RSA Wrap.

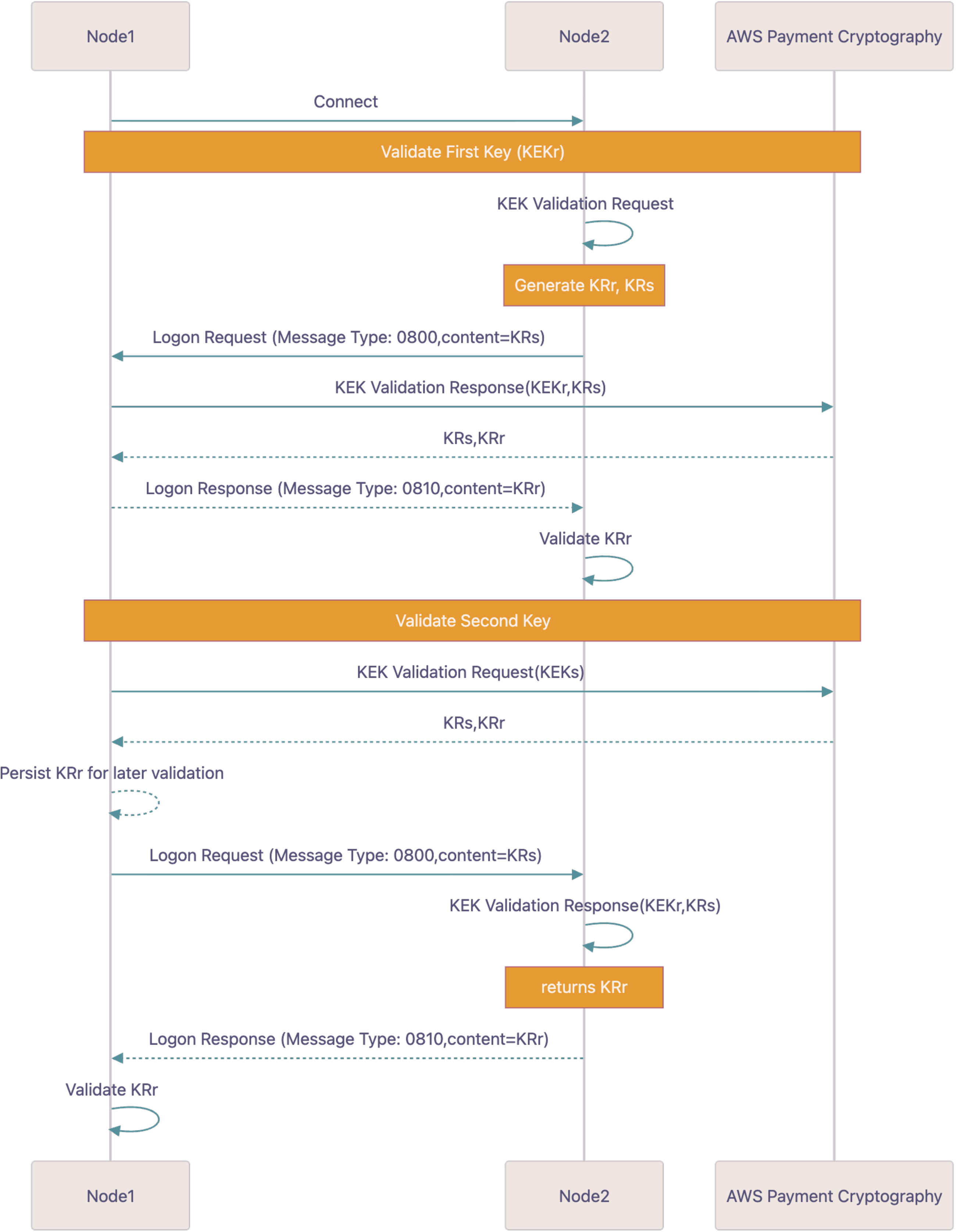
3.Export KEKs

Based on your selection above, you will receive public key certificate from node2. You will run export using that certificate to protect the key (or derive a key if using ECDH).

4. Import KEKr

Based on your selection above, you will send a public key certificate to node2. You will run import using that certificate to to load node 2's KEKr into the service.

Validation of KEK



When your service (node1) connects to node2, each side will ensure that they are using the same KEK for subsequent operations using a process called KEK Validation.

1. Steps to validate the first key

1.1 Receive KRs

Node2 will generate a KRs and send it to you as part of the logon process. They may use AWS Payment Cryptography to generate this value or another solution.

1.2 Generate KEK Validation Response

Your node will generate a KEK Validation response with inputs as the KEK(r) and the KRs provided in step 1.

Example

```
cat >> generate-kek-validation-response.json
{
    "KekValidationType": {
        "KekValidationResponse": {
            "RandomKeySend": "9217DC67B8763BABCFDF3DADFCD0F84A"
        }
    },
    "RandomKeySendVariantMask": "VARIANT_MASK_82",
    "KeyIdentifier": "arn:aws:payment-cryptography:us-east-2:111122223333:key/
ov6icy4ryas4zcza"
}
```

```
$ aws payment-cryptography-data generate-as2805-kek-validation --cli-input-json
 file://generate-kek-validation-response.json
```

```
{
 "KeyArn": "arn:aws:payment-cryptography:us-east-2:111122223333:key/
ov6icy4ryas4zcza",
 "KeyCheckValue": "0A3674",
 "RandomKeyReceive": "A4B7E249C40C98178C1B856DB7FB76EB",
 "RandomKeySend": "9217DC67B8763BABCFDF3DADFCD0F84A"
}
```

1.3 Return calculated KRr

Return the calculated KRr to node2. That node will compare it against the calculated value from step 1.

2.Steps to validate the second key

2.1 Generate KRr and KRs

Your node will generate a random value and an inverted(reversed) copy of this value using AWS Payment Cryptography. The service will output both of these values wrapped by the KEK(s). These are known as KR(s) and KR(r).

Example

```
cat >> generate-kek-validation-request.json
{
    "KekValidationType": {
        "KekValidationRequest": {
            "DeriveKeyAlgorithm": "TDES_2KEY"
        }
    },
"RandomKeySendVariantMask": "VARIANT_MASK_82",
    "KeyIdentifier": "arn:aws:payment-cryptography:us-east-2:111122223333:key/
rhfm6tenpxapkmrv"
}
```

```
$ aws payment-cryptography-data generate-as2805-kek-validation --cli-input-json
 file://generate-kek-validation-request.json
```

```
{
 "KeyArn": "arn:aws:payment-cryptography:us-east-2:111122223333:key/
rhfm6tenpxapkmrv",
 "KeyCheckValue": "DC1081",
 "RandomKeyReceive": "A4B7E249C40C98178C1B856DB7FB76EB",
 "RandomKeySend": "9217DC67B8763BABCFDF3DADFCD0F84A"
}
```

2.2 Send KRs to node2

Send the KRs to node2. Keep the KRr for later validation.

2.3 Node2 generates KEK validation response

Node2 uses the KEKr and KRs, generates the KRr and sends it back to your service.

2.4 Validate response

Compare KRr from step 1 and the value returned from step 3. If they match, proceed.

Creation and transmission of working keys

Typical working keys used in AS2805 includes two sets of keys:

Keys between nodes such as: zone pin key(ZPK), zone encryption key (ZEK) and zone authentication key(ZAK).

Keys between terminals and nodes such as: terminal main key(TMK) and terminal pin key(TPK) if not using DUKPT.

Note

We recommend minimizing key per terminal keys and leveraging techniques such as TR-34 and DUKPT whenever possible that use smaller numbers of keys.

Example

In this example, we've used optional tags to track the purpose and use of this key. Tags are not used as part of the cryptographic function of the system but can be used for categorization, financial tracking and can be used to apply IAM policies.

```
cat >> create-zone-pin-key.json
{
    "KeyAttributes": {
        "KeyUsage": "TR31_P0_PIN_ENCRYPTION_KEY",
        "KeyClass": "SYMMETRIC_KEY",
        "KeyAlgorithm": "TDES_2KEY",
        "KeyModesOfUse": {
            "Encrypt": true,
            "Decrypt": true,
            "Wrap": true,
            "Unwrap": true,
            "Generate": false,
```

```
            "Sign": false,
            "Verify": false,
            "DeriveKey": false,
            "NoRestrictions": false
        }
    },
    "KeyCheckValueAlgorithm": "ANSI_X9_24",
    "Exportable": true,
    "Enabled": true,
    "Tags": [
        {
            "Key": "AS2805_KEYTYPE",
            "Value": "ZONE_PIN_KEY_VARIANT28"
        }
    ]
}
```

```
$ aws payment-cryptography-data create-key --cli-input-json file://create-zone-pin-key.json --region ap-southeast-2
```

```
{
 "Key": {
 "KeyArn": "arn:aws:payment-cryptography:us-east-2:111122223333:key/alsuwfxug3pgy6xh",
 "KeyAttributes": {
 "KeyUsage": "TR31_P0_PIN_ENCRYPTION_KEY",
 "KeyClass": "SYMMETRIC_KEY",
 "KeyAlgorithm": "TDES_2KEY",
 "KeyModesOfUse": {
 "Encrypt": true,
 "Decrypt": true,
 "Wrap": true,
 "Unwrap": true,
 "Generate": false,
 "Sign": false,
 "Verify": false,
 "DeriveKey": false,
 "NoRestrictions": false
 }
 },
 "KeyCheckValue": "9A325B",
 "KeyCheckValueAlgorithm": "ANSI_X9_24",
 "Enabled": true,
 "Exportable": true,
```

```
 "KeyState": "CREATE_COMPLETE",
 "KeyOrigin": "AWS_PAYMENT_CRYPTOGRAPHY",
 "CreateTimestamp": "2025-12-17T09:05:27.586000-08:00",
 "UsageStartTimestamp": "2025-12-17T09:05:27.570000-08:00"
 }
}
```

Exporting working keys

To maintain compatibility with other parties, AWS Payment Cryptography supports AS2805 symmetric key wrapping techniques which use key variants instead of keyblocks like TR-31. If multiple keys are shared between parties, each should be exported individually. If data is sent bi-directionally, there may be two keys between parties of the same type such as ZAK(s) and ZAK(r) that are used by each side to generate message authentication codes.

The additional parameters to import and export in these formats are specified on the commands.

```
cat >> export-zone-pin-key.json
{
    "ExportKeyIdentifier": "arn:aws:payment-cryptography:us-east-2:111122223333:key/
alsuwfxug3pgy6xh",
    "KeyMaterial": {
        "As2805KeyCryptogram": {
            "WrappingKeyIdentifier": "arn:aws:payment-cryptography:us-
east-2:111122223333:key/rhfm6tenpxapkmrv",
            "As2805KeyVariant: "PIN_ENCRYPTION_KEY_VARIANT_28"
        }
    }
}
```

```
$ aws payment-cryptography-data export-key --cli-input-json file://export-zone-pin-
key.json --region ap-southeast-2
```

```
{
    "WrappedKey": {
        "KeyCheckValue": "DC1081",
        "KeyCheckValueAlgorithm": "ANSI_X9_24",
        "KeyMaterial": "HDC10AEF038E695DDD72AF08DC1BB422D",
        "WrappedKeyMaterialFormat": "KEY_CRYPTOGRAM",
        "WrappingKeyArn": "arn:aws:payment-cryptography:us-east-2:111122223333:key/
rhfm6tenpxapkmrv"
```

```
    }
}
```

Pin Translation

AS2805 describes a session specific key derivation mode in section 6.4. It serves a similiar purpose as DUKPT and either algorithm can be used as DUKPT is covered in section 6.7. In this scheme, a session pin key (known as a KPE) is derived from the Terminal Pin Key using SystemTraceAuditNumber(STAN) and TransactionAmount as the derivation data.

Translate pin is a common function that can translate to/from a variety of formats. In this example, we translate a pin from a KPE to a pin encryption key (PEK) such as when sending a pin to a payment network.

```
cat >> translate-pin-as2805.json
{
    "EncryptedPinBlock": "B3B34B43BAB5F81A",
    "IncomingKeyIdentifier": "arn:aws:payment-cryptography:us-east-2:111122223333:key/
ivi5ksfsuplneuyt",
    "IncomingTranslationAttributes": {
        "IsoFormat0": {
            "PrimaryAccountNumber": "9999179999900013"
        }
    },
      "IncomingAs2805Attributes": {
        "SystemTraceAuditNumber": "000348",
        "TransactionAmount": "000000000328"
    },
    "OutgoingKeyIdentifier": "",
    "OutgoingTranslationAttributes": {
        "IsoFormat0": {
            "PrimaryAccountNumber": "9999179999900013"
        }
    }
}
```

```
$ aws payment-cryptography-data translate-pin-data --cli-input-json file://translate-
pin-as2805.json  --region ap-southeast-2
```

```
{
    "WrappedKey": {
```

```
        "KeyCheckValue": "DC1081",
        "KeyCheckValueAlgorithm": "ANSI_X9_24",
        "KeyMaterial": "HDC10AEF038E695DDD72AF08DC1BB422D",
        "WrappedKeyMaterialFormat": "KEY_CRYPTOGRAM",
        "WrappingKeyArn": "arn:aws:payment-cryptography:us-east-2:111122223333:key/
rhfm6tenpxapkmrv"
    }
}
```

Mac Generation and Validation

The generate and verify MAC commands support a variety of MACs including HMAC, CMAC, EMV MAC, etc. For AS2805, there is an additional variation defined in AS2805.4.1. Typically in AS2805, incoming messages are verified using this MAC and outgoing messages include a MAC as well.

```
cat verify-mac.json
{
    "KeyIdentifier": "arn:aws:payment-cryptography:us-east-2:111122223333:key/
qnobl5lghrzunce6",
    "Mac": "86304058",
    "MessageData": "73D8BA54D3852951DAEA41",
    "VerificationAttributes": {
        "Algorithm": "AS2805_4_1"
    }
}
```

```
$ aws payment-cryptography-data verify-mac --cli-input-json file://verify-mac.json --
region ap-southeast-2
```

```
{
    "KeyIdentifier": "arn:aws:payment-cryptography:us-east-2:111122223333:key/
qnobl5lghrzunce6",
    "KeyCheckValue": "2976E7"
}
```

Security in AWS Payment Cryptography

Cloud security at AWS is the highest priority. As an AWS customer, you benefit from a data center and network architecture that is built to meet the requirements of the most security-sensitive organizations.

Security is a shared responsibility between AWS and you. The shared responsibility model describes this as security of the cloud and security in the cloud:

- **Security of the cloud**—AWS is responsible for protecting the infrastructure that runs AWS services in the AWS Cloud. AWS also provides you with services that you can use securely. Third-party auditors regularly test and verify the effectiveness of our security as part of the AWS Compliance Programs. To learn about the compliance programs that apply to AWS Payment Cryptography, see AWS Services in Scope by Compliance Program.
- **Security in the cloud**—Your responsibility is determined by the AWS service that you use. You are also responsible for other factors including the sensitivity of your data, your company's requirements, and applicable laws and regulations.

This topic helps you understand how to apply the shared responsibility model when using AWS Payment Cryptography. It shows you how to configure AWS Payment Cryptography to meet your security and compliance objectives. You also learn how to use other AWS services that help you to monitor and secure your AWS Payment Cryptography resources.

Topics

- Data protection in AWS Payment Cryptography
- Resilience in AWS Payment Cryptography
- Infrastructure security in AWS Payment Cryptography
- Connecting to AWS Payment Cryptography through a VPC endpoint
- Using hybrid post-quantum TLS
- Security best practices for AWS Payment Cryptography

Data protection in AWS Payment Cryptography

The AWS shared responsibility model applies to data protection in AWS Payment Cryptography. As described in this model, AWS is responsible for protecting the global infrastructure that runs all

of the AWS Cloud. You are responsible for maintaining control over your content that is hosted on this infrastructure. You are also responsible for the security configuration and management tasks for the AWS services that you use. For more information about data privacy, see the Data Privacy FAQ. For information about data protection in Europe, see the AWS Shared Responsibility Model and GDPR blog post on the *AWS Security Blog*.

For data protection purposes, we recommend that you protect AWS account credentials and set up individual users with AWS IAM Identity Center or AWS Identity and Access Management (IAM). That way, each user is given only the permissions necessary to fulfill their job duties. We also recommend that you secure your data in the following ways:

- Use multi-factor authentication (MFA) with each account.
- Use SSL/TLS to communicate with AWS resources. We require TLS 1.2 and recommend TLS 1.3.
- Set up API and user activity logging with AWS CloudTrail. For information about using CloudTrail trails to capture AWS activities, see Working with CloudTrail trails in the *AWS CloudTrail User Guide*.
- Use AWS encryption solutions, along with all default security controls within AWS services.
- Use advanced managed security services such as Amazon Macie, which assists in discovering and securing sensitive data that is stored in Amazon S3.
- If you require FIPS 140-3 validated cryptographic modules when accessing AWS through a command line interface or an API, use a FIPS endpoint. For more information about the available FIPS endpoints, see Federal Information Processing Standard (FIPS) 140-3.

We strongly recommend that you never put confidential or sensitive information, such as your customers' email addresses, into tags or free-form text fields such as a **Name** field. This includes when you work with AWS Payment Cryptography or other AWS services using the console, API, AWS CLI, or AWS SDKs. Any data that you enter into tags or free-form text fields used for names may be used for billing or diagnostic logs. If you provide a URL to an external server, we strongly recommend that you do not include credentials information in the URL to validate your request to that server.

AWS Payment Cryptography stores and protects your payment encryption keys to make them highly available while providing you with strong and flexible access control.

Topics

- Protecting key material

- Data encryption
- Encryption at rest
- Encryption in transit
- Internetwork traffic privacy

Protecting key material

By default, AWS Payment Cryptography protects the cryptographic key material for payment keys managed by the service. In addition, AWS Payment Cryptography offers options for importing key material that is created outside of the service. For technical details about payment keys and key material, see AWS Payment Cryptography Cryptographic Details.

Data encryption

The data in AWS Payment Cryptography consists of AWS Payment Cryptography keys, the encryption key material they represent, and their usage attributes. Key material exists in plaintext only within AWS Payment Cryptography hardware security modules (HSMs) and only when in use. Otherwise, the key material and attributes are encrypted and stored in durable persistent storage.

The key material that AWS Payment Cryptography generates or loads for payment keys never leaves the boundary of AWS Payment Cryptography HSMs unencrypted. It can be exported encrypted by AWS Payment Cryptography API operations.

Encryption at rest

AWS Payment Cryptography generates key material for payment keys in PCI PTS HSM-listed HSMs. When not in use, key material is encrypted by an HSM key and written to durable, persistent storage. The key material for Payment Cryptography keys and the encryption keys that protect the key material never leave the HSMs in plaintext form.

Encryption and management of key material for Payment Cryptography keys is handled entirely by the service.

For more details, see AWS Key Management Service Cryptographic Details.

Encryption in transit

Key material that AWS Payment Cryptography generates or loads for payment keys is never exported or transmitted in AWS Payment Cryptography API operations in cleartext. AWS Payment Cryptography uses key identifiers to represent the keys in API operations.

However, some API operations export keys encrypted by a previously shared or asymmetric key exchange key. Also, customers can use API operations to import encrypted key material for payment keys.

All AWS Payment Cryptography API calls must be signed and be transmitted using Transport Layer Security (TLS). AWS Payment Cryptography requires TLS versions and cipher suites defined by PCI as "strong cryptography". All service endpoints support TLS 1.2—1.3 and hybrid post-quantum TLS.

For more details, see AWS Key Management Service Cryptographic Details.

Internetwork traffic privacy

AWS Payment Cryptography supports an AWS Management Console and a set of API operations that enable you to create and manage payment keys and use them in cryptographic operations.

AWS Payment Cryptography supports two network connectivity options from your private network to AWS.

- An IPSec VPN connection over the internet.
- AWS Direct Connect, which links your internal network to an AWS Direct Connect location over a standard Ethernet fiber-optic cable.

All Payment Cryptography API calls must be signed and be transmitted using Transport Layer Security (TLS). The calls also require a modern cipher suite that supports perfect forward secrecy. Traffic to the hardware security modules (HSMs) that store key material for payment keys is permitted only from known AWS Payment Cryptography API hosts over the AWS internal network.

To connect directly to AWS Payment Cryptography from your virtual private cloud (VPC) without sending traffic over the public internet, use VPC endpoints, powered by AWS PrivateLink. For more information, see Connecting to AWS Payment Cryptography through a VPC endpoint.

AWS Payment Cryptography also supports a hybrid post-quantum key exchange option for the Transport Layer Security (TLS) network encryption protocol. You can use this option with TLS when you connect to AWS Payment Cryptography API endpoints.

Resilience in AWS Payment Cryptography

AWS global infrastructure is built around AWS Regions and Availability Zones. Regions provide multiple physically separated and isolated Availability Zones, which are connected through low-latency, high-throughput, and highly redundant networking. With Availability Zones, you can design and operate applications and databases that automatically fail over between zones without interruption. Availability Zones are more highly available, fault tolerant, and scalable than traditional single or multiple data center infrastructures.

For more information about AWS Regions and Availability Zones, see AWS Global Infrastructure.

Regional isolation

AWS Payment Cryptography is a Regional service that is available in multiple regions.

The Regionally isolated design of AWS Payment Cryptography ensures that an availability issue in one AWS Region cannot affect AWS Payment Cryptography operation in any other Region. AWS Payment Cryptography is designed to ensure zero planned downtime, with all software updates and scaling operations performed seamlessly and imperceptibly.

The AWS Payment Cryptography Service Level Agreement (SLA) includes a service commitment of 99.99% for all Payment Cryptography APIs. To fulfill this commitment, AWS Payment Cryptography ensures that all data and authorization information required to execute an API request is available on all regional hosts that receive the request.

The AWS Payment Cryptography infrastructure is replicated in at least three Availability Zones (AZs) in each Region. To ensure that multiple host failures do not affect AWS Payment Cryptography performance, AWS Payment Cryptography is designed to service customer traffic from any of the AZs in a Region.

Changes that you make to the properties or permissions of a payment key are replicated to all hosts in the Region to ensure that subsequent request can be processed correctly by any host in the Region. Requests for cryptographic operations using your payment key are forwarded to a fleet of AWS Payment Cryptography hardware security modules (HSMs), any of which can perform the operation with the payment key.

Multi-tenant design

The multi-tenant design of AWS Payment Cryptography enables it to fulfill the availability SLA, and to sustain high request rates, while protecting the confidentiality of your keys and data.

Multiple integrity-enforcing mechanisms are deployed to ensure that the payment key that you specified for the cryptographic operation is always the one that is used.

The plaintext key material for your Payment Cryptography keys is protected extensively. The key material is encrypted in the HSM as soon as it is created, and the encrypted key material is immediately moved to secure storage. The encrypted key is retrieved and decrypted within the HSM just in time for use. The plaintext key remains in HSM memory only for the time needed to complete the cryptographic operation. Plaintext key material never leaves the HSMs; it is never written to persistent storage.

For more information about the mechanisms that AWS Payment Cryptography uses to secure your keys, see AWS Payment Cryptography Cryptographic Details.

Infrastructure security in AWS Payment Cryptography

As a managed service, AWS Payment Cryptography is protected by the AWS global network security procedures that are described in the Amazon Web Services: Overview of Security Processes whitepaper.

You use AWS published API calls to access AWS Payment Cryptography through the network. Clients must support Transport Layer Security (TLS) 1.2 or later. Clients must also support cipher suites with perfect forward secrecy (PFS) such as Ephemeral Diffie-Hellman (DHE) or Elliptic Curve Ephemeral Diffie-Hellman (ECDHE). Most modern systems such as Java 7 and later support these modes.

Additionally, requests must be signed using an access key ID and a secret access key that is associated with an IAM principal. Or you can use the AWS Security Token Service (AWS STS) to generate temporary security credentials to sign requests.

Isolation of physical hosts

The security of the physical infrastructure that AWS Payment Cryptography uses is subject to the controls described in the Physical and Environmental Security section of Amazon Web Services: Overview of Security Processes. You can find more detail in compliance reports and third-party audit findings listed in the previous section.

AWS Payment Cryptography is supported by dedicated commercial-off-the-shelf PCI PTS HSM-listed hardware security modules (HSMs). The key material for AWS Payment Cryptography keys is

stored only in volatile memory on the HSMs, and only while the Payment Cryptography key is in use. HSMs are in access controlled racks within Amazon data centers that enforce dual control for any physical access. For detailed information about the operation of AWS Payment Cryptography HSMs, see AWS Payment Cryptography Cryptographic Details.

Connecting to AWS Payment Cryptography through a VPC endpoint

You can connect directly to AWS Payment Cryptography through a private interface endpoint in your virtual private cloud (VPC). When you use an interface VPC endpoint, communication between your VPC and AWS Payment Cryptography is conducted entirely within the AWS network.

AWS Payment Cryptography supports Amazon Virtual Private Cloud (Amazon VPC) endpoints powered by AWS PrivateLink. Each VPC endpoint is represented by one or more Elastic Network Interfaces (ENIs) with private IP addresses in your VPC subnets.

The interface VPC endpoint connects your VPC directly to AWS Payment Cryptography without an internet gateway, NAT device, VPN connection, or AWS Direct Connect connection. The instances in your VPC do not need public IP addresses to communicate with AWS Payment Cryptography.

Regions

AWS Payment Cryptography supports VPC endpoints and VPC endpoint policies in all AWS Regions in which AWS Payment Cryptography is supported.

Topics

- Considerations for AWS Payment Cryptography VPC endpoints
- Creating a VPC endpoint for AWS Payment Cryptography
- Connecting to an AWS Payment Cryptography VPC endpoint
- Controlling access to a VPC endpoint
- Using a VPC endpoint in a policy statement
- Logging your VPC endpoint

Considerations for AWS Payment Cryptography VPC endpoints

Note

Although VPC endpoints allows you to connect to the service in as few as one availability zone(AZ), we recommend connecting to three availability zones for high availability and redundancy purposes.

Before you set up an interface VPC endpoint for AWS Payment Cryptography, review the Interface endpoint properties and limitations topic in the *AWS PrivateLink Guide*.

AWS Payment Cryptography support for a VPC endpoint includes the following.

- You can use your VPC endpoint to call all AWS Payment Cryptography Control plane operations and AWS Payment Cryptography Data plane operations from a VPC.
- You can create an interface VPC endpoint that connects to an AWS Payment Cryptography region endpoint.
- AWS Payment Cryptography consists of a control plane and a data plane. You can chose to setup one or both sub-services AWS PrivateLink but each is configured separately.
- You can use AWS CloudTrail logs to audit your use of AWS Payment Cryptography keys through the VPC endpoint. For details, see Logging your VPC endpoint.

Creating a VPC endpoint for AWS Payment Cryptography

You can create a VPC endpoint for AWS Payment Cryptography by using the Amazon VPC console or the Amazon VPC API. For more information, see Create an interface endpoint in the *AWS PrivateLink Guide*.

- To create a VPC endpoint for AWS Payment Cryptography, use the following service names:

```
com.amazonaws.region.payment-cryptography.controlplane
```

```
com.amazonaws.region.payment-cryptography.dataplane
```

For example, in the US West (Oregon) Region (`us-west-2`), the service names would be:

```
com.amazonaws.us-west-2.payment-cryptography.controlplane
```

```
com.amazonaws.us-west-2.payment-cryptography.dataplane
```

To make it easier to use the VPC endpoint, you can enable a private DNS name for your VPC endpoint. If you select the **Enable DNS Name** option, the standard AWS Payment Cryptography DNS hostname resolves to your VPC endpoint. For example, `https://controlplane.payment-cryptography.us-west-2.amazonaws.com` would resolve to a VPC endpoint connected to service name `com.amazonaws.us-west-2.payment-cryptography.controlplane`.

This option makes it easier to use the VPC endpoint. The AWS SDKs and AWS CLI use the standard AWS Payment Cryptography DNS hostname by default, so you do not need to specify the VPC endpoint URL in applications and commands.

For more information, see Accessing a service through an interface endpoint in the *AWS PrivateLink Guide*.

Connecting to an AWS Payment Cryptography VPC endpoint

You can connect to AWS Payment Cryptography through the VPC endpoint by using an AWS SDK, the AWS CLI or AWS Tools for PowerShell. To specify the VPC endpoint, use its DNS name.

For example, this list-keys command uses the `endpoint-url` parameter to specify the VPC endpoint. To use a command like this, replace the example VPC endpoint ID with one in your account.

```
$ aws payment-cryptography list-keys --endpoint-url https://
vpce-1234abcdf5678c90a-09p7654s-us-east-1a.ec2.us-east-1.vpce.amazonaws.com
```

If you enabled private hostnames when you created your VPC endpoint, you do not need to specify the VPC endpoint URL in your CLI commands or application configuration. The standard AWS Payment Cryptography DNS hostname resolves to your VPC endpoint. The AWS CLI and SDKs use this hostname by default, so you can begin using the VPC endpoint to connect to an AWS Payment Cryptography regional endpoint without changing anything in your scripts and applications.

To use private hostnames, the `enableDnsHostnames` and `enableDnsSupport` attributes of your VPC must be set to `true`. To set these attributes, use the ModifyVpcAttribute operation. For details, see View and update DNS attributes for your VPC in the *Amazon VPC User Guide*.

Controlling access to a VPC endpoint

To control access to your VPC endpoint for AWS Payment Cryptography, attach a *VPC endpoint policy* to your VPC endpoint. The endpoint policy determines whether principals can use the VPC endpoint to call AWS Payment Cryptography operations with specific AWS Payment Cryptography resources.

You can create a VPC endpoint policy when you create your endpoint, and you can change the VPC endpoint policy at any time. Use the VPC management console, or the CreateVpcEndpoint or ModifyVpcEndpoint operations. You can also create and change a VPC endpoint policy by using an AWS CloudFormation template. For help using the VPC management console, see Create an interface endpoint and Modifying an interface endpoint in the *AWS PrivateLink Guide*.

For help writing and formatting a JSON policy document, see the IAM JSON Policy Reference in the *IAM User Guide*.

Topics

- About VPC endpoint policies
- Default VPC endpoint policy
- Creating a VPC endpoint policy
- Viewing a VPC endpoint policy

About VPC endpoint policies

For an AWS Payment Cryptography request that uses a VPC endpoint to be successful, the principal requires permissions from two sources:

- An identity-based policy must give the principal permission to call the operation on the resource (AWS Payment Cryptography keys or alias).
- A VPC endpoint policy must give the principal permission to use the endpoint to make the request.

For example, a key policy might give a principal permission to call Decrypt on a particular AWS Payment Cryptography keys. However, the VPC endpoint policy might not allow that principal to call `Decrypt` on that AWS Payment Cryptography keys by using the endpoint.

Or a VPC endpoint policy might allow a principal to use the endpoint to call StopKeyUsage on certain AWS Payment Cryptography keys. But if the principal doesn't have those permissions from a IAM policy, the request fails.

Default VPC endpoint policy

Every VPC endpoint has a VPC endpoint policy, but you are not required to specify the policy. If you don't specify a policy, the default endpoint policy allows all operations by all principals on all resources over the endpoint.

However, for AWS Payment Cryptography resources, the principal must also have permission to call the operation from an IAM policy. Therefore, in practice, the default policy says that if a principal has permission to call an operation on a resource, they can also call it by using the endpoint.

```
{
  "Statement": [
    {
      "Action": "*",
      "Effect": "Allow",
      "Principal": "*",
      "Resource": "*"
    }
  ]
}
```

To allow principals to use the VPC endpoint for only a subset of their permitted operations, create or update the VPC endpoint policy.

Creating a VPC endpoint policy

A VPC endpoint policy determines whether a principal has permission to use the VPC endpoint to perform operations on a resource. For AWS Payment Cryptography resources, the principal must also have permission to perform the operations from an IAM policy.

Each VPC endpoint policy statement requires the following elements:

- The principal that can perform actions

- The actions that can be performed
- The resources on which actions can be performed

The policy statement doesn't specify the VPC endpoint. Instead, it applies to any VPC endpoint to which the policy is attached. For more information, see [Controlling access to services with VPC endpoints]() in the *Amazon VPC User Guide*.

The following is an example of a VPC endpoint policy for AWS Payment Cryptography. When attached to a VPC endpoint, this policy allows `ExampleUser` to use the VPC endpoint to call the specified operations on the specified AWS Payment Cryptography keys. Before using a policy like this one, replace the example principal and [key identifier]() with valid values from your account.

```
{
   "Statement":[
      {
         "Sid": "AllowDecryptAndView",
         "Principal": {"AWS": "arn:aws:iam::111122223333:user/ExampleUser"},
         "Effect":"Allow",
         "Action": [
             "payment-cryptography:Decrypt",
             "payment-cryptography:GetKey",
             "payment-cryptography:ListAliases",
             "payment-cryptography:ListKeys",
             "payment-cryptography:GetAlias"
          ],
         "Resource": "arn:aws:payment-cryptography:us-east-2:111122223333:key/
kwapwa6qaifllw2h"
      }
   ]
}
```

AWS CloudTrail logs all operations that use the VPC endpoint. However, your CloudTrail logs don't include operations requested by principals in other accounts or operations for AWS Payment Cryptography keys in other accounts.

As such, you might want to create a VPC endpoint policy that prevents principals in external accounts from using the VPC endpoint to call any AWS Payment Cryptography operations on any keys in the local account.

The following example uses the aws:PrincipalAccount global condition key to deny access to all principals for all operations on all AWS Payment Cryptography keys unless the principal is in the local account. Before using a policy like this one, replace the example account ID with a valid one.

```
{
  "Statement": [
    {
      "Sid": "AccessForASpecificAccount",
      "Principal": {"AWS": "*"},
      "Action": "payment-cryptography:*",
      "Effect": "Deny",
      "Resource": "arn:aws:payment-cryptography:*:111122223333:key/*",
      "Condition": {
        "StringNotEquals": {
          "aws:PrincipalAccount": "111122223333"
        }
      }
    }
  ]
}
```

Viewing a VPC endpoint policy

To view the VPC endpoint policy for an endpoint, use the VPC management console or the DescribeVpcEndpoints operation.

The following AWS CLI command gets the policy for the endpoint with the specified VPC endpoint ID.

Before using this command, replace the example endpoint ID with a valid one from your account.

```
$ aws ec2 describe-vpc-endpoints \
--query 'VpcEndpoints[?VpcEndpointId==`vpce-1234abcdf5678c90a`].[PolicyDocument]'
--output text
```

Using a VPC endpoint in a policy statement

You can control access to AWS Payment Cryptography resources and operations when the request comes from VPC or uses a VPC endpoint. To do so, use one an IAM policy

- Use the `aws:sourceVpce` condition key to grant or restrict access based on the VPC endpoint.

- Use the `aws:sourceVpc` condition key to grant or restrict access based on the VPC that hosts the private endpoint.

Note

The `aws:sourceIP` condition key is not effective when the request comes from an Amazon VPC endpoint. To restrict requests to a VPC endpoint, use the `aws:sourceVpce` or `aws:sourceVpc` condition keys. For more information, see Identity and access management for VPC endpoints and VPC endpoint services in the *AWS PrivateLink Guide*.

You can use these global condition keys to control access to AWS Payment Cryptography keys, aliases, and to operations like CreateKey that don't depend on any particular resource.

For example, the following sample key policy allows a user to perform particular cryptographic operations with a AWS Payment Cryptography keys only when the request uses the specified VPC endpoint, blocking access both from the Internet and AWS PrivateLink connections (if setup). When a user makes a request to AWS Payment Cryptography, the VPC endpoint ID in the request is compared to the `aws:sourceVpce` condition key value in the policy. If they do not match, the request is denied.

To use a policy like this one, replace the placeholder AWS account ID and VPC endpoint IDs with valid values for your account.

JSON

```
{
    "Id": "example-key-1",
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "EnableIAMPolicies",
            "Effect": "Allow",
            "Principal": {
                "AWS": [
                    "arn:aws:iam::111122223333:root"
                ]
            },
            "Action": [
```

```
                "payment-cryptography:*"
            ],
            "Resource": "*"
        },
        {
            "Sid": "RestrictUsageToMyVPCEndpoint",
            "Effect": "Deny",
            "Principal": "*",
            "Action": [
                "payment-cryptography:EncryptData",
                "payment-cryptography:DecryptData"
            ],
            "Resource": "arn:aws:payment-cryptography:us-east-1:111122223333:key/
*",
            "Condition": {
                "StringNotEquals": {
                    "aws:sourceVpce": "vpce-1234abcdf5678c90a"
                }
            }
        }

    ]
}
```

You can also use the `aws:sourceVpc` condition key to restrict access to your AWS Payment Cryptography keys based on the VPC in which VPC endpoint resides.

The following sample key policy allows commands that manage the AWS Payment Cryptography keys only when they come from `vpc-12345678`. In addition, it allows commands that use the AWS Payment Cryptography keys for cryptographic operations only when they come from `vpc-2b2b2b2b`. You might use a policy like this one if an application is running in one VPC, but you use a second, isolated VPC for management functions.

To use a policy like this one, replace the placeholder AWS account ID and VPC endpoint IDs with valid values for your account.

JSON

```
{
    "Id": "example-key-2",
    "Version":"2012-10-17",
```

```
    "Statement": [
        {
            "Sid": "AllowAdminActionsFromVPC12345678",
            "Effect": "Allow",
            "Principal": {
                "AWS": "111122223333"
            },
            "Action": [
               "payment-cryptography:Create*",
               "payment-cryptography:Encrypt*",
               "payment-cryptography:ImportKey*",
               "payment-cryptography:GetParametersForImport*",
               "payment-cryptography:TagResource",
               "payment-cryptography:UntagResource"
            ],
            "Resource": "*",
            "Condition": {
                "StringEquals": {
                    "aws:sourceVpc": "vpc-12345678"
                }
            }
        },
        {
            "Sid": "AllowKeyUsageFromVPC2b2b2b2b",
            "Effect": "Allow",
            "Principal": {
                "AWS": "111122223333"
            },
            "Action": [
                "payment-cryptography:Encrypt*",
                "payment-cryptography:Decrypt*"
            ],
            "Resource": "*",
            "Condition": {
                "StringEquals": {
                    "aws:sourceVpc": "vpc-2b2b2b2b"
                }
            }
        },
        {
            "Sid": "AllowListReadActionsFromEverywhere",
            "Effect": "Allow",
            "Principal": {
                "AWS": "111122223333"
```

```
            },
            "Action": [
                "payment-cryptography:List*",
                "payment-cryptography:Get*"
            ],
            "Resource": "*"
        }
    ]
}
```

Logging your VPC endpoint

AWS CloudTrail logs all operations that use the VPC endpoint. When a request to AWS Payment Cryptography uses a VPC endpoint, the VPC endpoint ID appears in the AWS CloudTrail log entry that records the request. You can use the endpoint ID to audit the use of your AWS Payment Cryptography VPC endpoint.

To protect your VPC, requests that are denied by a VPC endpoint policy, but otherwise would have been allowed, are not recorded in AWS CloudTrail.

For example, this sample log entry records a GenerateMac request that used the VPC endpoint. The `vpcEndpointId` field appears at the end of the log entry.

```
{
      "eventVersion": "1.08",
      "userIdentity": {
          "principalId": "TESTXECZ5U9M4LGF2N6Y5:i-98761b8890c09a34a",
          "arn": "arn:aws:sts::111122223333:assumed-role/samplerole/
i-98761b8890c09a34a",
          "accountId": "111122223333",
          "accessKeyId": "TESTXECZ5U2ZULLHHMJG",
          "sessionContext": {
              "sessionIssuer": {
                  "type": "Role",
                  "principalId": "TESTXECZ5U9M4LGF2N6Y5",
                  "arn": "arn:aws:iam::111122223333:role/samplerole",
                  "accountId": "111122223333",
                  "userName": "samplerole"
              },
              "webIdFederationData": {},
              "attributes": {
```

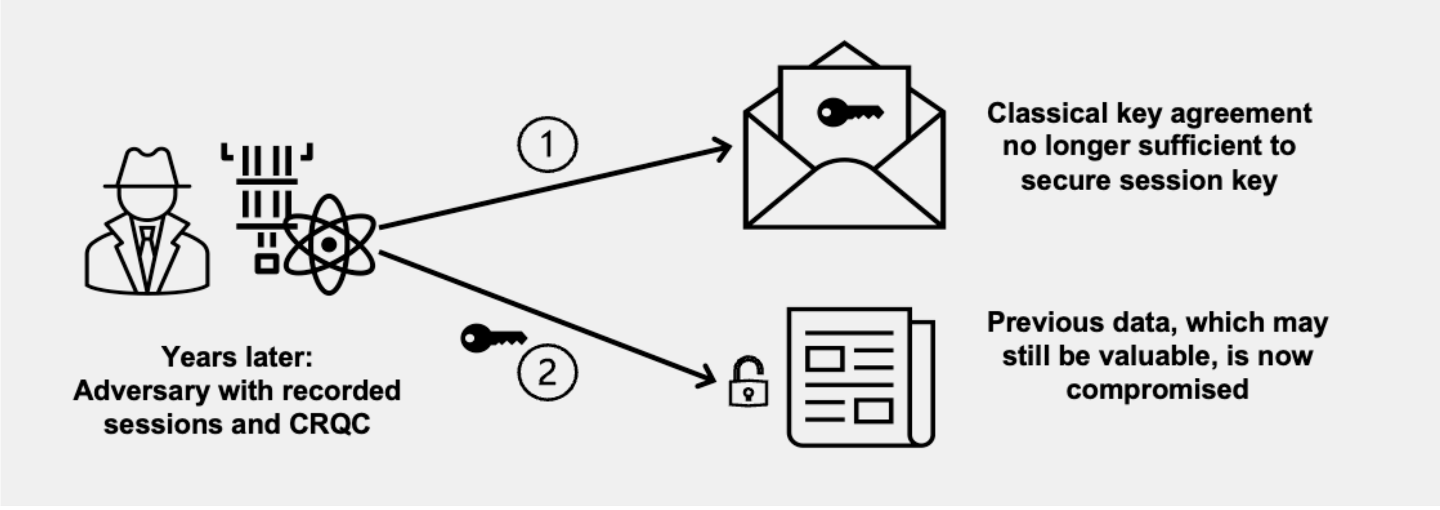
```
                "creationDate": "2024-05-27T19:34:10Z",
                "mfaAuthenticated": "false"
            },
            "ec2RoleDelivery": "2.0"
        }
    },
    "eventTime": "2024-05-27T19:49:54Z",
    "eventSource": "payment-cryptography.amazonaws.com",
    "eventName": "CreateKey",
    "awsRegion": "us-east-1",
    "sourceIPAddress": "172.31.85.253",
    "userAgent": "aws-cli/2.14.5 Python/3.9.16 Linux/6.1.79-99.167.amzn2023.x86_64
 source/x86_64.amzn.2023 prompt/off command/payment-cryptography.create-key",
    "requestParameters": {
        "keyAttributes": {
            "keyUsage": "TR31_M1_ISO_9797_1_MAC_KEY",
            "keyClass": "SYMMETRIC_KEY",
            "keyAlgorithm": "TDES_2KEY",
            "keyModesOfUse": {
                "encrypt": false,
                "decrypt": false,
                "wrap": false,
                "unwrap": false,
                "generate": true,
                "sign": false,
                "verify": true,
                "deriveKey": false,
                "noRestrictions": false
            }
        },
        "exportable": true
    },
    "responseElements": {
        "key": {
            "keyArn": "arn:aws:payment-cryptography:us-east-2:111122223333:key/
kwapwa6qaifllw2h",
            "keyAttributes": {
                "keyUsage": "TR31_M1_ISO_9797_1_MAC_KEY",
                "keyClass": "SYMMETRIC_KEY",
                "keyAlgorithm": "TDES_2KEY",
                "keyModesOfUse": {
                    "encrypt": false,
                    "decrypt": false,
                    "wrap": false,
```

```
                        "unwrap": false,
                        "generate": true,
                        "sign": false,
                        "verify": true,
                        "deriveKey": false,
                        "noRestrictions": false
                    }
                },
                "keyCheckValue": "A486ED",
                "keyCheckValueAlgorithm": "ANSI_X9_24",
                "enabled": true,
                "exportable": true,
                "keyState": "CREATE_COMPLETE",
                "keyOrigin": "AWS_PAYMENT_CRYPTOGRAPHY",
                "createTimestamp": "May 27, 2024, 7:49:54 PM",
                "usageStartTimestamp": "May 27, 2024, 7:49:54 PM"
            }
        },
        "requestID": "f3020b3c-4e86-47f5-808f-14c7a4a99161",
        "eventID": "b87c3d30-f3ab-4131-87e8-bc54cfef9d29",
        "readOnly": false,
        "eventType": "AwsApiCall",
        "managementEvent": true,
        "recipientAccountId": "111122223333",
        "vpcEndpointId": "vpce-1234abcdf5678c90a",
        "eventCategory": "Management",
        "tlsDetails": {
            "tlsVersion": "TLSv1.3",
            "cipherSuite": "TLS_AES_128_GCM_SHA256",
            "clientProvidedHostHeader": "vpce-1234abcdf5678c90a-
oo28vrvr.controlplane.payment-cryptography.us-east-1.vpce.amazonaws.com"
        }
    }
```


Using hybrid post-quantum TLS

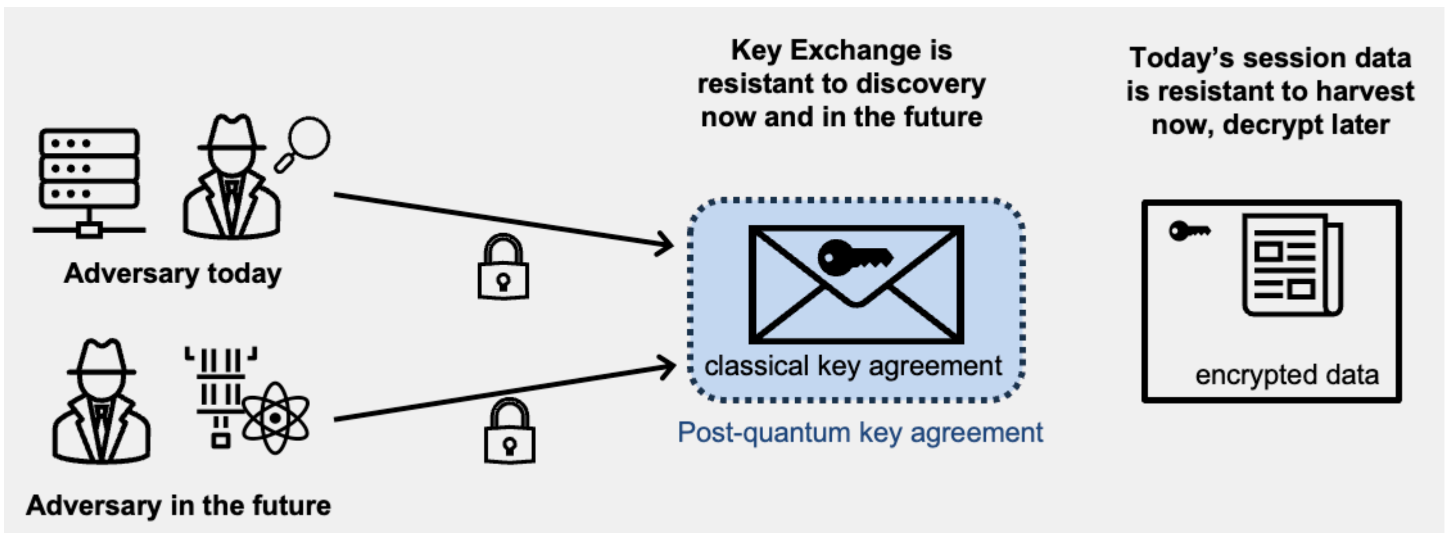
AWS Payment Cryptography and many other services supports a hybrid post-quantum key exchange option for the Transport Layer Security (TLS) network encryption protocol. You can use this TLS option when you connect to API endpoints or when using the AWS SDKs. These optional hybrid post-quantum key exchange features are at least as secure as the TLS encryption we use today and are likely to provide additional long-term security benefits.

The data that you send to enabled services is protected in transit by the encryption provided by a Transport Layer Security (TLS) connection. The classic cipher suites based on RSA and ECC that AWS Payment Cryptography supports for TLS sessions make brute force attacks on the key exchange mechanisms infeasible with current technology. However, if large scale or cryptographically relevant quantum computers (CRQC) becomes practical in the future, the existing TLS key exchange mechanisms will be susceptible to these attacks. It is possible that adversaries may begin harvesting encrypted data now with the hope that they can decrypt it in the future (harvest now, decrypt later). If you're developing applications that rely on the long-term confidentiality of the data passed over a TLS connection, you should consider a plan to migrate to post-quantum cryptography before large-scale quantum computers become available for use. AWS is working to prepare for this future, and we want you to be well-prepared, too.



To protect data encrypted today against potential future attacks, AWS is participating with the cryptographic community in the development of quantum-resistant or *post-quantum* algorithms. AWS has implemented *hybrid* post-quantum key exchange cipher suites that combine classic and post-quantum elements to ensure that your TLS connection is at least as strong as it would be with classic cipher suites.

These hybrid cipher suites are available for use on your production workloads when using recent versions of the AWS SDKs. For more information about how to enable/disable this behavior, please see ???



About hybrid post-quantum key exchange in TLS

The algorithms that AWS uses are a *hybrid* that combines Elliptic Curve Diffie-Hellman (ECDH), a classic key exchange algorithm used today in TLS, with Module-Lattice-Based Key-Encapsulation Mechanism (ML-KEM), a public-key encryption and key-establishment algorithm that the National Institute for Standards and Technology (NIST) has designated as its first standard post-quantum key-agreement algorithm. This hybrid uses each of the algorithms independently to generate a key. Then it combines the two keys cryptographically.

Learn more about PQC

For information about the post-quantum cryptography project at the National Institute for Standards and Technology (NIST), see Post-Quantum Cryptography.

For information about NIST post-quantum cryptography standardization, see Post-Quantum Cryptography Standardization.

Enabling hybrid post-quantum TLS

AWS SDKs and tools have cryptographic capabilities and configuration that differ across language and runtime. There are three ways that an AWS SDK or tool currently provides PQ TLS support:

Topics

- SDKs with PQ TLS enabled by default
- Opt-in PQ TLS support
- SDKs that rely on System OpenSSL

- AWS SDKs and tools not planning to support PQ TLS

SDKs with PQ TLS enabled by default

Note

As of 6-Nov-2025, AWS SDK and its underlying CRT libraries for MacOS and Windows uses system libraries for TLS, so PQ TLS capabilities on those platforms is generally determined by system-level support.

AWS SDK for Go

The AWS SDK for Go uses Golang's own TLS implementation provided by its standard library. Golang supports and prefers PQ TLS as of v1.24, so AWS SDK for Go users can enable PQ TLS by simply upgrading Golang to v1.24

AWS SDK for JavaScript (browser)

The AWS SDK for JavaScript (browser) uses the browser's TLS stack, so the SDK will negotiate PQ TLS if the browser runtime supports and prefers it. Firefox launched support for PQ TLS in v132.0. Chrome announced support for PQ TLS in v131. Edge supports opt-in PQ TLS in v120 for desktop and 140 for Android.

AWS SDK for Node.js

As of Node.js v22.20 (LTS) and v24.9.0, Node.js statically links and bundles OpenSSL 3.5. This means that PQ TLS is enabled and preferred by default for those and subsequent versions.

AWS SDK for Kotlin

The Kotlin SDK supports and prefers PQ TLS on Linux as of v1.5.78. Because AWS SDK for Kotlin's CRT-based client relies on system libraries for TLS on MacOS and Windows, support for PQ TLS will depend on those underlying system libraries.

AWS SDK for Rust

The AWS SDK for Rust distributes distinct packages (known as "crates" in the Rust ecosystem) for each service client. These are all managed in a consolidated GitHub repository, but each

service client follows its own version and release cadence. The consolidated SDK released PQ TLS preference on 8/29/25, so any individual service client version released after that date will support and prefer PQ TLS by default.

You can determine the minimum version supporting PQ TLS for a particular service client by navigating to the relevant crates.io version URL (for example, AWS Payment Cryptography's is here) and finding the first version published after 29-Aug-25. Any service client version published after 29-Aug-25 will have PQ TLS enabled and preferred by default.

Opt-in PQ TLS support

AWS SDK for C++

By default, the C++ SDK uses platform-native clients like libcurl and WinHttp. Libcurl generally relies on system OpenSSL for TLS, so PQ TLS is only enabled by default if system OpenSSL is ≥ v3.5. You can override this default in C++ SDK v1.11.673 or later, and opt-in to the AwsCrtHttpClient which supports and enables PQ TLS by default.

Notes on Building for Opt-In PQ TLS You can fetch the SDK's CRT dependencies with this script. Building the SDK from source is described here and here, but note that you may need a few additional CMake flags:

```
-DUSE_CRT_HTTP_CLIENT=ON \
-DUSE_TLS_V1_2=OFF \
-DUSE_TLS_V1_3=ON \
-DUSE_OPENSSL=OFF \
```

AWS SDK for Java

As of v2, AWS SDK for Java provides an AWS Common Runtime (AWS CRT) HTTP Client that can be configured to perform PQ TLS. As of v2.35.11, the AwsCrtHttpClient enables and prefers PQ TLS by default wherever it's used.

SDKs that rely on System OpenSSL

Several AWS SDKs and tools depend on the system's libcrypto/libssl library for TLS. The system library most often used is OpenSSL. OpenSSL enabled PQ TLS support in version 3.5, so the easiest

way to configure these SDKs and tools for PQ TLS is to use it on an operating system distribution that has at least OpenSSL 3.5 installed.

You can also configure a Docker container to use OpenSSL 3.5 to enable PQ TLS on any system that supports Docker. See Post-quantum TLS in Python for an example of setting this up for Python.

AWS CLI

PQ TLS support with the AWS CLI installer is coming soon. To enable immediately, you can use alternative installers for the AWS CLI, which varies by operating system, and can enable PQ TLS.

For MacOS, install the AWS CLI via Homebrew and ensure that your Homebrew-vended OpenSSL is upgraded to version 3.5+. You can do this with “brew install openssl@3.6” and validate with “brew list | grep openssl”.

For Ubuntu or Debian Linux: ensure the Linux distribution you are using has OpenSSL 3.5+ installed as system OpenSSL. Then, install the AWS CLI using apt or PyPI. With these prerequisites, the AWS CLI vended by apt or PyPI will be configured to negotiate PQ-TLS. For step-by-step instructions to validate the installation, see github repository and accompanying blog post.

AWS SDK for PHP

The AWS SDK for PHP relies on system libssl/libcrypto. To use PQ TLS, use this SDK on an operating system distribution that has at least OpenSSL 3.5 installed.

AWS SDK for Python (Boto3)

The AWS SDK for Python (Boto3) relies on system libssl/libcrypto. To use PQ TLS, use this SDK on an operating system distribution that has at least OpenSSL 3.5 installed.

AWS SDK for Ruby

The AWS SDK for Ruby relies on system libssl/libcrypto. To use PQ TLS, use this SDK on an operating system distribution that has at least OpenSSL 3.5 installed.

AWS SDK for .NET

On Linux, AWS SDK for .NET relies on system libssl/libcrypto. To use PQ TLS, use this SDK on an operating system distribution that has at least OpenSSL 3.5 installed. On Windows and MacOS, PQ TLS is available starting in .NET 10 and Windows 11. On MacOS, TLS 1.3 support (a prerequisite for

PQ TLS) can be enabled by opting-in to Apple's Network.framework as described here. Assuming a minimum .NET version of 10, PQ TLS should then be enabled.

AWS SDKs and tools not planning to support PQ TLS

There are currently no plans to support the following language SDKs and tools:

- AWS SDK for SAP
- AWS SDK for Swift
- AWS Tools for Windows PowerShell

Security best practices for AWS Payment Cryptography

AWS Payment Cryptography supports many security features that are either built-in or that you can optionally implement to enhance the protection of your encryption keys and ensure that they are used for their intended purpose, including IAM policies, an extensive set of policy condition keys to refine your key policies and IAM policies and built-in enforcement of PCI PIN rules regarding key blocks.

> **Important**
>
> The general guidelines provided do not represent a complete security solution. Because not all best practices are appropriate for all situations, these are not intended to be prescriptive.

- **Key Usage and Modes of Use**: AWS Payment Cryptography follows and enforces key usage and mode of use restrictions as described in ANSI X9 TR 31-2018 Interoperable Secure Key Exchange Key Block Specification and consistent with PCI PIN Security Requirement 18-3. This limits the ability to use a single key for multiple purposes and cryptographically binds the key metadata (such as permitted operations) to the key material itself. AWS Payment Cryptography automatically enforces these restrictions such as that a key encryption key (TR31_K0_KEY_ENCRYPTION_KEY) cannot also be used for data decryption. See Understanding key attributes for AWS Payment Cryptography key for more details.
- **Limit sharing of symmetric key material**: Only share symmetric key material (such as Pin Encryption Keys or Key Encryption Keys) with at most one other entity. If there is a need to transit sensitive material to more entities or partners, create additional keys. AWS Payment

Cryptography never exposes symmetric key material or asymmetric private key material in the clear.

- **Use aliases or tags to associate keys with certain use cases or partners**: Aliases can be used to easily denote the use case associated with a key such as alias/BIN_12345_CVK to denote a card verification key associated with BIN 12345. To provide more flexibility, consider creating tags such as bin=12345, use_case=acquiring,country=us,partner=foo. Aliases and tags can also be used to limit access such as enforcing access controls between issuing and acquiring use cases.
- **Practice least privileged access**: IAM can be used to limit production access to systems rather than individuals, such as prohibiting individual users from creating keys or running cryptographic operations. IAM can also be used to limit access to both commands and keys that may not be applicable for your use case, such as limiting the ability to generate or validate pins for an acquirer. Another way to use least privileged access is to restrict sensitive operations (such as key import) to specific service accounts. See AWS Payment Cryptography identity-based policy examples for examples.

See also

- Identity and access management for AWS Payment Cryptography
- Security best practices in IAM in the *IAM User Guide*

Compliance validation for AWS Payment Cryptography

As with other AWS services, customers require a clear understanding of the shared responsibility model for security and compliance. As a service that specifically supports payments, compliance with applicable PCI standards is particularly important to understand for AWS Payment Cryptography customers. AWS PCI DSS and PCI 3DS assessments include AWS Payment Cryptography. There may be references to the service in the Shared Responsibility Guides, available from AWS Artifact, for these reports. PCI PIN Security and Point-to-Point Encryption (P2PE) assessments are specific to AWS Payment Cryptography.

This section provides information on the status and scope of the service's compliance and information that will be helpful in planning PCI PIN Security and PCI P2PE assessments of your applications.

Topics

- Compliance of the service
- PIN Compliance Planning
- Using the AWS Payment Cryptography Decryption Component in P2PE solutions

Compliance of the service

Third-party auditors assess the security and compliance of AWS Payment Cryptography as part of multiple AWS compliance programs. These include SOC, PCI, and others.

AWS Payment Cryptography has been assessed for several PCI standards in addition to PCI DSS and PCI 3DS. These include PCI PIN Security (PCI PIN) and PCI Point-to-Point (P2PE) Encryption. Please see AWS Artifact for available attestations and compliance guides.

For a list of AWS services in scope of specific compliance programs, see AWS Services in Scope by Compliance Program. For general information, see AWS Compliance Programs.

You can download third-party audit reports using AWS Artifact. For more information, see Downloading Reports in AWS Artifact.

Your compliance responsibility when using AWS Payment Cryptography is determined by the sensitivity of your data, your company's compliance objectives, and applicable laws and regulations. AWS provides the following resources to help with compliance:

- Security and Compliance Quick Start Guides—These deployment guides discuss architectural considerations and provide steps for deploying security- and compliance-focused baseline environments on AWS.
- AWS Compliance Resources—This collection of workbooks and guides might apply to your industry and location.
- Evaluating Resources with Rules in the *AWS Config Developer Guide*—AWS Config; assesses how well your resource configurations comply with internal practices, industry guidelines, and regulations.
- AWS Security Hub CSPM—This AWS service provides a comprehensive view of your security state within AWS that helps you check your compliance with security industry standards and best practices.

PIN Compliance Planning

This guide describes the documentation and evidence that you will need to prepare for a PCI PIN assessment of your PIN processing application that uses AWS Payment Cryptography.

As with other AWS services and compliance standards, it is your responsibility to use the service securely, configuring access control and using security parameters in alignment with PCI PIN requirements. This guide will discuss those configurations when appropriate to a meeting a requirement.

Topics

- Common Topics
- Assessment Scope
- Transaction Processing Operations

Common Topics

Migrating applications from connecting to HSM to a managed service such as AWS Payment Cryptography brings up common issues and concepts for customers and their assessors. This section provides information to clarify how the secure use of the service addresses these situations.

Topics

- Shared Responsibility

- Minimum HSM configuration
- Key exchange between customer and APC

Shared Responsibility

Customers that have assumed complete security and compliance responsibility for applications will restructure their compliance to take advantage of AWS Payment Cryptography's key management, security controls and managed HSM capabilities ("the service"). This will shift some requirements completely to AWS, as attested by AWS Payment Cryptography's third party assessments. Some requirements will be shared between the customer's application and the service. An application is responsible for:

- Providing accurate information to the service
- Using security controls according to the service's recommendations and PCI PIN security requirements
- Implementing required security controls using the tools provided by the service

Customers and their assessors will use shared responsibility and implementation guides published with compliance attestations on AWS Artifact to implement controls and control monitoring then plan for and complete assessments.

Minimum HSM configuration

The PCI Data Security Standard, the foundational standard for other PCI standards, requires all systems to be configured with minimum functionality needed for its function. PCI PIN, P2PE, and other solution standards apply this requirement to HSMs in the solution. HSMs must only enable functions needed for the solution.

AWS services should be treated as systems and configured for minimum required functionality. Payment Card Industry Data Security Standard (PCI DSS) v4.0 on AWS recommends using IAM to configure minimum functionality for each AWS service used by the solution. This also applies to AWS Payment Cryptography. IAM policies enable fine-grained permissions to restrict cryptographic functions only to the application components that rely on them.

Key exchange between customer and APC

PIN PIN Security requirements 8-4 and 15-2 require public keys for exchange and loading of keys are authenticated and integrity protected. For remote key loading of POI, functionally described

in ANSI/ASC X9 TR-34 and governed by PCI PIN Annex A, public keys are most often conveyed in certificates signed by an Annex A2-compliant certificate authority. For exchanges between organization, public keys use other mechanisms for authenticity and integrity.

All interactions between customer and AWS are via AWS APIs, which mutually authenticate each API call and assure the integrity of calls and responses using TLS. Authentication of the customer application is managed by AWS Identity and Access Management with mechanisms such as Security Tokens and SigV4. AWS API endpoints are authenticated by the customer using TLS server authentication, which is built into AWS SDKs. Then TLS assures the confidentiality and integrity of all data passed between the customer and each AWS API.

APC APIs GetParametersForImport and ImportKey implement a key transfer from the customer to the service. While the Certificate Authority(CA) provided by GetParametersForImport is not Annex A2-compliant, it is secure and unique to the account. While this CA cannot be relied on for compliance with requirements 8-4 and 15-2, it does provide integrity verification of imported key. You can also use your own CA by leveraging the GetCertificateSigningRequest API.

The mechanisms that provide public key authentication and integrity assurance are:

- Authentication provided by AWS API authentication
- Integrity of the key is provided by the MAC feature of the certificate provided by GetParametersForImport, even if the identity information in the certificate is not trusted. Integrity of the key is also assured by MAC used by TLS protecting the session between the customer and AWS

Certificates and key blocks provided by APC are compliant with Annex A1, which specifies requirements for certificates and key protection by asymmetric methods.

Assessment Scope

The first step in planning any assessment is documenting the scope. For PCI PIN, the scope is systems and processes that protect PINs, including protection of the cryptographic keys and devices that protect them - payment terminals, also called points-of-interaction (POI), HSMs, and other secure cryptographic devices (SCD).

We will not address requirements where you retain full responsibility because these address areas outside of the scope of the service. For example, configuration and provisioning of payment terminals. Refer to the AWS Payment Cryptography Shared Responsibility Guide for PCI PIN, available on AWS Artifact

Topics

- Shared Responsibility
- High-Level Network Diagrams
- Key Table
- Document References

Shared Responsibility

AWS Payment Cryptography is an Encryption and Support Organization (ESO) and a PIN-Acquiring Third-Party Servicer (TPS), as defined by the Visa PIN Security Program and listed on the Visa Global Service Provider Registry, under "Amazon Web Services, LLC". This means that the service is allowed by Visa to be used by PIN-Acquiring Third-Party VisaNet Processor (VNP), PIN-Acquiring Client VisaNet Processor Acting as a Service Provider, and other TPS and ESO providers without requiring further assessment by customer PIN assessors (PCI Qualified PIN Assessors or PCI QPA).

Other card brands or payment network providers may rely on the Visa PIN Security Program or have their own programs. Contact AWS Support for questions about service compliance for other payment network programs.

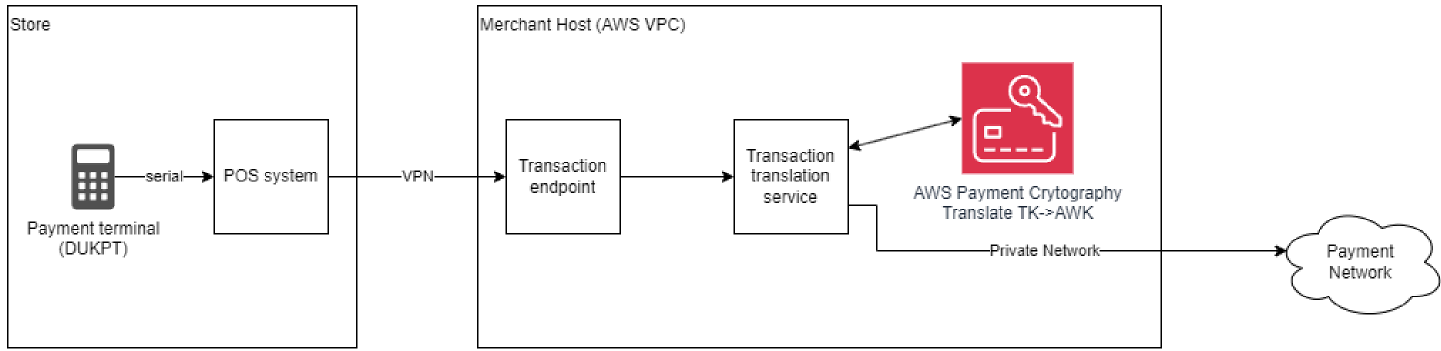
AWS provides the PCI PIN Security attestation of compliance (AOC) and Shared Responsibility Guide for AWS Payment Cryptography in AWS Artifact. Use of service providers in PIN processing has be common for many years, however, the PCI PIN Security Standard, up through version 3.1, does not address third party service provider management. Neither does the Visa PIN Security Program. Customer QPA have followed the model established with the PCI DSS AOC and Shared Responsibility Guide of referring to AWS' compliance as successful the test for applicable requirements.

High-Level Network Diagrams

The PCI PIN Reporting Template requires, "For entities engaged in the processing of PIN based transaction provide a network schematic describing PIN based transaction flows with the associated key type usage. Additionally, KIFs and entities engaged in remote key distribution using asymmetric techniques should provide keying material flows"

AWS Payment Cryptography has reported the internal service structure for our PCI PIN assessment. Your diagrams will illustrate calling the service APIs for PIN processing.

Example high level network diagram for a PIN applications using AWS Payment Cryptography:



Key Table

The report requires that all keys protecting PINs, directly or indirectly, are listed. Any keys that exist in the service can be listed with the ListKeysAPI .

Be sure to provide the key list for all regions and accounts that own keys for your application.

Document References

Vendor documentation and recommendations for secure use of AWS Payment Cryptography is in the User's Guide and API Reference. These are linked, as appropriate, in this guidance.

Transaction Processing Operations

PCI PIN requirements are organized in Control Objectives. Each Control Objective groups requirements for securing an aspect of security for PINs.

Topics

- Control Objective 1: PINs used in transactions governed by these requirements are processed using equipment and methodologies that ensure they are kept secure.
- Control Objective 2: Cryptographic keys used for PIN encryption/decryption and related key management are created using processes that ensure that it is not possible to predict any key or determine that certain keys are more probable than other keys.
- Control Objective 3: Keys are conveyed or transmitted in a secure manner.
- Control Objective 4: Key-loading to HSMs and POI PIN-acceptance devices is handled in a secure manner.
- Control Objective 5: Keys are used in a manner that prevents or detects their unauthorized usage.
- Control Objective 6: Keys are administered in a secure manner.

- Control Objective 7: Equipment used to process PINs and keys is managed in a secure manner.

Control Objective 1: PINs used in transactions governed by these requirements are processed using equipment and methodologies that ensure they are kept secure.

Requirement 1: HSMs used by AWS Payment Cryptography were assessed as part of our PCI PIN assessment. For customers using the service, Requirement 1-3 and 1-4 are “In Place” relative to the HSM managed by the service. The findings for HSM will state that testing was attested to by the AWS QPA. The PIN Attestation of Compliance is available to be referenced on AWS Artifact. Other SCD, like POI, in your solution will need to be inventoried and referenced.

Requirement 2: Documentation of your procedures must specify how cardholder PINs are protected with regards to divulging to your personnel, the PIN translation protocol(s) implemented, and protection during on-line and off-line processing. In addition, your documentation should contain summary of cryptographic key management methods used within each zone.

Requirement 3: POI must be configured for secure PIN encryption and transmission. AWS Payment Cryptography supports only PIN block translations specified in Requirement 3-3.

Requirement 4: The application must not store PIN blocks. The PIN blocks, even encrypted, must not be retained in transaction journals or logs. The service does not store PIN blocks and the PIN assessment verifies that they are not in logs.

Note that the PCI PIN Security standard is applies to acquiring “the secure management, processing, and transmission of personal identification number (PIN) data during online and offline payment card transaction processing at ATMs and point-of-sale (POS) terminals”, as stated in the standard. However, the standard is often used for assessing cryptographic key management for payments outside of that intended scope. This may include issuer use cases where PINs are stored. Exceptions to requirements for these cases should be agreed with intended audience for the assessment.

Control Objective 2: Cryptographic keys used for PIN encryption/decryption and related key management are created using processes that ensure that it is not possible to predict any key or determine that certain keys are more probable than other keys.

Requirement 5: Key generation by AWS Payment Cryptography was assessed as part of our PCI PIN assessment. This can be specified in the key table “Generated by” column.

Requirement 6: Security controls for keys held in AWS Payment Cryptography were assessed as part of the service’s PCI PIN assessment. Include descriptions of security controls pertaining to key generation within your application and with any other service providers.

Requirement 7: You must have a key-generation policy documentation which should specify how keys are generated and all affected parties must be aware of these procedures/policies. Procedures for key creation using the APC API should include use of roles with key creation permissions and approvals for running scripts or other code that creates keys. AWS CloudTrail logs contain all CreateKey events with date and time, key ARN, and user ids. HSM serial numbers and logs for access to physical media was assessed as part of the service’s PIN assessment.

Control Objective 3: Keys are conveyed or transmitted in a secure manner.

Requirement 8: Key conveyance with AWS Payment Cryptography was assessed as part of our PCI PIN assessment. You will need to document key protection mechanisms for transfers prior to import to and after export from AWS Payment Cryptography. The service provides key check values for all keys to validate correct conveyance.

Requirement 8-4 requires that public keys are conveyed in a manner that protects their integrity and authenticity. Conveyance between your application and AWS is controlled by the application’s authentication to AWS, using AWS Identity and Access Management methods, AWS’ API end point authentication to the application via TLS server certificates. Additionally, public keys exported from or imported to AWS Payment Cryptography have certificates signed by ephemeral, customer-specific CAs (See GetPublicKeyCertificate, GetParametersForImport, and GetParametersForExport). These CAs cannot be used as the sole method of authentication, because they are not compliant with PCI PIN Security Annex A2. However, the certificates still provide integrity assurance for public keys with IAM providing authentication.

When exchanging public keys with your business partners using asymmetric methods, you must provide for authentication of the business via the communications channel, using a secure file exchange website, for example.

Requirement 9: The service does not use or directly support clear text key components.

Requirement 10: The service enforces relative key strength of protecting keys for conveyance. You are responsible for key conveyance prior to import to and after export from AWS Payment Cryptography and using API and TR-31 parameters that are accurate for key import, export, and generation. You should have documented procedures to describe the key conveyance mechanisms and the list of cryptographic keys used for the conveyance.

Requirement 11: Documentation of your procedures must specify how keys are conveyed. Procedures for key conveyance using the AWS Payment Cryptography API should include use of roles with key import and export permissions and approvals for running scripts or other code that creates keys. AWS CloudTrail logs contain all ImportKey and ExportKey events.

Control Objective 4: Key-loading to HSMs and POI PIN-acceptance devices is handled in a secure manner.

Requirement 12: You are responsible for loading keys from components or shares. Management of HSM main keys was assessed as part of the service's PIN assessment. AWS Payment Cryptography does not load keys from individual shares or components. See the Cryptographic details section.

Requirements 13 and 14: You will need to describe key protection for transfers prior to import to and after export from the service.

Requirement 15: AWS Payment Cryptography provides key check values for all keys in the service and integrity assurance for public keys. Your application is responsible for using these checks to validate keys after import to or export from the service. You should document the procedures to ensure that a validation mechanism in place.

Requirement 15-2 requires that public keys are loaded in a manner that protects their integrity and authenticity. ImportKey, together with GetParametersForImport, provides for validation of provided signing certificates. If provided certificates are self-signed, than authentication must be provided by a separate mechanism, for example secure file exchange.

Requirement 16: Documentation of your procedures must specify how keys are loaded to the service. Procedures for key import using the API should include use of roles with key import permissions and approvals for running scripts or other code that loads keys. AWS CloudTrail logs contain all ImportKey events. You should include the logging mechanisms in the documentation. The service provides key check values for all keys to validate correct key loading.

Control Objective 5: Keys are used in a manner that prevents or detects their unauthorized usage.

Requirement 17: The service provides mechanisms, such as tags and aliases, for keys that enable tracking of key sharing relationships. Additionally, key check values should be kept separately to demonstrate that known or default key values are not used when keys are shared.

Requirement 18: The service provides key integrity checks, via GetKey and ListKeys, and key management events, via AWS CloudTrail, that can be used to detect unauthorized substitution or monitor synchronization of keys between parties. The service stores keys exclusively in key blocks. You are responsible for key storage and use prior to import to and after export from AWS Payment Cryptography.

You should have procedures in place for an immediate investigation should any discrepancy occur during processing of PIN based transactions or unexpected key management events.

Requirement 19: The service uses keys exclusively in key blocks, enforcing KeyUsage, KeyModeOfUse, and other key attributes for all operations. This includes restriction on private key operations. You should use your public keys for a single purpose e:g encryption or digital signature verification but not both. You should use separate accounts for production and test/development systems.

Requirement 20: You retain responsibility for this requirement.

Control Objective 6: Keys are administered in a secure manner.

Requirement 21: Key storage and use with AWS Payment Cryptography was assessed as part of the service's PCI PIN assessment. For key component related storage requirements, you are responsible to store them as delineated under 21-2 and 21-3. You will need to describe key protection mechanisms in your policy documentation prior to import to and after export from the service.

Requirement 22: Key compromise procedures for AWS Payment Cryptography were assessed as part of the service's PCI PIN assessment. You will need to describe key compromise detection and response procedures, including monitoring and response to notification from AWS.

Requirement 23: AWS Payment Cryptography does not support variants or other reversible key calculation methods. APC main keys or keys enciphered by them are never available to customers. Use of reversible key calculation was assessed as part of the service's PCI PIN assessment.

Requirement 24: Destruction practices for internal secret and private keys AWS Payment Cryptography was assessed as part of the service’s PCI PIN assessment. You will need to describe key destruction procedure for keys prior to import to and after export from APC. Key component related destruction requirements (24-2.2 and 24-2.3) remain your responsibility.

Requirement 25: Access to secret and private keys within AWS Payment Cryptography was assessed as part of the service's PCI PIN assessment. You will need to have a process and documentation for access controls for keys prior to import to and after export from AWS Payment Cryptography.

Requirement 26: You will need to describe logging for any access to keys, key components, or related materials used outside of the service. Logs for all key management activities that your application does with the service are available via AWS CloudTrail.

Requirement 27: You will need to describe backup procedures for keys, key components, or related materials used outside of the service.

Requirement 28: Procedures for all key administration using the API should include use of roles with key administration permissions and approvals for running scripts or other code that manages keys. AWS CloudTrail logs contain all key administration events

Control Objective 7: Equipment used to process PINs and keys is managed in a secure manner.

Requirement 29: Your requirements for physical and logical protections for HSMs are met by use of AWS Payment Cryptography.

Requirement 30: Your application will retain responsibility for all physical and logical protection of POI device requirements.

Requirement 31: Protection of secure cryptographic devices (SCD) used by AWS Payment Cryptography was assessed as part of the service’s PCI PIN assessment. You will need to demonstrate protection of any other SCDs used by your application.

Requirement 32: Use of SCDs used by AWS Payment Cryptography was assessed as part of the service’s PCI PIN assessment. You will need to demonstrate access control and protection of any other SCDs used by your application.

Requirement 33: You will need to describe protections of any PIN-processing equipment under your control.

Using the AWS Payment Cryptography Decryption Component in P2PE solutions

PCI P2PE Solutions can use the AWS Payment Cryptography Decryption Component. This is documented in the PCI Point-to-Point Encryption: Security Requirements and Testing Procedures, Section P2PE Solutions and Use of Third Parties and/or P2PE Component Providers: “A solution provider (or a merchant as a solution provider) can outsource certain P2PE functions to PCI-listed P2PE component providers and report use of the PCI-listed P2PE component(s) in their P2PE Report on Validation (P-ROV)”, which is available on the PCI website.

As with other AWS services and compliance standards, it is your responsibility to use the service securely, configuring access control and using security parameters in alignment with PCI P2PE requirements. The *AWS Payment Cryptography P2PE Decryption Component User’s Guide*, which is available on AWS Artifact, has detailed instructions for integrating AWS Payment Cryptography with your PCI P2PE Solution and the annual decryption component report, which is required for compliance reporting.

Identity and access management for AWS Payment Cryptography

AWS Identity and Access Management (IAM) is an AWS service that helps an administrator securely control access to AWS resources. IAM administrators control who can be *authenticated* (signed in) and *authorized* (have permissions) to use AWS Payment Cryptography resources. IAM is an AWS service that you can use with no additional charge.

Topics

- Audience
- Authenticating with identities
- Managing access using policies
- How AWS Payment Cryptography works with IAM
- AWS Payment Cryptography identity-based policy examples
- Troubleshooting AWS Payment Cryptography identity and access

Audience

How you use AWS Identity and Access Management (IAM) differs based on your role:

- **Service user** - request permissions from your administrator if you cannot access features (see Troubleshooting AWS Payment Cryptography identity and access)
- **Service administrator** - determine user access and submit permission requests (see How AWS Payment Cryptography works with IAM)
- **IAM administrator** - write policies to manage access (see AWS Payment Cryptography identity-based policy examples)

Authenticating with identities

Authentication is how you sign in to AWS using your identity credentials. You must be authenticated as the AWS account root user, an IAM user, or by assuming an IAM role.

You can sign in as a federated identity using credentials from an identity source like AWS IAM Identity Center (IAM Identity Center), single sign-on authentication, or Google/Facebook

credentials. For more information about signing in, see How to sign in to your AWS account in the *AWS Sign-In User Guide*.

For programmatic access, AWS provides an SDK and CLI to cryptographically sign requests. For more information, see AWS Signature Version 4 for API requests in the *IAM User Guide*.

AWS account root user

When you create an AWS account, you begin with one sign-in identity called the AWS account *root user* that has complete access to all AWS services and resources. We strongly recommend that you don't use the root user for everyday tasks. For tasks that require root user credentials, see Tasks that require root user credentials in the *IAM User Guide*.

IAM users and groups

An *IAM user* is an identity with specific permissions for a single person or application. We recommend using temporary credentials instead of IAM users with long-term credentials. For more information, see Require human users to use federation with an identity provider to access AWS using temporary credentials in the *IAM User Guide*.

An *IAM group* specifies a collection of IAM users and makes permissions easier to manage for large sets of users. For more information, see Use cases for IAM users in the *IAM User Guide*.

IAM roles

An *IAM role* is an identity with specific permissions that provides temporary credentials. You can assume a role by switching from a user to an IAM role (console) or by calling an AWS CLI or AWS API operation. For more information, see Methods to assume a role in the *IAM User Guide*.

IAM roles are useful for federated user access, temporary IAM user permissions, cross-account access, cross-service access, and applications running on Amazon EC2. For more information, see Cross account resource access in IAM in the *IAM User Guide*.

Managing access using policies

You control access in AWS by creating policies and attaching them to AWS identities or resources. A policy defines permissions when associated with an identity or resource. AWS evaluates these policies when a principal makes a request. Most policies are stored in AWS as JSON documents. For

more information about JSON policy documents, see Overview of JSON policies in the *IAM User Guide*.

Using policies, administrators specify who has access to what by defining which **principal** can perform **actions** on what **resources**, and under what **conditions**.

By default, users and roles have no permissions. An IAM administrator creates IAM policies and adds them to roles, which users can then assume. IAM policies define permissions regardless of the method used to perform the operation.

Identity-based policies

Identity-based policies are JSON permissions policy documents that you attach to an identity (user, group, or role). These policies control what actions identities can perform, on which resources, and under what conditions. To learn how to create an identity-based policy, see Define custom IAM permissions with customer managed policies in the *IAM User Guide*.

Identity-based policies can be *inline policies* (embedded directly into a single identity) or *managed policies* (standalone policies attached to multiple identities). To learn how to choose between managed and inline policies, see Choose between managed policies and inline policies in the *IAM User Guide*.

Resource-based policies

Resource-based policies are JSON policy documents that you attach to a resource. Examples include IAM *role trust policies* and Amazon S3 *bucket policies*. In services that support resource-based policies, service administrators can use them to control access to a specific resource. You must specify a principal in a resource-based policy.

Resource-based policies are inline policies that are located in that service. You can't use AWS managed policies from IAM in a resource-based policy.

Access control lists (ACLs)

Access control lists (ACLs) control which principals (account members, users, or roles) have permissions to access a resource. ACLs are similar to resource-based policies, although they do not use the JSON policy document format.

Amazon S3, AWS WAF, and Amazon VPC are examples of services that support ACLs. To learn more about ACLs, see Access control list (ACL) overview in the *Amazon Simple Storage Service Developer Guide*.

Other policy types

AWS supports additional policy types that can set the maximum permissions granted by more common policy types:

- **Permissions boundaries –** Set the maximum permissions that an identity-based policy can grant to an IAM entity. For more information, see Permissions boundaries for IAM entities in the *IAM User Guide*.
- **Service control policies (SCPs) –** Specify the maximum permissions for an organization or organizational unit in AWS Organizations. For more information, see Service control policies in the *AWS Organizations User Guide*.
- **Resource control policies (RCPs) –** Set the maximum available permissions for resources in your accounts. For more information, see Resource control policies (RCPs) in the *AWS Organizations User Guide*.
- **Session policies –** Advanced policies passed as a parameter when creating a temporary session for a role or federated user. For more information, see Session policies in the *IAM User Guide*.

Multiple policy types

When multiple types of policies apply to a request, the resulting permissions are more complicated to understand. To learn how AWS determines whether to allow a request when multiple policy types are involved, see Policy evaluation logic in the *IAM User Guide*.

How AWS Payment Cryptography works with IAM

Before you use IAM to manage access to AWS Payment Cryptography, you should understand what IAM features are available to use with AWS Payment Cryptography. To get a high-level view of how AWS Payment Cryptography and other AWS services work with IAM, see AWS Services That Work with IAM in the *IAM User Guide*.

Topics

- AWS Payment Cryptography Identity-based policies
- Authorization based on AWS Payment Cryptography tags

AWS Payment Cryptography Identity-based policies

With IAM identity-based policies, you can specify allowed or denied actions and resources as well as the conditions under which actions are allowed or denied. AWS Payment Cryptography supports specific actions, resources, and condition keys. To learn about all of the elements that you use in a JSON policy, see IAM JSON Policy Elements Reference in the *IAM User Guide*.

Actions

Administrators can use AWS JSON policies to specify who has access to what. That is, which **principal** can perform **actions** on what **resources**, and under what **conditions**.

The `Action` element of a JSON policy describes the actions that you can use to allow or deny access in a policy. Include actions in a policy to grant permissions to perform the associated operation.

Policy actions in AWS Payment Cryptography use the following prefix before the action: `payment-cryptography:`. For example, to grant someone permission to execute an AWS Payment Cryptography `VerifyCardData` API operation, you include the `payment-cryptography:VerifyCardData` action in their policy. Policy statements must include either an `Action` or `NotAction` element. AWS Payment Cryptography defines its own set of actions that describe tasks that you can perform with this service.

To specify multiple actions in a single statement, separate them with commas as follows:

```
"Action": [
      "payment-cryptography:action1",
      "payment-cryptography:action2"
```

You can specify multiple actions using wildcards (*). For example, to specify all actions that begin with the word `List` (such as `ListKeys` and `ListAliases`), include the following action:

```
"Action": "payment-cryptography:List*"
```

To see a list of AWS Payment Cryptography actions, see Actions Defined by AWS Payment Cryptography in the *IAM User Guide*.

Resources

Administrators can use AWS JSON policies to specify who has access to what. That is, which **principal** can perform **actions** on what **resources**, and under what **conditions**.

The `Resource` JSON policy element specifies the object or objects to which the action applies. As a best practice, specify a resource using its [Amazon Resource Name (ARN)](). For actions that don't support resource-level permissions, use a wildcard (*) to indicate that the statement applies to all resources.

```
"Resource": "*"
```

The payment-cryptography key resource has the following ARN:

```
arn:${Partition}:payment-cryptography:${Region}:${Account}:key/${keyARN}
```

For more information about the format of ARNs, see [Amazon Resource Names (ARNs) and AWS Service Namespaces]().

For example, to specify the `arn:aws:payment-cryptography:us-east-2:111122223333:key/kwapwa6qaifllw2h` instance in your statement, use the following ARN:

```
"Resource": "arn:aws:payment-cryptography:us-east-2:111122223333:key/kwapwa6qaifllw2h"
```

To specify all keys that belong to a specific account, use the wildcard (*):

```
"Resource": "arn:aws:payment-cryptography:us-east-2:111122223333:key/*"
```

Some AWS Payment Cryptography actions, such as those for creating keys, cannot be performed on a specific resource. In those cases, you must use the wildcard (*).

```
"Resource": "*"
```

To specify multiple resources in a single statement, use a comma as shown below:

```
"Resource": [
```

```
        "resource1",
        "resource2"
```

Examples

To view examples of AWS Payment Cryptography identity-based policies, see AWS Payment Cryptography identity-based policy examples.

Authorization based on AWS Payment Cryptography tags

You can attach tags to AWS Payment Cryptography resources or pass tags in a request to AWS Payment Cryptography. To control access based on tags, you provide tag information in the condition element of a policy using the `payment-cryptography:ResourceTag/key-name`, `aws:RequestTag/key-name`, or `aws:TagKeys` condition keys.

AWS Payment Cryptography identity-based policy examples

By default, IAM users and roles don't have permission to create or modify AWS Payment Cryptography resources. They also can't perform tasks using the AWS Management Console, AWS CLI, or AWS API. An IAM administrator must create IAM policies that grant users and roles permission to perform specific API operations on the specified resources they need. The administrator must then attach those policies to the IAM users or groups that require those permissions.

To learn how to create an IAM identity-based policy using these example JSON policy documents, see Creating Policies on the JSON Tab in the *IAM User Guide*.

Topics

- Policy best practices
- Using the AWS Payment Cryptography console
- Allow users to view their own permissions
- Ability to access all aspects of AWS Payment Cryptography
- Ability to call APIs using specified keys
- Ability to specifically deny a resource

Policy best practices

Identity-based policies determine whether someone can create, access, or delete AWS Payment Cryptography resources in your account. These actions can incur costs for your AWS account. When you create or edit identity-based policies, follow these guidelines and recommendations:

- **Get started with AWS managed policies and move toward least-privilege permissions** – To get started granting permissions to your users and workloads, use the *AWS managed policies* that grant permissions for many common use cases. They are available in your AWS account. We recommend that you reduce permissions further by defining AWS customer managed policies that are specific to your use cases. For more information, see AWS managed policies or AWS managed policies for job functions in the *IAM User Guide*.
- **Apply least-privilege permissions** – When you set permissions with IAM policies, grant only the permissions required to perform a task. You do this by defining the actions that can be taken on specific resources under specific conditions, also known as *least-privilege permissions*. For more information about using IAM to apply permissions, see Policies and permissions in IAM in the *IAM User Guide*.
- **Use conditions in IAM policies to further restrict access** – You can add a condition to your policies to limit access to actions and resources. For example, you can write a policy condition to specify that all requests must be sent using SSL. You can also use conditions to grant access to service actions if they are used through a specific AWS service, such as CloudFormation. For more information, see IAM JSON policy elements: Condition in the *IAM User Guide*.
- **Use IAM Access Analyzer to validate your IAM policies to ensure secure and functional permissions** – IAM Access Analyzer validates new and existing policies so that the policies adhere to the IAM policy language (JSON) and IAM best practices. IAM Access Analyzer provides more than 100 policy checks and actionable recommendations to help you author secure and functional policies. For more information, see Validate policies with IAM Access Analyzer in the *IAM User Guide*.
- **Require multi-factor authentication (MFA)** – If you have a scenario that requires IAM users or a root user in your AWS account, turn on MFA for additional security. To require MFA when API operations are called, add MFA conditions to your policies. For more information, see Secure API access with MFA in the *IAM User Guide*.

For more information about best practices in IAM, see Security best practices in IAM in the *IAM User Guide*.

Using the AWS Payment Cryptography console

To access the AWS Payment Cryptography console, you must have a minimum set of permissions. These permissions must allow you to list and view details about the AWS Payment Cryptography resources in your AWS account. If you create an identity-based policy that is more restrictive than the minimum required permissions, the console won't function as intended for entities (IAM users or roles) with that policy.

To ensure that those entities can still use the AWS Payment Cryptography console, also attach the following AWS managed policy to the entities. For more information, see Adding Permissions to a User in the *IAM User Guide*.

You don't need to allow minimum console permissions for users that are making calls only to the AWS CLI or the AWS API. Instead, allow access to only the actions that match the API operation that you're trying to perform.

Allow users to view their own permissions

This example shows how you might create a policy that allows IAM users to view the inline and managed policies that are attached to their user identity. This policy includes permissions to complete this action on the console or programmatically using the AWS CLI or AWS API.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "ViewOwnUserInfo",
            "Effect": "Allow",
            "Action": [
                "iam:GetUserPolicy",
                "iam:ListGroupsForUser",
                "iam:ListAttachedUserPolicies",
                "iam:ListUserPolicies",
                "iam:GetUser"
            ],
            "Resource": ["arn:aws:iam::*:user/${aws:username}"]
        },
        {
            "Sid": "NavigateInConsole",
            "Effect": "Allow",
            "Action": [
                "iam:GetGroupPolicy",
```

```
                "iam:GetPolicyVersion",
                "iam:GetPolicy",
                "iam:ListAttachedGroupPolicies",
                "iam:ListGroupPolicies",
                "iam:ListPolicyVersions",
                "iam:ListPolicies",
                "iam:ListUsers"
            ],
            "Resource": "*"
        }
    ]
}
```

Ability to access all aspects of AWS Payment Cryptography

Warning

This example provides wide permissions and is not recommended. Consider least priviledged access models instead.

In this example, you want to grant an IAM user in your AWS account access to all of your AWS Payment Cryptography keys and the ability to call all AWS Payment Cryptography apis including both `ControlPlane` and `DataPlane` operations.

JSON

```
{
      "Version":"2012-10-17",
      "Statement": [
            {
               "Effect": "Allow",
               "Action": [
                  "payment-cryptography:*"
               ],
               "Resource": [
                  "*"
               ]
            }
      ]
```

```
}
```

Ability to call APIs using specified keys

In this example, you want to grant an IAM user in your AWS account access to one of your AWS Payment Cryptography key, `arn:aws:payment-cryptography:us-east-2:111122223333:key/kwapwa6qaifllw2h` and then use this resource in two APIs, `GenerateCardValidationData` and `VerifyCardValidationData`. Conversely, the IAM user will not have access to use this key on other operations such as `DeleteKey` or `ExportKey`

Resources can be either keys, prefixed with `key` or aliases, prefixed with `alias`.

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
            {
                "Effect": "Allow",
                "Action": [
                    "payment-cryptography:VerifyCardValidationData",
                    "payment-cryptography:GenerateCardValidationData"
                ],
                "Resource": [
                    "arn:aws:payment-cryptography:us-east-2:111122223333:key/
kwapwa6qaifllw2h"
                ]
            }
    ]
}
```

Ability to specifically deny a resource

> **Warning**
>
> Carefully consider the implications of granting wildcard access. Consider a least privilege model instead.

In this example, you want to permit an IAM user in your AWS account access to any of your AWS Payment Cryptography key but want to deny permissions to one specific key. The user will have access to `VerifyCardData` and `GenerateCardData` with all keys with the exception of the one specified in the deny statement.

JSON

```
{
      "Version":"2012-10-17",
      "Statement": [
          {
              "Effect": "Allow",
              "Action": [
                  "payment-cryptography:VerifyCardValidationData",
                  "payment-cryptography:GenerateCardValidationData"
              ],
              "Resource": [
                "arn:aws:payment-cryptography:us-east-2:111122223333:key/*"
              ]
          },
          {
              "Effect": "Deny",
              "Action": [
                "payment-cryptography:GenerateCardValidationData"
              ],
              "Resource": [
                "arn:aws:payment-cryptography:us-
east-2:111122223333:key/kwapwa6qaifllw2h"
              ]
          }
      ]
  }
```

Troubleshooting AWS Payment Cryptography identity and access

Topics will be added to this section as IAM-related issues that are specific to AWS Payment Cryptography are identified. For general troubleshooting content on IAM topics, refer to the troubleshooting section of the *IAM User Guide*.

Monitoring AWS Payment Cryptography

Monitoring is an important part of maintaining the reliability, availability, and performance of AWS Payment Cryptography and your other AWS solutions. AWS provides the following monitoring tools to watch AWS Payment Cryptography, report when something is wrong, and take automatic actions when appropriate:

- *Amazon CloudWatch* monitors your AWS resources and the applications you run on AWS in real time. You can collect and track metrics, create customized dashboards, and set alarms that notify you or take actions when a specified metric reaches a threshold that you specify. For example, you can have CloudWatch track usage of certain APIs or notify you if you are approaching your AWS Payment Cryptography quotas. For more information, see the Amazon CloudWatch User Guide.
- *Amazon CloudWatch Logs* enables you to monitor, store, and access your log files from Amazon EC2 instances, CloudTrail, and other sources. CloudWatch Logs can monitor information in the log files and notify you when certain thresholds are met. You can also archive your log data in highly durable storage. For more information, see the Amazon CloudWatch Logs User Guide.
- *AWS CloudTrail* captures API calls and related events made by or on behalf of your AWS account and delivers the log files to an Amazon S3 bucket that you specify. You can identify which users and accounts called AWS, the endpoint called, the resources(keys) used, the source IP address from which the calls were made, and when the calls occurred. For more information, see the AWS CloudTrail User Guide.

Topics

- Logging AWS Payment Cryptography API calls using AWS CloudTrail

Logging AWS Payment Cryptography API calls using AWS CloudTrail

AWS Payment Cryptography is integrated with AWS CloudTrail, a service that provides a record of actions taken by a user, role, or an AWS service in AWS Payment Cryptography. CloudTrail captures all API calls for AWS Payment Cryptography as events. The calls captured include calls from the console and code calls to the API operations. If you create a trail, you can enable continuous delivery of CloudTrail events to an Amazon S3 bucket, including events for AWS Payment

Cryptography. If you don't configure a trail, you can still view the most recent management (Control Plane) events in the CloudTrail console in **Event history**. Using the information collected by CloudTrail, you can determine the request that was made to AWS Payment Cryptography, the IP address from which the request was made, who made the request, when it was made, and additional details.

To learn more about CloudTrail, see the AWS CloudTrail User Guide.

Topics

- AWS Payment Cryptography information in CloudTrail
- Control plane events in CloudTrail
- Data events in CloudTrail
- Understanding AWS Payment Cryptography Control Plane log file entries
- Understanding AWS Payment Cryptography Data plane log file entries

AWS Payment Cryptography information in CloudTrail

CloudTrail is enabled on your AWS account when you create the account. When activity occurs in AWS Payment Cryptography, that activity is recorded in a CloudTrail event along with other AWS service events in **Event history**. You can view, search, and download recent events in your AWS account. For more information, see Viewing Events with CloudTrail Event History.

For an ongoing record of events in your AWS account, including events for AWS Payment Cryptography, create a trail. A *trail* enables CloudTrail to deliver log files to an Amazon S3 bucket. By default, when you create a trail in the console, the trail applies to all AWS Regions. The trail logs events from all Regions in the AWS partition and delivers the log files to the Amazon S3 bucket that you specify. Additionally, you can configure other AWS services to further analyze and act upon the event data collected in CloudTrail logs. For more information, see the following:

- Overview for creating a trail
- CloudTrail supported services and integrations
- Configuring Amazon SNS notifications for CloudTrail
- Receiving CloudTrail log files from multiple Regions
- Receiving CloudTrail log files from multiple accounts

Every event or log entry contains information about who generated the request. The identity information helps you determine the following:

- Whether the request was made with root or AWS Identity and Access Management (IAM) user credentials.
- Whether the request was made with temporary security credentials for a role or federated user.
- Whether the request was made by another AWS service.

For more information, see the CloudTrail userIdentity element.

Control plane events in CloudTrail

CloudTrail logs AWS Payment Cryptography operations, such as CreateKey, ImportKey, DeleteKey, ListKeys, TagResource, and all other control plane operations.

Data events in CloudTrail

Data events provide information about the resource operations performed on or in a resource, such as encrypting a payload or translating a pin. Data events are high-volume activities that CloudTrail does not log by default. You can enable data events API action logging for AWS Payment Cryptography data plane events by using CloudTrail APIs or console. For more information, see Logging data events in the *AWS CloudTrail User Guide*.

With CloudTrail, you must use advanced event selectors to decide which AWS Payment Cryptography API activities are logged and recorded. To log AWS Payment Cryptography data plane events, you must include the resource type `AWS Payment Cryptography key` and `AWS Payment Cryptography alias`. Once this is set, you can refine your logging preferences further by selecting specific data events for recording, such as using the `eventName` filter to track `EncryptData` events. For more information, see `AdvancedEventSelector` in the *AWS CloudTrail API Reference*.

Note

To subscribe to AWS Payment Cryptography data events, you must utilize advanced event selectors. We recommend subscribing to key and alias events to ensure that you receive all events.

AWS Payment Cryptography data events:

- DecryptData
- EncryptData
- GenerateCardValidationData
- GenerateMac
- GeneratePinData
- ReEncryptData
- TranslatePinData
- VerifyAuthRequestCryptogram
- VerifyCardValidationData
- VerifyMac
- VerifyPinData

Additional charges apply for data events. For more information, see AWS CloudTrail Pricing.

Understanding AWS Payment Cryptography Control Plane log file entries

A trail is a configuration that enables delivery of events as log files to an Amazon S3 bucket that you specify. CloudTrail log files contain one or more log entries. An event represents a single request from any source and includes information about the requested action, the date and time of the action, request parameters, and so on. CloudTrail log files aren't an ordered stack trace of the public API calls, so they don't appear in any specific order.

The following example shows a CloudTrail log entry that demonstrates the AWS Payment Cryptography `CreateKey` action.

```
      {
    CloudTrailEvent: {
      tlsDetails= {
        TlsDetails: {
          cipherSuite=TLS_AES_128_GCM_SHA256,
          tlsVersion=TLSv1.3,
          clientProvidedHostHeader=controlplane.paymentcryptography.us-
west-2.amazonaws.com
```

```
        }
      },
    requestParameters=CreateKeyInput (
      keyAttributes=KeyAttributes(
        KeyUsage=TR31_B0_BASE_DERIVATION_KEY,
        keyClass=SYMMETRIC_KEY,
        keyAlgorithm=AES_128,
        keyModesOfUse=KeyModesOfUse(
          encrypt=false,
          decrypt=false,
          wrap=false
          unwrap=false,
          generate=false,
          sign=false,
          verify=false,
          deriveKey=true,
          noRestrictions=false)
        ),
      keyCheckValueAlgorithm=null,
      exportable=true,
      enabled=true,
      tags=null),
    eventName=CreateKey,
    userAgent=Coral/Apache-HttpClient5,
    responseElements=CreateKeyOutput(
      key=Key(
        keyArn=arn:aws:payment-cryptography:us-
east-2:111122223333:key/5rplquuwozodpwsp,
          keyAttributes=KeyAttributes(
            KeyUsage=TR31_B0_BASE_DERIVATION_KEY,
            keyClass=SYMMETRIC_KEY,
            keyAlgorithm=AES_128,
            keyModesOfUse=KeyModesOfUse(
              encrypt=false,
              decrypt=false,
              wrap=false,
              unwrap=false,
              generate=false,
              sign=false,
              verify=false,
              deriveKey=true,
              noRestrictions=false)
            ),
          keyCheckValue=FE23D3,
```

```
          keyCheckValueAlgorithm=ANSI_X9_24,
          enabled=true,
          exportable=true,
          keyState=CREATE_COMPLETE,
          keyOrigin=AWS_PAYMENT_CRYPTOGRAPHY,
          createTimestamp=Sun May 21 18:58:32 UTC 2023,
          usageStartTimestamp=Sun May 21 18:58:32 UTC 2023,
          usageStopTimestamp=null,
          deletePendingTimestamp=null,
          deleteTimestamp=null)
        ),
      sourceIPAddress=192.158.1.38,
        userIdentity={
          UserIdentity: {
            arn=arn:aws:sts::111122223333:assumed-role/TestAssumeRole-us-west-2/
ControlPlane-IntegTest-68211a2a-3e9d-42b7-86ac-c682520e0410,
            invokedBy=null,
            accessKeyId=TESTXECZ5U2ZULLHHMJG,
            type=AssumedRole,
            sessionContext={
              SessionContext: {
                sessionIssuer={
                  SessionIssuer: {arn=arn:aws:iam::111122223333:role/TestAssumeRole-us-
west-2,
                  type=Role,
                  accountId=111122223333,
                  userName=TestAssumeRole-us-west-2,
                  principalId=TESTXECZ5U9M4LGF2N6Y5}
                },
                attributes={
                  SessionContextAttributes: {
                    creationDate=Sun May 21 18:58:31 UTC 2023,
                    mfaAuthenticated=false
                  }
                },
                webIdFederationData=null
              }
            },
          username=null,
          principalId=TESTXECZ5U9M4LGF2N6Y5:ControlPlane-User,
          accountId=111122223333,
          identityProvider=null
        }
      },
```

```
      eventTime=Sun May 21 18:58:32 UTC 2023,
      managementEvent=true,
      recipientAccountId=111122223333,
      awsRegion=us-west-2,
      requestID=151cdd67-4321-1234-9999-dce10d45c92e,
      eventVersion=1.08, eventType=AwsApiCall,
      readOnly=false,
      eventID=c69e3101-eac2-1b4d-b942-019919ad2faf,
      eventSource=payment-cryptography.amazonaws.com,
      eventCategory=Management,
      additionalEventData={
    }
  }
}
```

The following example shows a CloudTrail log entry that demonstrates the AWS Payment Cryptography enabling Multi-Region key replication.


```
{
    "eventVersion": "1.11",
    "userIdentity": {
        "accountId": "111122223333",
        "invokedBy": "payment-cryptography.amazonaws.com"
    },
    "eventTime": "2025-08-15T17:50:41Z",
    "eventSource": "payment-cryptography.amazonaws.com",
    "eventName": "SynchronizeMultiRegionKey",
    "awsRegion": "us-east-1",
    "sourceIPAddress": "payment-cryptography.amazonaws.com",
    "userAgent": "payment-cryptography.amazonaws.com",
    "requestParameters": null,
    "responseElements": null,
    "eventID": "55c0fcbc-5b2e-4bd2-a976-99305be6e6fc",
    "readOnly": false,
    "eventType": "AwsServiceEvent",
    "managementEvent": true,
    "recipientAccountId": "111122223333",
    "serviceEventDetails": {
        "keyArn": "arn:aws:payment-cryptography:us-east-1:111122223333:key/key-id",
        "replicationRegion": "us-east-2"
    },
    "eventCategory": "Management"
```

```
}
```

Understanding AWS Payment Cryptography Data plane log file entries

Data plane events can optionally be configured and function similarly to control plane logs but are typically much higher volumes. Given the sensitive nature of some inputs and outputs to AWS Payment Cryptography data plane operations, you may find certain fields with the message "*** Sensitive Data Redacted ***". This is not configurable and is intended to prevent sensitive data from appearing in logs or trails.

The following example shows a CloudTrail log entry that demonstrates the AWS Payment Cryptography `EncryptData` action.

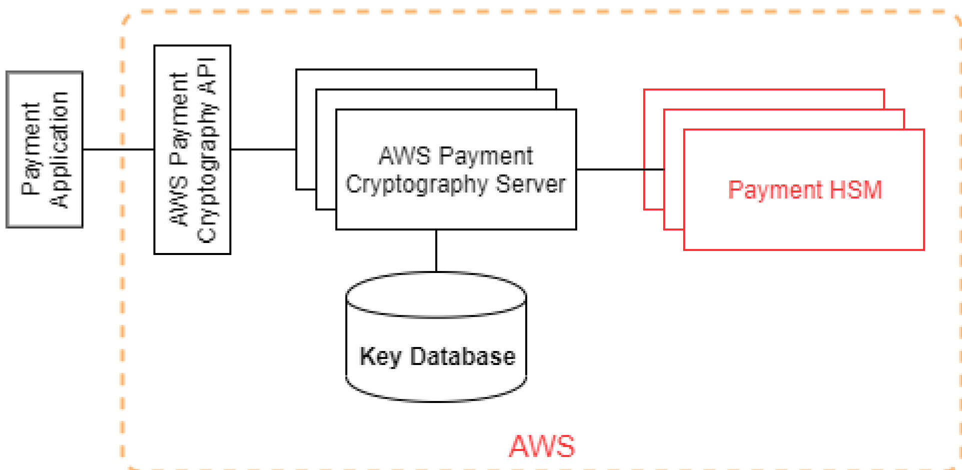

```
      {
        "Records": [
            {
                "eventVersion": "1.09",
                "userIdentity": {
                    "type": "AssumedRole",
                    "principalId": "TESTXECZ5U2ZULLHHMJG:DataPlane-User",
                    "arn": "arn:aws:sts::111122223333:assumed-role/Admin/DataPlane-
User",
                    "accountId": "111122223333",
                    "accessKeyId": "TESTXECZ5U2ZULLHHMJG",
                    "userName": "",
                    "sessionContext": {
                        "sessionIssuer": {
                            "type": "Role",
                            "principalId": "TESTXECZ5U9M4LGF2N6Y5",
                            "arn": "arn:aws:iam::111122223333:role/Admin",
                            "accountId": "111122223333",
                            "userName": "Admin"
                        },
                        "attributes": {
                            "creationDate": "2024-07-09T14:23:05Z",
                            "mfaAuthenticated": "false"
                        }
                    }
                },
                "eventTime": "2024-07-09T14:24:02Z",
                "eventSource": "payment-cryptography.amazonaws.com",
```

```
                "eventName": "GenerateCardValidationData",
                "awsRegion": "us-east-2",
                "sourceIPAddress": "192.158.1.38",
                "userAgent": "aws-cli/2.17.6 md/awscrt#0.20.11 ua/2.0 os/macos#23.4.0
 md/arch#x86_64 lang/python#3.11.8 md/pyimpl#CPython cfg/retry-mode#standard md/
installer#exe md/prompt#off md/command#payment-cryptography-data.generate-card-
validation-data",
                "requestParameters": {
                    "key_identifier": "arn:aws:payment-cryptography:us-
east-2:111122223333:key/5rplquuwozodpwsp",
                    "primary_account_number": "*** Sensitive Data Redacted ***",
                    "generation_attributes": {
                        "CardVerificationValue2": {
                            "card_expiry_date": "*** Sensitive Data Redacted ***"
                        }
                    }
                },
                "responseElements": null,
                "requestID": "f2a99da8-91e2-47a9-b9d2-1706e733991e",
                "eventID": "e4eb3785-ac6a-4589-97a1-babdd3d4dd95",
                "readOnly": true,
                "resources": [
                    {
                        "accountId": "111122223333",
                        "type": "AWS::PaymentCryptography::Key",
                        "ARN": "arn:aws:payment-cryptography:us-
east-2:111122223333:key/5rplquuwozodpwsp"
                    }
                ],
                "eventType": "AwsApiCall",
                "managementEvent": false,
                "recipientAccountId": "111122223333",
                "eventCategory": "Data",
                "tlsDetails": {
                    "tlsVersion": "TLSv1.3",
                    "cipherSuite": "TLS_AES_128_GCM_SHA256",
                    "clientProvidedHostHeader": "dataplane.payment-cryptography.us-
east-2.amazonaws.com"
                }
            }
        ]
    }
```

Cryptographic details

AWS Payment Cryptography provides a web interface to generate and manage cryptographic keys for payment transactions. AWS Payment Cryptography offers standard key management services and payment transaction cryptography and tools you can use for centralized management and auditing. This documentation provides a detailed description of the cryptographic operations you can use in AWS Payment Cryptography to assist you in evaluating the features offered by the service.

AWS Payment Cryptography contains multiple interfaces (including a RESTful API, through the AWS CLI, AWS SDK and the AWS Management Console) to request cryptographic operations of a distributed fleet of PCI PTS HSM-validated hardware security modules.



AWS Payment Cryptography is a tiered service consisting of web-facing AWS Payment Cryptography hosts and a tier of HSMs. The grouping of these tiered hosts forms the AWS Payment Cryptography stack. All requests to AWS Payment Cryptography must be made over the Transport Layer Security protocol (TLS) and terminate on an AWS Payment Cryptography host. The service hosts only allow TLS with a cipher suite that provides perfect forward secrecy. The service authenticates and authorizes your requests using the same credential and policy mechanisms of IAM that are available for all other AWS API operations.

AWS Payment Cryptography servers connect to the underlying HSM via a private, non-virtual network. Connections between service components and HSM are secured with mutual TLS (mTLS) for authentication and encryption.

Topics

- Design goals
- Foundations
- Internal operations
- Customer operations

Design goals

AWS Payment Cryptography is designed to meet the following requirements:

- **Trustworthy** — Use of keys is protected by access control policies that you define and manage. There is no mechanism to export plaintext AWS Payment Cryptography keys. The confidentiality of your cryptographic keys is crucial. Multiple Amazon employees with role-specific access to quorum-based access controls are required to perform administrative actions on the HSMs. No Amazon employees have access to HSM main (or master) keys or backups. Main keys cannot be synchronized with HSMs that are not part of an AWS Payment Cryptography region. All other keys are protected by HSM main keys. Therefore, customer AWS Payment Cryptography keys are not usable outside of the AWS Payment Cryptography service operating within a customer's account.
- **Low-latency and high throughput** — AWS Payment Cryptography provides cryptographic operations at latency and throughput level suitable for managing payment cryptographic keys and processing payment transactions.
- **Durability** — The durability of cryptographic keys is designed to be equal that of the highest durability services in AWS. A single cryptographic key can be shared with a payment terminal, EMV chip card, or other secure cryptographic device (SCD) that is in use for many years.
- **Independent Regions** — AWS provides independent regions for customers who need to restrict data access in different regions or need to comply with data residency requirements. Key usage can be isolated within an AWS Region.
- **Secure source of random numbers** — Because strong cryptography depends on truly unpredictable random number generation, AWS Payment Cryptography provides a high-quality and validated source of random numbers. All key generation for AWS Payment Cryptography uses PCI PTS HSM-listed HSM, operating in PCI mode.
- **Audit** — AWS Payment Cryptography records the use and management of cryptographic keys in CloudTrail logs and service logs available via Amazon CloudWatch. You can use CloudTrail logs to inspect use of your cryptographic keys, including the use of keys by accounts that you have shared keys with. AWS Payment Cryptography is audited by third party assessors against

applicable PCI, card brand, and regional payment security standards. Attestations and Shared Responsibility guides are available on AWS Artifact.

- **Elastic** — AWS Payment Cryptography scales out and in according to your demand. Instead of predicting and reserving HSM capacity, AWS Payment Cryptography provides payment cryptography on-demand. AWS Payment Cryptography takes responsibility for maintaining the security and compliance of HSM to provide sufficient capacity to meet customer's peak demand.

Foundations

The topics in this chapter describe the cryptographic primitives of AWS Payment Cryptography and where they are used. They also introduce the basic elements of the service.

Topics

- Cryptographic primitives
- Entropy and random number generation
- Symmetric key operations
- Asymmetric key operations
- Key storage
- Key import using symmetric keys
- Key import using asymmetric keys
- Key export
- Derived Unique Key Per Transaction (DUKPT) protocol
- Key hierarchy

Cryptographic primitives

AWS Payment Cryptography uses parameter-able, standard cryptographic algorithms so that applications can implement the algorithms needed for their use case. The set of cryptographic algorithms is defined by PCI, ANSI X9, EMVco, and ISO standards. All cryptography is performed by PCI PTS HSM standard-listed HSMs running in PCI mode.

Entropy and random number generation

AWS Payment Cryptography key generation is performed on the AWS Payment Cryptography HSMs. The HSMs implement a random number generator that meets the PCI PTS HSM requirement for all supported key types and parameters.

Symmetric key operations

Symmetric key algorithms and key strengths defined in ANSI X9 TR 31, ANSI X9.24, and PCI PIN Annex C are supported:

- **Hash functions** — Algorithms from the SHA2 and SHA3 family with output size greater than 2551. Except for backwards compatibility with pre-PCI PTS POI v3 terminals.
- **Encryption and decryption** — AES with key size greater than or equal to 128 bits, or TDEA with keys size greater than or equal to 112 bits (2 key or 3 key).
- **Message Authentication Codes (MACs)** CMAC or GMAC with AES, as well as HMAC with an approved hash function and a key size greater than or equal to 128.

AWS Payment Cryptography uses AES 256 for HSM main keys, data protection keys, and TLS session keys.

Note: Some of the listed functions are used internally to support standard protocols and data structures. See the API documentation for algorithms supported by specific actions.

Asymmetric key operations

Asymmetric key algorithms and key strengths defined in ANSI X9 TR 31, ANSI X9.24, and PCI PIN Annex C are supported:

- **Approved key establishment schemes** — as described in NIST SP800-56A (ECC/FCC2-based key agreement), NIST SP800-56B (IFC-based key agreement), and NIST SP800-38F (AES-based key encryption/wrapping).

AWS Payment Cryptography hosts only allow connections to the service using TLS with a cipher suite that provides perfect forward secrecy.

Note: Some of the listed functions are used internally to support standard protocols and data structures. See the API documentation for algorithms supported by specific actions.

Key storage

AWS Payment Cryptography keys are protected by HSM AES 256 main keys and stored in ANSI X9 TR 31 key blocks in an encrypted database. The database is replicated to in-memory database on AWS Payment Cryptography servers.

According to PCI PIN Security Normative Annex C, AES 256 keys are equally as strong as or stronger than:

- 3-key TDEA
- RSA 15360 bit
- ECC 512 bit
- DSA, DH, and MQV 15360/512

Key import using symmetric keys

AWS Payment Cryptography supports import of cryptograms and key blocks with symmetric or public keys with a symmetric key encryption key (KEK) that is as strong or stronger than the protected key for import.

Key import using asymmetric keys

AWS Payment Cryptography supports import of cryptograms and key blocks with symmetric or public keys protected by a private key encryption key (KEK) that is as strong or stronger than the protected key for import. The public key provided for decryption must have its authenticity and integrity ensured by a certificate from an authority trusted by the customer.

Public KEK provided by AWS Payment Cryptography have the authentication and integrity protection of a certificate authority (CA) with attested compliance to PCI PIN Security and PCI P2PE Annex A.

Key export

Keys can be exported and protected by keys with the appropriate KeyUsage and that are as strong as or stronger than the key to be exported.

Derived Unique Key Per Transaction (DUKPT) protocol

AWS Payment Cryptography supports with TDEA and AES base derivation keys (BDK) as described by ANSI X9.24-3.

Key hierarchy

The AWS Payment Cryptography key hierarchy ensures that keys are always protected by keys as strong as or stronger than the keys they protect.

Payment Cryptographic Keys

Key Protection Keys
Authentication Keys
Key Encrytion Keys
PIN and Data Protection Keys
Regional Main Key
Profile Main Key (LMK/MFK)
Root of trust for service public KEK keys
AWS Payment Cryptography Keys
Customer-provider trusted root for public KEK keys
Public Migration KEK
Symmetric Migration KEK
Business Partner KEK
Payment Network ZCMK
Payment Application KEK
DUKPT BDK
Customer Keys
DUKPT Transaction Keys
Card Verification Key
PIN Verification key
Payment Network Working Keys
PIN Creation PEK

AWS Payment Cryptography keys are used for key protection within the service:

Key	Description
Regional Main Key	Protects virtual HSM images, or profiles, used for cryptographic processing. This key exists only in HSM and secure backups.
Profile Main Key	Top level customer key protection key, traditionally called a Local Master Key (LMK) or Master File Key (MFK) for customer keys. This key exists only in HSM and secure backups. Profiles define distinct HSM configurations as required by security standards for payments use cases.
Root of trust for AWS Payment Cryptography public key encryption key (KEK) keys	The trusted root public key and certificate for authenticating and validating public keys supplied by AWS Payment Cryptography for key import and export using asymmetric keys.

Customer keys are grouped by keys used to protect other keys and keys that protect payment-related data. These are examples of customer keys of both types:

Key	Description
Customer-provided trusted root for public KEK keys	Public key and certificate supplied by you as the root of trust for authenticating and validating public keys that you supply for key import and export using asymmetric keys.
Key Encryption Keys (KEK)	KEK are used solely to encrypt other keys for exchange between external key stores and AWS Payment Cryptography, business partners, payment networks, or different applications within your organization.
Derived Unique Key Per Transaction (DUKPT) base derivation key (BDK)	BDKs are used to create unique keys for each payment terminal and translate transactions

Key	Description
	from multiple terminals to a single acquiring bank, or acquirer, working key. The best practice, which is required by PCI Point-to-Point Encryption (P2PE), is that different BDKs are used for different terminal models, key injection or initialization services, or other segmentation to limit the impact of compromising a BDK.
Payment network zone control master key (ZCMK)	ZCMK, also referred to as zone keys or zone master keys, are provided by payment networks to establish initial working keys.
DUKPT transaction keys	Payment terminals configured for DUKPT derive a unique key for the terminal and transaction. The HSM receiving the transacti on can determine the key from the terminal identifier and transaction sequence number.
Card data preparation keys	EMV issuer master keys, EMV card keys and verification values, and card personalization data file protection keys are used to create data for individual cards for use by a card personalization provider. These keys and cryptographic validation data are also used by issuing banks, or issuers, for authenticating card data as part of authorizing transactions.
Card data preparation keys	EMV issuer master keys, EMV card keys and verification values, and card personalization data file protection keys are used to create data for individual cards for use by a card personalization provider. These keys and cryptographic validation data are also used by issuing banks, or issuers, for authenticating card data as part of authorizing transactions.

Key	Description
Payment network working keys	Often referred to as issuer working key or acquirer working key, these are the keys that encrypt transaction sent to or received from payment networks. These keys are rotated frequently by the network, often daily or hourly. These are PIN encryption keys (PEK) for PIN/Debit transactions.
Personal Identification Number (PIN) encryptio n keys (PEK)	Applications that create or decrypt PIN blocks use PEK to prevent storage or transmission of clear text PIN.

Internal operations

This topic describes internal requirements implemented by the service to secure customer keys and cryptographic operations for a globally distributed and scalable payment cryptography and key management service.

Topics

- HSM protection
- General key management
- Management of customer keys
- Communication security
- Logging and monitoring

HSM protection

HSM specifications and lifecycle

AWS Payment Cryptography uses a fleet of commercially available HSMs. The HSMs are FIPS 140-2 Level 3 validated and also use firmware versions and the security policy listed on the PCI Security Standards Council approved PCI PTS Devices list as PCI HSM v3 compliant. The PCI PTS HSM standard includes additional requirements for the manufacturing, shipment, deployment,

management, and destruction of HSM hardware which are important for payment security and compliance but not addressed by FIPS 140.

Third party assessors verify HSM make model, firmware, configuration, lifecycle physical management, change control, operator access controls, main key management, and all PCI PIN and P2PE requirements related to HSMs and HSM operations.

All HSMs are operated in PCI Mode and configured with the PCI PTS HSM security policy. Only functions required to support AWS Payment Cryptography use cases are enabled. AWS Payment Cryptography does not provide for printing, display, or return of clear text PINs.

HSM device physical security

Only HSMs that have device keys signed by an AWS Payment Cryptography certificate authority (CA) by the manufacturer prior to delivery can be used by the service. The AWS Payment Cryptography is a sub-CA of the manufacturer's CA that is the root of trust for HSM manufacturer and device certificates. The manufacturer's CA has attested compliance with PCI PIN Security Annex A and PCI P2PE Annex A. The manufacturer verifies that all HSM with device keys signed by the AWS Payment Cryptography CA are shipped to AWS' designated receiver.

As required by PCI PIN Security, the manufacturer supplies a list of serial numbers via a different communication channel than the HSM shipment. These serial numbers are checked at each step in the process of HSM installation into AWS data centers. Finally, AWS Payment Cryptography operators validate the list of installed HSM against the manufacturer's list before adding the serial number to list of HSM permitted to receive AWS Payment Cryptography keys.

HSMs are in secure storage or under dual control at all times, which includes:

- Shipment from the manufacturer to an AWS rack assembly facility.
- During rack assembly.
- Shipment from the rack assembly facility to a data center.
- Receipt and installation into a data center secure processing room. HSM racks enforce dual control with card access-controlled locks, alarmed door sensors, and cameras.
- During operations.
- During decommissioning and destruction.

A complete chain-of-custody, with individual accountability, is maintained and monitored for each HSM.

HSM initialization

An HSM is only initialized as part of the AWS Payment Cryptography fleet after its identity and integrity are validated by serial numbers, manufacturer installed device keys, and firmware checksum. After the authenticity and integrity of an HSM validated, it is configured, including enabling PCI Mode. Then AWS Payment Cryptography region main keys and profile main keys are established and the HSM is available to the service.

HSM service and repair

HSM have serviceable components that do not require violation of the device's cryptographic boundary. These components include cooling fans, power supplies, and batteries. If an HSM or another device within the HSM rack needs service, dual control is maintained during the entire period that the rack is open.

HSM decommissioning

Decommissioning occurs due to end-of-life or failure of an HSM. HSM are logically zeroized before removal from their rack, if functional, then destroyed within secure processing rooms of AWS data centers. They are never returned to the manufacturer for repair, used for another purpose, or otherwise removed from a secure processing room before destruction.

HSM firmware update

HSM firmware updates are applied when required to maintain alignment with PCI PTS HSM and FIPS 140-2 (or FIPS 140-3) listed versions, if an update is security related, or it is determined that customers can benefit from features in a new version. AWS Payment Cryptography HSMs run off-the-shelf firmware, matching PCI PTS HSM-listed versions. New firmware versions are validated for integrity with the PCI or FIPS certified firmware versions then tested for functionality before rollout to all HSMs.

Operator access

Operators can have non-console access to HSM for troubleshooting in rare cases that information gathered from HSM during normal operations is insufficient to identify a problem or plan a change. The following steps are executed:

- Troubleshooting activities are developed and approved and the non-console session is scheduled.
- An HSM is removed from customer processing service.

- Main keys are deleted, under dual control.
- Operator is permitted non-console access to the HSM to perform approved troubleshooting activities, under dual control.
  - After termination of the non-console session, the initial provisioning process is performed on the HSM, returning the standard firmware and configuration, then synchronizing the main key, before returning the HSM to serving customers.
  - Records of the session are recorded in change tracking.
  - Information obtained from the session is used for planning future changes.

All non-console access records are reviewed for process compliance and potential changes to HSM monitoring, the non-console-access management process, or operator training.

General key management

All HSM in a region are synchronized to a Region Main Key. A Region Main Key protects at least one Profile Main Key. A Profile Main Key protects customer keys.

All main keys are generated by an HSM and distributed to by symmetric key distribution using asymmetric techniques, aligned with ANSI X9 TR 34 and PCI PIN Annex A.

Generation

AES 256 bit Main keys are generated on one of the HSM provisioned for the service HSM fleet, using the PCI PTS HSM random number generator.

Region main key synchronization

HSM region main keys are synchronized by the service across the regional fleet with mechanisms defined by ANSI X9 TR-34, which include:

- Mutual authentication using key distribution host (KDH) and key receiving device (KRD) keys and certificates to provide authentication and integrity of for public keys.
- Certificates are signed by a certificate authority (CA) that meets the requirements of PCI PIN Annex A2, except for asymmetric algorithms and key strengths appropriate for protecting AES 256 bit keys.
- Identification and key protection for the distributed symmetric keys is consistent with ANSI X9 TR-34 and PCI PIN Annex A1, except for asymmetric algorithms and key strengths appropriate for protecting AES 256 bit keys.

Region main keys are established for HSMs that have been authenticated and provisioned for a region by:

- A main key is generated on an HSM in the region. That HSM is designated as the key distribution host.
- All provisioned HSMs in the region generate KRD authentication token, which contain the public key of the HSM and non-replayable authentication information.
- KRD tokens are added to the KDH allow list after the KDH validates the identity and permission of the HSM to receive keys.
- The KDH produces an authenticable main key token for each HSM. Tokens contain KDH authentication information and encrypted main key that is loadable only on an HSM that it has been created for.
- Each HSM is sent the main key token built for it. After validating the HSM's own authentication information and the KDH authentication information, the main key is decrypted by the KRD private key and loaded into the main key.

In the event that a single HSM must be re-synchronized with a region:

- It is re-validated and provisioned with firmware and configuration.
- If it is new to the region:
  - The HSM generates a KRD authentication token.
  - The KDH adds the token to its allow list.
  - The KDH generates a main key token for the HSM.
  - The HSM loads the main key.
  - The HSM is made available to the service.

This assures that:

- Only HSM validated for AWS Payment Cryptography processing within a region can receive that region's master key.
- Only a master key from an AWS Payment Cryptography HSM can be distributed to an HSM in the fleet.

Region main key rotation

Region main keys are rotated at the expiration of the crypto period, in the unlikely event of a suspected key compromise, or after changes to the service that are determined to impact the security of the key.

A new region main key is generated and distributed as with initial provisioning. Saved profile main keys must be translated to the new region main key.

Region main key rotation does not impact customer processing.

Profile main key synchronization

Profile main keys are protected by region main keys. This restricts a profile to a specific region.

Profile main keys are provisioned accordingly:

- A profile main key is generated on an HSM that has the region main key synchronized.
- The profile main key is stored and encrypted with the profile configuration and other context.
- The profile is used for customer cryptographic functions by any HSM in the region with the region main key.

Profile main key rotation

Profile main keys are rotated at the expiration of the crypto period, after suspected key compromise, or after changes to the service that are determined to impact the security of the key.

Rotation steps:

- A new profile main key is generated and distributed as a pending main key as with initial provisioning.
- A background process translates customer key material from the established profile main key to the pending main key.
- When all customer keys have been encrypted with the pending key, the pending key is promoted to the profile main key.
- A background process deletes customer key material protected by the expired key.

Profile main key rotation does not impact customer processing.

Protection

Keys depend only on the key hierarchy for protection. Protection of main keys is critical to prevent loss or compromise all customer keys.

Region main keys are restorable from backup only to HSM authenticated and provisioned for the service. These keys can only be stored as mutually authenticable, encrypted main key tokens from a specific KDH for a specific HSM.

Profile master keys are stored with profile configuration and context information encrypted by region.

Customer keys are stored in key blocks, protected by a profile master key.

All keys exist exclusively within an HSM or stored protected by another key of equal or stronger cryptographic strength.

Durability

Customer keys for transaction cryptography and business functions must be available even in extreme situations that would typically cause outages. AWS Payment Cryptography utilizes a multiple level redundancy model across availability zones and AWS regions. Customer's requiring higher availability and durability for payment cryptographic operations than what is provided by the service should implement multi-region architectures.

HSM authentication and main key tokens are saved and may be used to restore a main key or synchronize with a new main key, in the event that an HSM must be reset. The tokens are archived and used only under dual control when required.

Operator access to HSM main keys

Main keys exist only in HSM managed by the service and secured in secure AWS facilities. Main keys cannot be exported from any HSM or synchronized to an HSM that is not initialized by the manufacturer for use in the service. AWS operators cannot obtain main keys in any form that could be loaded into an HSM not managed by the service.

Management of customer keys

At AWS, customer trust is our top priority. You maintain full control of your keys that you import to or create in the service under your AWS account. You retain responsibility for configuring access to keys.

AWS Payment Cryptography is a service provider that uses HSMs and manages keys on behalf of customers, similar to long-standing payment service providers. The service has complete responsibility for HSM physical and logical security. Key management responsibility is shared between the service and customers because the customer must provide accurate information about keys created by or imported to the service, which the service uses to enforce correct key use and management. AWS data segregation protections are used to ensure that compromise of keys belonging one AWS account cannot compromise keys belonging to another.

AWS Payment Cryptography has full responsibility for the HSM physical compliance and key management for keys managed by the service. This requires ownership and management of HSM main keys and protection of customer keys managed by the AWS Payment Cryptography.

Customer key space separation

AWS Payment Cryptography enforces key policies for all key use, including limiting principals to the account owning the key, unless a key is explicitly shared with another account.

AWS accounts provide complete environment segregation between customers or applications analogous to non-cloud implementations in different data centers. Each account provides isolated access control, networking, compute resources, data storage, cryptographic keys for data protection and payment transactions, and all AWS resources. AWS services like Organizations and Control Tower enable enterprise management of separate application accounts, analogous to cages or rooms within an enterprise data center.

Operator access to customer keys

Customer keys managed by the service are stored protected by partition main keys and can only be used by the owning customer account or account the owner has specifically configured for key sharing. AWS operators cannot export or perform key management or cryptographic operations with customer keys using manual access to the service, which is managed by AWS manual operator access mechanisms.

Service code that implements customer key management and use is subject to AWS secure code practices as assessed per the AWS PCI DSS assessment.

Backup and recovery

Keys and key information stored internally by the service for a region is backed up to encrypted archives by AWS. Archives require dual control by AWS to restore.

Key blocks

All keys are stored and processed in ANSI X9.143 format key blocks.

Keys may be imported into the service from cryptograms or other key block formats supported by ImportKey. Similarly, keys may be exported, if they are exportable, to other key block formats or cryptograms supported by key export profiles.

Key use

Key use is restricted to the configured KeyUsage by the service. The service will fail any requests with inappropriate key usage, mode of use, or algorithm for the requested cryptographic operation.

Key exchange relationships

PCI PIN Security and PCI P2PE require that organizations that share keys that encrypt PINs or carddata, including the key exchange keys (KEK) used to share those keys, not share the same keys with any other organizations. It is a best practice that symmetric keys are shared between only 2 parties for a single purpose, including within the same organization. This minimizes the impact of suspected key compromises that force replacing impacted keys.

Even business cases that require sharing keys between more than 2 parties, should keep the number of parties to the minimum number.

AWS Payment Cryptography provides key tags that can be used to track and enforce key usage within those requirements.

For example, KEK and BDK for different key injection facilities can by identified by setting a “KIF”=“POSStation” for all keys shared with that service provider. Another example would be to tag keys shared with payment networks with “Network”=“PayCard”. Tagging enables you to create access controls and create audit reports to enforce and demonstrate your key management practices.

Key deletion

DeleteKey marks keys in the database for deletion after a customer-configurable period. After this period the key is irretrievably deleted. This is a safety mechanism to prevent the accidental or malicious deletion of a key. Keys marked for deletion are not available for any actions except RestoreKey.

Deleted keys remain in service backups for 7 days after deletion. They are not restorable during this period.

Keys belonging to closed AWS accounts are marked for deletion. If the account is reactivated before the deletion period is reached any keys marked for deletion are restored, but disabled. They must be re-enabled by you in order to use them for cryptographic operations.

Communication security

External

AWS Payment Cryptography API endpoints meet AWS security standards including TLS at or above 1.2 and Signature Version 4 for authentication and integrity of requests.

Incoming TLS connections are terminated on network load balancers and forwarded to API handlers over internal TLS connections.

Internal

Internal communications between service components and between service components and other AWS service are protected by TLS using strong cryptography.

HSM are on a private, non-virtual network that is only reachable from service components. All connections between HSM and service components are secured with mutual TLS (mTLS), at or above TLS 1.2. Internal certificates for TLS and mTLS are managed by Amazon Certificate Manager using an AWS Private Certificate Authority. Internal VPCs and the HSM network are monitored for unexpected activities and configuration changes.

Logging and monitoring

Internal service logs include:

- CloudTrail logs of AWS service calls made by the service
- CloudWatch logs of both events directly logged to CloudWatch logs or events from HSM
- Log files from HSM and service systems
- Log archives

All log sources monitor and filter for sensitive information, including about keys. Logs are systematically reviewed to ensure that they contain do not contain sensitive customer information.

Access to logs is restricted to individuals needed for completing job roles.

All logs are retained in alignment with AWS log retention policies.

Customer operations

AWS Payment Cryptography has full responsibility for the HSM physical compliance under PCI standards. The service also provides a secure key store and ensures that keys can only be used for the purposes permitted by PCI standards and specified by you during creation or import. You are responsible for configuring key attributes and access to leverage the security and compliance capabilities of the service.

Topics

- Generating keys
- Importing keys
- Exporting keys
- Deleting keys
- Rotating keys

Generating keys

When creating keys, you set the attributes that the service uses to enforce compliant use of the key:

- Algorithm and key length
- Usage
- Availability and expiration

Tags that are used for attribute-based access control (ABAC) are used to limit keys for use with specific partners or applications should also be set during creation. Be sure to include policies to limit roles permitted to delete or change tags.

You should ensure that the policies that determine the roles that may use and manage the key are set prior to the creation of the key.

Note

IAM policies on the CreateKey commands may be used to enforce and demonstrate dual control for key generation.

Importing keys

When importing keys, the attributes to enforce compliant use of the key are set by the service using the cryptographically bound information in the key block. The mechanism for setting fundamental key context is to use key blocks created with the source HSM and protected by a shared or asymmetric KEK. This aligns with PCI PIN requirements and preserves usage, algorithm, and key strength from the source application.

Important key attributes, tags, and access control policies must be established on import in addition to the information in the key block.

Importing keys using cryptograms does not transfer key attributes from the source application. You must set the attributes appropriately by using this mechanism.

Often keys are exchanged using clear text components, transmitted by key custodians, then loaded with ceremony implementing dual control in a secure room. This is not directly supported by AWS Payment Cryptography. The API will export a public key with a certificate that can be imported by your own HSM to export a key block that is importable by the service. The enables use of your own HSM for loading clear text components.

You should use Key check values (KCV) to verify that imported keys match source keys.

IAM policies on the ImportKey API may be used to enforce and demonstrate dual control for key import.

Exporting keys

Sharing keys with partners or on-premises applications may require exporting keys. Using key blocks for exports maintains fundamental key context with the encrypted key material.

Key tags can be used to limit the export of keys to KEK that share the same tag and value.

AWS Payment Cryptography does not provide or display clear text key components. This requires direct access by key custodians to PCI PTS HSM or ISO 13491 tested secure cryptographic devices

(SCD) for display or printing. You can establish an asymmetric KEK or a symmetric KEK with your SCD to conduct the clear text key component creation ceremony under dual control.

Key check values (KCV) should be used to verify that imported by the destination HSM match source keys.

Deleting keys

You can use the delete key API to schedule keys for deletion after a period of time that you configure. Before that time keys are recoverable. Once keys are deleted they are permanently removed from the service.

IAM policies on the DeleteKey API may be used to enforce and demonstrate dual control for key deletion.

Rotating keys

The effect of key rotation can be implemented using key alias by creating or importing a new key, then modifying the key alias to refer to the new key. The old key would be deleted or disabled, depending on your management practices.

Quotas for AWS Payment Cryptography

Your AWS account has default quotas, formerly referred to as limits, for each AWS service. Unless otherwise noted, each quota is region-specific. You can request increases for some quotas, and other quotas cannot be increased.

Name	Default	Adjustable	Description
Aliases	Each supported Region: 2,000	Yes	The maximum number of aliases you can have in this account in the current Region.
Combined rate of control plane requests	Each supported Region: 5 per second	Yes	The maximum number of control plane requests per second that you can make in this account in the current Region. This quota applies to all control plane operations combined.
Combined rate of data plane requests (asymmetric)	Each supported Region: 20 per second	Yes	The maximum number of requests per second for data plane operations with an asymmetric key that you can make in this account in the current Region. This quota applies to all data plane operations combined.
Combined rate of data plane requests (symmetric)	Each supported Region: 500 per second	Yes	The maximum number of requests per second for data plane operation

Name	Default	Adjustable	Description
			s with a symmetric key that you can make in this account in the current Region. This quota applies to all data plane operations combined.
Keys	Each supported Region: 2,000	Yes	The maximum number of keys you can have in this account in the current Region, excluding deleted keys.

Document history for the AWS Payment Cryptography User Guide

The following table describes the documentation releases for AWS Payment Cryptography.

Change	Description	Date
New Feature - AS2805	Support for algorithms and flows to support AS2805 regional support	December 17, 2025
New Feature - Multi-Region key replication	With Multi-Region key replication, you can replicate your AWS Payment Cryptography keys to multiple AWS Regions.	September 10, 2025
New Feature - ECDH	With this release, ECDH can be used to establish a shared KEK for further key exchange.	March 30, 2025
New Key Exchange Guidance	New guidance provided for key exchanges. Information on common JCB commands also added.	January 31, 2025
New region launch	Added endpoints for new region launch in Europe (Frankfurt), Europe (Ireland), Asia Pacific (Singapore) and Asia Pacific (Tokyo)	July 31, 2024
CloudTrail for Data Plane and Dynamic Keys	Added information about utilizing CloudTrail for data plane (cryptographic) operations including examples. Also added	July 10, 2024

	information about utilizing Dynamic Keys for certain functions to better support one-time or limited use keys that should not be imported into AWS Payment Cryptography	
Updated Examples	Added new examples for card issuing	July 1, 2024
Feature release	Adding information on VPC endpoints(PrivateLink) and iCVV examples.	May 30, 2024
Feature release	Information added on new features around key import/export using RSA and exporting DUKPT IPEK/IK keys.	January 15, 2024
Initial release	Initial release of the AWS Payment Cryptography User Guide	June 8, 2023