Voltage SecureData Payments LPOS SDK Developer Guide

Contents

1

The Voltage SecureData Payments LPOS SDK Solution

The Voltage SecureData Payments Lightweight Point Of Sale (LPOS) SDK is a small footprint toolkit that implements POS side encryption utilizing Voltage Structure Preserving Encryption procedures (SPE) in conjunction with Derived Unique Key Per Transaction (DUKPT) key management in a constrained environment. It functions in the operating system or firmware layer within a POS device.

In conjunction with the Voltage SecureData Payments Host SDK, you can create end-to-end protection of credit card transactions within the merchant and processor or acquirer networks without the exchange of keys. The LPOS SDK encrypts data for use by the Host processor only.

note Voltage does not perform any certification testing on POS devices manufactured by its partners, nor do we have any access to payment processor environments.

Overview and Architecture

The LPOS SDK consists of C-language source code that you port and compile for a target environment. Its main purpose is to use Voltage Structure Preserving Encryption (SPE) in conjunction with the DUKPT key management protocol to encrypt payment data on a POS device.

A symmetric Format Preserving Encryption (FPE) session key is derived from the Initial PIN Encryption Key (IPEK) and a Key Serial Number (KSN). The KSN is formed from a unique user defined device identifier, and an internal transaction counter. It is 80 bits long, with the first 59 bits being used for identifying information and 21 bits being used for a transaction counter. The identifying information. The decrypting entity uses the information in the KSN to select the correct Base Derivation Key (BDK) within a DUKPT district on the decrypting side of the process. Each time a transaction is completed, the POS device increments the transaction count, creating a new encryption key. No two encryption keys are alike, and cannot be derived from previous or future keys.

The IPEK is derived from the BDK. The BDK is created and associated with a DUKPT district's master secret. You create the district and associate the BDK using the Management Console. For More information in using the Management Console to create a DUKPT district and a BDK, see the *Voltage*

SecureData Administrator Guide. The master secret should be stored in a Hardware Security Module (HSM), or in a Tamper-Resistant Security Module (TRSM). For information on HSMs, see the *Voltage SecureData HSM Supplement*.

The processor injects the IPEK and KSN into the POS device during manufacture time. The IPEK represents the first encryption key. However, it is only used to derive future keys, and is not used to encrypt any data.

Once the POS device is initialized with the IPEK and KSN, the process of encrypting data is as follows:

1. The transaction counter is incremented. The previously used encryption key (or the IPEK in the case of initial encryption) is discarded.
2. The appropriate encryption key, based on the transaction counter within the KSN, encrypts the data.
3. The encrypted data along with the KSN is sent to the Host for further processing. An encryption key is never sent.
4. The POS gets approval from the Host system for the transaction.
5. The POS is now ready for the next transaction.

When encrypted information and the KSN are sent to the Host for processing, the identifying information in the KSN tells the Host which district and BDK to use, and the transaction counter tells it which key to derive from the BDK. The derived key is decrypts the data, and sends it for further processing.

You may have multiple BDKs per district, each BDK having its own identifying value. The indentifying information in the KSN points to the proper district and BDK. This allows for multiple systems to use a single district with the KSN's information acting as a BDK selector.

The LPOS SDK is completely DUKPT standards compliant regarding key management. The Voltage DUKPT implementation uses FPE/AES to encrypt data, and TDES to encrypt keys.

The LPOS supports data encryption modularity. If some portion of encryption is implemented in hardware, then the LPOS software-based versions can be replaced. See Chapter 4, "LPOS Modularity" for more information.

DUKPT

DUKPT is a key management scheme in which a unique key is derived from a fixed key and used for each separate transaction. If a derived key is compromised, future and past keys and transaction data are still protected since the next or prior keys cannot be easily determined. The features of the DUKPT scheme are:

- Originating and receiving entities can agree on a transaction key based on an IPEK, advanced once per transaction. The LPOS SDK and the Host SDK derives their respective keys based on a transaction count, or serial number, and the original IPEK key at the processor.
- Each transaction has a distinct key from all other transactions.

- Any compromised key cannot compromise the previous or following key.
- Every device generates a separate key sequence.
- No key agreement protocol is required between originators and receivers of encrypted messages. Each entity can create its own key with specific key information passing between the entities. Encrypting and decrypting entities can create their own matching encrypting and decrypting keys.

The Voltage LPOS Solution

The LPOS extends previous Payments software, its POS/Host protocol, and IB-KEEP. This includes the following features:

- A POS injection process that initializes the POS device.
- DUKPT key management.
- Terminal Encryption Procedure 3 (TEP3) for PAN, Track1, Track2, and arbitrary random numeric data. The numeric data can be whatever data is necessary by the decrypting entity such that it can create the decrypting key.
- The LPOS, a new lightweight and modular SDK.

The POS Injection Process

Prior to encrypting data on a POS device, the processor injects the device with an initial IPEK and KSN. The following is the injection process:

1. On the Voltage SecureData Management Console, at the processor, you create a DUKPT district and BDK(s).
2. The BDK is exported to the processor.
3. The processor derives the IPEK from the BDK and KSN.
4. In a secure injection facility, the processor injects the POS terminals with the IPEK, and the KSN.
5. The POS device is now ready to start encrypting transactions.

The Key Management and Encryption Process

The key management and encryption process involves software on both the POS side and the Host side of the payments system. On the POS side, the software:

- Uses the VpLposLoadIPEK function to generate the initial set of future keys. From the future key corresponding to the transaction counter, the terminal generates the data and Message Authentication Code (MAC) keys that are used for TEP3 encryption of the Track and PAN data.
- Subsequently each time the terminal advances the transaction counter it a generates new key along with MAC keys corresponding to the transaction counter value. You send this information either embedded or not embedded in the ciphertext.

On the Host side, the encrypted data must be decrypted and processed:

- TEP3 encrypted data arrives from a terminal. The Host determines which BDK to use from the KSN value from the terminal. Using the derived BDK and the KSN, the Host derives the terminal's IPEK.
- Using the IPEK and the transaction counter which the host obtains from the KSN, the Host derives the decryption key and MAC keys to decrypt or translate the data.

The LPOS architecture and data flow is illustrated in Figure 1-1.

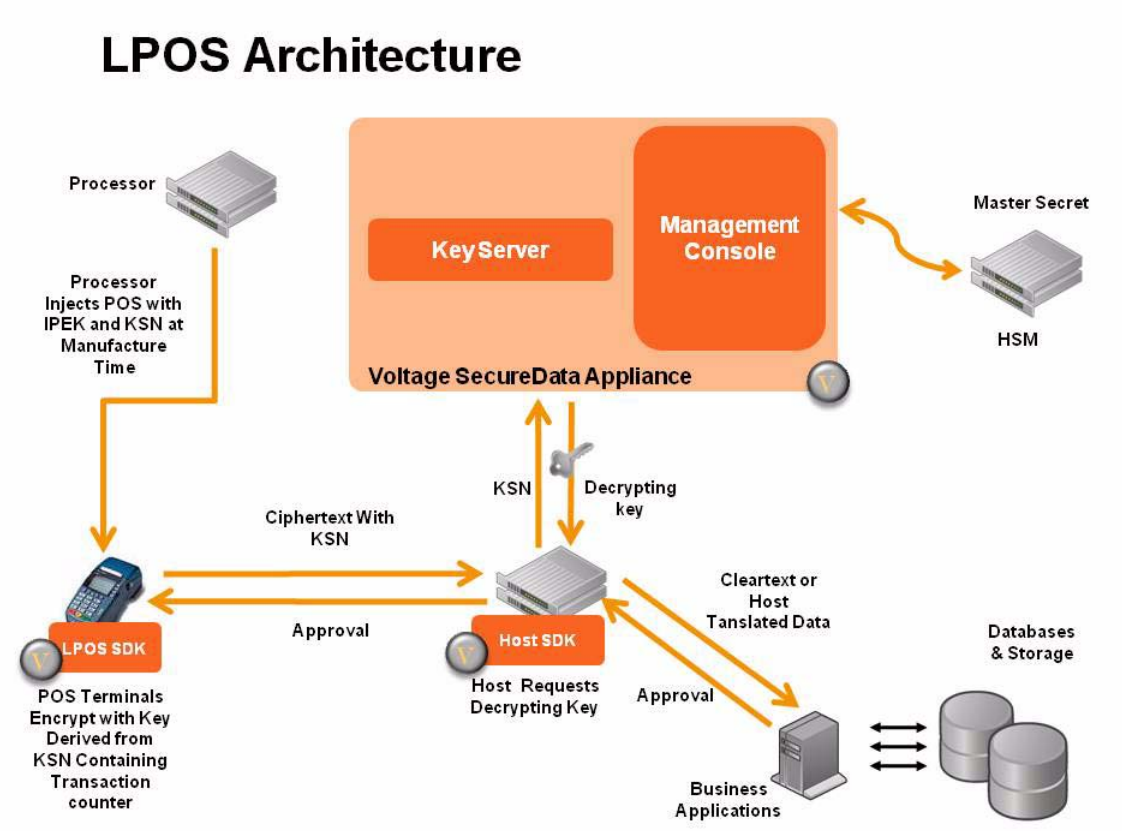

Figure 1-1 LPOS Data Flow

Terminal Encryption Procedure 3

DUKPT uses TEP3. TEP3 functionality differs depending on the type of data to be processed and encrypted. The way PANs are handled is distinct from the way Tracks are handled. TEP3 is similar to Voltage's TEP2 encryption, but supports the DUKPT key management system.

Track 1 and Track 2 Processing

The LPOS SDK handles any legal length Track 1 or 2 that conforms to the ISO specifications. Track 1 maximum length is 78 (79 with check digit). Track 2 maximum length is 39 (40 with check digit). The LPOS SDK calculates the minimums according to the specification to be 19 digits for Track 1 and 16 digits for Track 2.

TEP3 performs the following tasks for a plaintext Track1 and Track2:

- Encrypts the center digits of the data and passes through the remaining digits and Luhn checksum unchanged.
- Tweaks encryption of the center digits of the data with the leading and trailing digits.
- Encrypts all the characters of the discretionary field.
- Embeds a 20 bit KSN in the ciphertext.
- Embeds a variable length integrity check in the ciphertext.

PAN Processing

There are two different PAN encryption modes, external KSN and embedded KSN. Embedded KSN mode embeds the 80 bit KSN in the ciphertext, and changes the encryption alphabet to alpha-numeric. External KSN mode leaves all the encrypted digits numeric, but you must separately transmit the KSN to the Host for correct decryption. The choice of which of these modes to choose depends on your local implementation.

Pans using external mode must be between 12 and 19 digits, with the middle digits of the PAN being encrypted. The first 6 and last 4 digits remain as cleartext.

2

Implementing the LPOS API

This chapter examines a sample LPOS DUKPT application. The LPOS API functions initialize DUKPT encryption, retrieve the KSN and transaction counter state, advance the key state, and encrypt data. Each function within the LPOS API returns a status value. This value must be checked after a function has completed to ensure that execution was successful.

You may build an LPOS application on non-embedded systems such as Windows or Linux using CMake. The LPOS SDK contains the file CMakeLists.txt that supports the building of an LPOS application on any platform that has CMake set up with a development toolchain.

Initializing the Voltage DUKPT Software

To load the Voltage DUKPT encryption software, call VPLposLoadIPEK with the IPEK and the KSN. This function sets up the future keys for use in encryption.

```
/// In the following example, the IPEK and KSN are hardcoded. In an
// actual application, these values would be supplied by the
// DUKPT key management system.
static const unsigned char
testIPEK[]={0x6a,0xc2,0x92,0xfa,0xa1,0x31,0x5b,0x4d, 0x85, 0x8a,
0xb3, 0xa3, 0xd7, 0xd5, 0x93, 0x3a};

static const unsigned char
testKSN[]={0xff,0xff,0x98,0x76,0x54,0x32,0x10,0xe0,0x00,0x00};
vp_lpos_status_t status;

// Initialize DUKPT encryption
   status = VpLposLoadIPEK(testIPEK, testKSN);

// always check status
   if(status != VP_LPOS_SUCCESS) return FAIL;
```

Advancing the Key State

Prior to encrypting the data for the current transaction, you must advance the key state with VpLposAdvanceKeyState. This step is necessary to maintain the security of the system. Advancing the key state increments the transaction code within the KSN, and creates new encryption keys.

```
vp_lpos_status_t status;

// Advance key state.
status = VpLposAdvanceKeyState();
if(status != VP_LPOS_SUCCESS) return FAIL;
```

The Host SDK needs the entire KSN, and not just the compressed transaction counter. Each processor needs to have a process to map terminal IDs, merchant IDs, and other organizational IDs, to the first 59 bits of the KSN. The ANSI standard gives many different options to accomplish this. Some processor companies may choose the send the entire KSN in the application protocol, while others may want to use the embedded transaction counter mode and keep an old application protocol which includes sending the terminal ID and merchant ID. These processors need to map the terminal ID and merchant ID on the Host side to the first 59 bits of the KSN before passing them to the HOST SDK.

In the case of a power down on the POS, the KSN, IPEK, and transaction counter must be kept in non-volatile memory, or saved through some other implementation-dependent manner that supports the ability to retain the this information.

Encrypting Data

There are separate functions for each type of data that you encrypt.

Encrypting Track1 and Track2 Data

The function to encrypt Track1 data is VpLposEncryptTrack1. The function to encrypt Track2 data is VpLposEncryptTrack2. In this example, and the following ones, the cleartext data is hardcoded. This information would normally come from the credit card itself, and be placed within a buffer.

```
char trackBuf[79];
vp_lpos_status_t status;
unsigned char len;

// encrypt the data
status =
VpLposEncryptTrack1(
   "%B1111222233334444555^LONGNAMEERSON/
   EBENZEROSIS^140310110000A100000000286000000", 79, trackBuf, &len);
if(status != VP_LPOS_SUCCESS) return FAIL;

status =
VpLposEncryptTrack2(";1111222233334444555=14031011000011000?",
   39, trackBuf, &len);
if(status != VP_LPOS_SUCCESS) return FAIL;
```

Encrypting PAN and CVV Data

There are two ways to encrypt PAN and CVV data. You may either embed the KSN within the ciphertext, or leave it external to the ciphertext. If you have the KSN external to the ciphertext, the resulting encrypted data remains numeric. If you embed the KSN within the ciphertext, the resulting value is in base64 encoding.

Embedded Encrypting

The VpLposEncryptPanandCVV function encrypts data with the KSN embedded in the ciphertext. As the data is embedded, the resulting ciphertext for the PAN in panBuf may appear similar to "111122UAACljq+B4555", and the resulting ciphertext for the CVV in cvvBuf may appear as "37i9". If you do not have a CVV number, pass in the value of 0 and it will be ignored. It is not necessary to explicitly use the KSN when executing this function, as it is handled by the LPOS API. However, if you encrypt in external mode, you must send the KSN to the Host separately.

```
char panBuf[19];
char cvvBuf[4];
vp_lpos_status_t status;
unsigned char len;
unsigned char cvvlen;

// encrypt the data
status = VpLposEncryptPANandCVV("1111222233334444555", 19, "1234"
   4, panBuf, &len, cvvBuf, &cvvlen);
if(status != VP_LPOS_SUCCESS) return FAIL;
```

External Encrypting

You use VpLposEncryptExternalPANandCVV to encrypt in external mode. If you use externally encrypting, the KSN is not embedded in the ciphertext, and the resulting ciphertext remains numeric. For example, the encrypted versions of the cleartext below could as "1111221952104894555" for the PAN in panBuf, and “2162” for the CVV in cvvBuf. If you do not have a CVV number, pass in the value of 0 and it will be ignored. To properly encrypt/decrypt in external mode, you must pass the KSN to the Host separately.

---

note Voltage does not supply a method of transmitting data with external encryption.

```
char panBuf[19];
char cvvBuf[4];
vp_lpos_status_t status;
unsigned char len;
unsigned char cvvlen;

unsigned char counterBuf[3];

// retrieve the current ksn
```

```
status = VpLposGetCurrentTransactionCounter(counterBuf);
// encrypt the data
status = VpLposEncryptExternalPANandCVV("1111222233334444555", 19,
"1234", 4, panBuf, cvvBuf);
if(status != VP_LPOS_SUCCESS) return FAIL;

// software that sends cryptogram to processor (not supplied by
// Voltage)
```

Encrypting Numeric Data

VpLposEncryptNumericData encrypts random strings of numeric data. This data could consist of a merchant ID, terminal ID, or other numeric data required by an implementation. The data must fall into the range of 0-9, and the resulting encrypted value will also fall within the values of 0-9. You may encrypt any number containing up to 64 digits long.

```
char numBuf[19];
vp_lpos_status_t status;

//
// function to send KSN to processor (not supplied by Voltage

status = VpLposEncryptNumericData("1234567890987654321",
   19, 0, 0, numBuf);
```

Once all transaction data has been sent, and approval information is received, this process then can repeat by advancing the key state when new transaction data arrives.

3

LPOS API Function Reference

This chapter lists the DUKPT functions available in the LPOS API. Each function returns a status value defined as

```
typedef char vp_lpos_status_t;
```

This status value can contain one of the following codes:

```
VP_LPOS_SUCCESS                       0
VP_LPOS_INTERNAL_ERROR                -1
VP_LPOS_KEY_LIMIT_EXCEEDED            -2
VP_LPOS_ERROR_NULL_ARG                -3
VP_LPOS_ERROR_INPUT_LENGTH_TOO_LONG   -4
VP_LPOS_ERROR_INPUT_LENGTH_TOO_SHORT  -5
```

If a function returns successfully, VP_LPOS_SUCCESS is returned, otherwise there has been an error.

VpLposAdvanceKeyState

Advances the internal DUKPT key state, discarding the current encryption keys, and generates a new set of future keys corresponding to the next valid transaction counter value. Encryption functions called afterwards use the new transaction counter within the KSN.

This function must be called prior to each transaction to ensure the security of the system. In the case of a power down on the POS, this value must be kept in non-volatile memory, or through some other implementation-dependent manner that supports the ability of the transaction counter to be retrieved.

note This function affects the 21 bits of the KSN that constitute the transaction number. The other 59 bits are implementation dependent, and are not affected by this function.

Function Signature

```
vp_lpos_status_t VpLposAdvanceKeyState();
```

VpLposEncryptExternalPANandCVV

Format preserving encryption of the PAN and CVV values. The result lengths are identical to the input lengths. The KSN is not embedded in the ciphertext. PAN, CVV, PANresult, and CVVresult must be separate buffers.

Function Signature

```
vp_lpos_status_t VpLposEncryptExternalPANandCVV(
   const char *PAN,
   unsigned char PANlen,
   const char *CVV,
   unsigned char CVVlen,
   char *PANresult,
   char *CVVresult)
```

Parameters

PAN
Input. The cleartext PAN buffer.

PANlen
Input. The length of the PAN buffer.

CVV
Input. The cleartext CVV buffer. If not used, pass 0.

CVVlen
Input. The length of the CVV buffer.

PANresult
Output. The ciphertext PAN buffer.

CVVresult
Output. The ciphertext CVV buffer.

VpLposEncryptNumericData

Encrypts an arbitrary string of decimal digits. Input and result can be the same buffer. This function can encrypt data such as merchant IDs, terminal IDs, and any other data as necessary. You can transmit this information to the Host for implementation dependent processing.

Function Signature

```
vp_lpos_status_t VpLposEncryptNumericData(
   const char *cleartextData,
   unsigned char inputLen,
   const unsigned char *tweak,
   unsigned char tweakLen,
   char *result);
```

Parameters

cleartextData
Input. The buffer of cleartext data.

inputLen
Input. The length of the cleartextData buffer.

tweak
Input. An optional tweak value buffer. Pass 0 for no tweak.

tweakLen
Input. The length of the tweak buffer.

result
Output. The resulting ciphertext.

VpLposEncryptPANandCVV

Encrypts the PAN and CVV values, embedding the KSN into the ciphertext. PAN, CVV, PANresult, and CVVresult must be separate buffers.

Function Signature

```
vp_lpos_status_t VpLposEncryptPANandCVV(
   const char *PAN,
   unsigned char PANlen,
   const char *CVV,
   unsigned char CVVlen,
   char *PANresult,
   unsigned char *PANresultLen,
   char *CVVresult,
   unsigned char *CVVresultLen)
```

Parameters

PAN
Input. The cleartext PAN buffer.

PANlen
Input. The length of the PAN buffer.

CVV
Input. The cleartext CVV buffer. If not used, pass 0.

CVVlen
Input. The length of the CVV buffer.

PANresult
Output. The ciphertext PAN buffer.

PANresultLen
Output. The length of the ciphertext PAN result buffer.

CVVresult
Output. The ciphertext CVV buffer.

CVVresultLen
Output. The length of the ciphertext CVV result buffer.

VpLposEncryptTrack1

Encrypts Track 1 data.

Function Signature

```
vp_lpos_status_t VpLposEncryptTrack1(
   const char *cleartextTrack1,
   unsigned char inputLen,
   char *ciphertextTrack1,
   unsigned char *resultLen);
```

Parameters

cleartextTrack1
Input. The cleartext Track 1 data buffer. The input and result buffers must be separate.

inputLen
Input. The length of the cleartextTrack1 buffer.

ciphertextTrack1
Output. The resulting ciphertext buffer for Track1. The input and result buffers must be separate.

resultLen
Output. The length of the ciphertextTrack1 buffer.

VpLposEncryptTrack2

Encrypts Track 2 data.

Function Signature

```
vp_lpos_status_t VpLposEncryptTrack2(
   const char *cleartextTrack2,
   unsigned char inputLen,
   char *ciphertextTrack2,
   unsigned char *resultLen);
```

Parameters

cleartextTrack2
Input. The cleartext Track 2 data buffer. The input and result buffers must be separate.

inputLen
Input. The length of the cleartextTrack2 input buffer.

ciphertextTrack2
Output. The resulting ciphertext buffer. The input and result buffers must be separate.

resultLen
Output. The length of the ciphertextTrack2 buffer.

VpLposGetCurrentKSN

Retrieves the current KSN value. The buffer will be populated with the full 80 bit KSN. The first 59 bits are from the value supplied to VpLposLoadIPEK and the last 21 bits are from the current value of the transaction counter.

Function Signature

```
vp_lpos_status_t VpLposGetCurrentKSN( unsigned char ksn[10] );
```

Parameter

ksn
Output. The value of the current 80 bit KSN.

VpLposGetCurrentTransactionCounter

Retrieves the current transaction counter value. You must retrieve this value if you do not embed the KSN within the ciphertext when encrypting the PAN and CVV values.

note This function returns the 21 bits that constitute the transaction number. The other 59 bits are implementation dependent.

Function Signature

```
vp_lpos_status_t VpLposGetCurrentTransactionCounter(
   unsigned char counter[3]);
```

Parameter

counter
Output. The transaction counter value ranges from [1..2^21-1], encoded as three bytes in big-endian format. The first byte contains the top 5 bits, and ranges from [0..31].

VpLposGetVersionInfo

Retrieves the version number of the current LPOS SDK in use.

Function Signature

```
extern vp_lpos_status_t VpLposGetVersionInfo(
   unsigned short *lposVersionNumber,
   const char **lposVersionString);
```

Parameters

lPosVersionNumber
Output. The version information for the LPOS API. The value is in two byes, from MSB to LSB:

- 4 bits for 0 (reserved)
- 4 bits for the major version
- 4 bits for the minor version
- 4 bits for the patch level.

For example, version 3.2.1 would be encoded as 0x0321.

lposVersionString
Output. A printable, human-readable string useful for logging or bug reports.

VpLposLoadIPEK

Initializes the LPOS API with an IPEK and KSN value. When this function returns, the encryption functions are ready for use, and generate ciphertexts with an embedded transaction counter starting at 1. The processor provides the IPEK and KSN values injected into the POS device.

Function Signature

```
vp_lpos_status_t VpLposLoadIPEK(
   const unsigned char *IPEK, const unsigned char *KSN);
```

Parameters

IPEK
Input. The IPEK must have 16 bytes**.**

KSN
Input. The KSN must have at least 8 bytes, the first 59 bits used as prefix.

4

LPOS Modularity

The LPOS SDK implements both AES and DES encryption internally in software. Some POS devices may have this functionality built in to the hardware. The LPOS API supports the ability to replace the included software implementations with hardware supplied versions through an abstraction layer. This chapter discusses how to use the abstraction layer to replace the included software versions with the hardware based encryption software on a particular device.

There are two parts to the abstraction layer. One for AES and another for DES. AES is used to encrypt credit card information with FPE. DES is the DUKPT standard way to encrypt keys. The source files for the abstraction layer are in crypto.c and crypto.h, supplied with the SDK.

AES Encryption

The Voltage DUKPT implementation uses FPE/AES for encrypting data. There are three AES functions that may be altered to support hardware implementations:

- crypto_aes_key
- crypto_aes_enc
- crypto_aes_dec

The data type crypto_aes_ctx should consist of a suitable data structure to hold an AES key schedule in the alternate AES implementation. If the AES implementation does not support precomputed or saved key schedules, the data type can be defined to store a copy of the AES key for later use.

crypto_aes_key

Initializes an AES context with a key of the specified length in bytes. It returns CRYPTO_SUCCESS (0) for success, or CRYPTO_ERROR(-1) for failure.

Function Signature

```
crypto_ret_t crypto_aes_key(
   crypto_aes_ctx_p ctx,
   const unsigned char *key,
   U8 len)
```

crypto_ase_enc

Encrypts one 16-byte block in place.

Function Signature

```
void crypto_aes_enc(const crypto_aes_ctx_p ctx, unsigned char *buf)
```

crypto_aes_dec

Decrypts one 16-byte block in place.

Function Signature

```
void crypto_aes_dec(const crypto_aes_ctx_p ctx, unsigned char *buf)
```

DES Encryption

DUKPT and the LPOS SDK uses DES to encrypt keys. There are two DES encryption functions that may be altered to support hardware implementations:

- crypto_des
- crypto_3des_owf

crypto_des

Encrypts one block of data in place with the passed key. The block of data and the key are both 8 bytes.

Function Signature

```
void crypto_des(const unsigned char *key, unsigned char *buf)
```

crypto_3des_owf

Using parameter keybuf as a 3DES key, encrypts both blocks of keybuf in 3DES ECB mode in place. This one way function is used to derive data encryption keys in DUKPT.

Function Signature

```
void crypto_3des_owf(unsigned char *keybuf)
```

Parameter

keybuf
A input/output 16 byte buffer used as a 3DES key.

5

Verifying an Implementation

Voltage supports three different crypto protocols between the POS device and the associated Host device. The first two crypto protocol versions are based on the Identity Based-Key Encryption and Encapsulation Protocol (IB-KEEP). The third version, implemented in the LPOS, is based on DUKPT. Voltage Security has developed a set of Web Services that can perform verification of point-of-sale devices running the Voltage POS and LPOS APIs. This service is designated the “IB-KEEP Responder” and its primary function is to accept encrypted dummy credit card data and return the plaintext version to prove correct operation of the underlying cryptographic mechanism. It is not intended to offer a secure decryption facility and should be used for non-sensitive data only.

Use the Responder for automated testing of any POS terminal that implements the Voltage SecureData Payments solution. It is also an auto-responder that accepts requests from a client, either encrypted credit card data or specific error triggers, and returns decrypted values or error codes allowing the caller to exercise both positive and negative test cases. The service also provides a means for a client to request a set of pubic parameters suitable for performing identity based encryption (IBE) functions as part of these tests.

note The LPOS API only supports TEP3 encryption. The POS API only supports TEP1 and TEP2 encryption.

While the IB-Keep Responder supports both the POS and LPOS verification, only the LPOS verification service is explained in this document. For information on the POS verification service, see the *Voltage SecureData Payments POS SDK Developer Guide*.

Endpoint URL and WSDL

The IB-KEEP Responder service may be accessed using either a standard HTTP connection on port 80 or a secure HTTPS session over SSL port 443 at the following URLs

```
http://cloudburst.voltage.com/payments
https://cloudburst.voltage.com/payments
```

These URLs publish a simple landing page, containing links to both the endpoint URL of the service and the associated Web Services Definition Language (WSDL) file. Respectively, these URLs are:

```
http[s]://cloudburst.voltage.com/payments/services/
   IbkeepResponder
```

and

```
http[s]://cloudburst.voltage.com/payments/services/
   IbkeepResponder?wsdl
```

The target namespace of the service is: payments.voltage.com

Complex Types

Two custom, complex data types are defined in the WSDL to support the LPOS verification service:

VpExCardData

A sequence of up to five values defining either encrypted or plaintext credit card data, used to pass encrypted card data and to return results. Each element is represented by a string:

Elements:

- **cvv** CVV number
- **numericData** Any numeric data
- **pan** Primary Account Number
- **track1** Track 1 (IATA) data
- **track2** Track 2 (ABA) data

VpError

Used to return an integer error code and an associated human readable text message in the event of an error encountered during processing of a request. Contains two elements as follows:

Elements:

- **errCode** Numeric code identifying error
- **errMessage** Explanatory text for error reported

Error Codes

The following VpError codes may be returned by the IB-KEEP Responder service:

NOT_YET_IMPLEMENTED

Value

999

Description

A request has been received for a feature not supported by this version of the interface. Returned by any version prior to 2.0 or for any options value other than 0. Also returned for any request where specified version is later than that currently implemented.

ILLEGAL_VERSION

Value

998

Description

The requested version cannot be parsed into the format “X.Y” where X and Y are integers.

REQUIRED_INPUT_MISSING

Value

997

Description

A mandatory parameter has been omitted from a request.

CONFIGURATION_FAILURE

Value

996

Description

A problem has been encountered in initializing the IB-KEEP Responder service. For example, the native library containing the API could not be loaded. This indicates an error in the server setup and should not be returned to clients in normal use.

DECRYPTION_FAILURE

Value

995

Description

An attempt to decrypt credit card data was unsuccessful. This may occur because either the encrypted card data is invalid. The Host API returns a more detailed error code together with an explanatory string in the accompanying error message.

Example of Error Response

Specifying a version of “7.3” in any request to the IB-KEEP Responder will return a fault of the form:

```
<soapenv:Envelope
   xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
   <soapenv:Body>
      <soapenv:Fault>
         <faultcode>soapenv:Server.generalException</faultcode>
         <faultstring/>
         <detail>
            <ns1:fault xmlns:ns1="payments.voltage.com">
               <ns1:errCode>999</ns1:errCode>
               <ns1:errMessage>Version requested [7.3] is later than
                  current release [2.0]</ns1:errMessage>
            </ns1:fault>
            <ns2:exceptionName xmlns:ns2="http://xml.apache.org/axis/">
               com.voltage.payments.VpError</ns2:exceptionName>
            <ns3:hostname xmlns:ns3="http://xml.apache.org/axis/">
               cloudburst</ns3:hostname>
         </detail>
      </soapenv:Fault>
   </soapenv:Body>
</soapenv:Envelope>
```

Requests and Responses

Version 2.0 of the Payments Host Simulator Web Service implements these SOAP bindings for the LPOS SDK.

bdkArray()

This request returns the bdkArray. The bdkArray contains a list of delimited strings of the format "1CE49246E009EC28F84540915658691B|1" where "1CE49246E009EC28F84540915658691B" Is the DUKPT hex value and 1 is the BDK number. The LPOS tester can use this array to derive the IPEK for each tested LPOS device.

Example

Sending the below request:

```
<soapenv:Envelope
      xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
      xmlns:pay="payments.voltage.com">
   <soapenv:Header/>
   <soapenv:Body>
      <pay:bdkArray/>
   </soapenv:Body>
</soapenv:Envelope>
```

Causes the following response:

```
<soapenv:Envelope
   xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
   <soapenv:Body>
      <bdkArrayResponse xmlns="payments.voltage.com">
         <bdkArrayReturn>asdfaba</bdkArrayReturn>
      </bdkArrayResponse>
   </soapenv:Body>
</soapenv:Envelope>
```

currentVersion()

This request is used to identify the version of the IB-KEEP Responder service currently running on the server. It takes no input parameters and returns a response containing a string of the form X.Y where X and Y are the major and minor version numbers.

This request returns the value of 2.0

Example

Sending the below request:

```
<soapenv:Envelope
      xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
      xmlns:pay="payments.voltage.com">
   <soapenv:Header/>
   <soapenv:Body>
      <pay:currentVersion/>
   </soapenv:Body>
</soapenv:Envelope>
```

Causes the following response:

```
<soapenv:Envelope
   xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
   <soapenv:Body>
      <currentVersionResponse xmlns="payments.voltage.com">
         <currentVersionReturn>2.0</currentVersionReturn>
```

```
      </currentVersionResponse>
   </soapenv:Body>
</soapenv:Envelope>
```

dposCardDataDecrypt

This request is used to pass DUKPT encrypted credit card data and information to the server and obtain the corresponding plaintext values.

Parameters

version

xsd:string

Indicates the version of the IB-KEEP Responder service being requested. Specified in the form of major and minor numbers separated by a period. Omit or pass empty string for current (latest) version.

callerID

xsd:string

A string allowing the POS vendor to identify themselves to the service. This may be used later to separate logging information and should be represented using a canonical form such as com.voltage.pos.device1.

referenceID

xsd:string

Optional client-defined data used to identify this specific request. It might contain a terminal ID, amount, or transaction data that are meaningful to the client application. This value is not interpreted by the server, but in future releases will be logged for future reference.

bdkNum

xsd:long

The BDK number associated with the BDK.

ksn

xsd:string

A user supplied DUKPT KSN value used for IPEK derivation.

externCount

xsd:int

Specifies if the transaction counter is external or embedded. Zero means embedded, and non-zero represents the transaction counter value used for encryption.

encryptedData

impl:VpExCardData

The encrypted credit card data to be processed. Any combination of data types (CVV, Numeric data, PAN, Tracks 1 or 2) may be included.

---

note When the externCount parameter is zero, for embedded transaction counters, and the transaction counter is greater than one, the numeric data should not be included in the same request. You must use a separate request for the numeric data with the correct transaction counter.

Example

Assume a request of the type shown below is sent to the IB-KEEP Responder. The encrypted and decrypted values are for this example only, and are not valid data:

```
<soapenv:Envelope
   xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
   xmlns:pay="payments.voltage.com">
<soapenv:Header/>
   <soapenv:Body>
      <pay:dposCardDataDecrypt>
         <pay:callerId>com.voltage.pos.device1</pay:callerId>
         <pay:bdkNum>54</pay:bdkNum>
         <pay:ksn>abcdadfkdfjsdlkfj</pay:ksn>
         <pay:externCount>5</pay:externCount>
         <pay:encryptedData>
            <pay:cvv>7522735512487118</pay:cvv>
            <pay:numericData>fjkdsa44</pay:numericData>
            <pay:pan>+++++++C3pOLAiGz</pay:pan>
            <pay:track1>
               w9KpXMJ8yXKSxRR0YeyHrKOA5NnnVCX3zyiQRHwEsodMC62gd
            </pay:track1>
            <pay:track2>9ylrapOUL+oQDWmx88rsQSyO0Sq</pay:track2>
         </pay:encryptedData>
      </pay:dposCardDataDecrypt>
   </soapenv:Body>
</soapenv:Envelope>
```

A response of the following form is returned:

```
<soapenv:Envelope
   xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
   <soapenv:Body>
      <dposCardDataDecryptResponse xmlns="payments.voltage.com">
         <dposCardDataDecryptReturn>
            <cvv>414</mid>
            <numericData>59555494942</numericData>
```

```
            <pan>7777771000000000</pan>
            <track1>%B3333333310000000000^840JOHNSON/GEORGE^1 1062 22?
            </track1>
            <track2>;3333331000000000=110622?8</track2>
         </dposCardDataDecryptReturn>
      </dposCardDataDecryptResponse>
   </soapenv:Body>
</soapenv:Envelope>
```

Client Considerations

To access the IB-KEEP Responder Web Service, any SOAP-compliant client that adheres to the published WSDL may be used. The Voltage service makes no assumptions regarding the client platform or the nature of the application software being used.

For initial familiarization with the interface, a graphical tool such as **soapUI** by *eviware* is recommended. Version 3.5.1 is currently available and may be downloaded from http://www.soapui.org/.

The best approach for building a client application depends on the development environment and the available tools. Java developers, for example, might consider Apache Axis (http://ws.apache.org/axis). Alternatively, gSOAP (http://www.cs.fsu.edu/~engelen/soap.html) is an open source, cross platform toolkit for developing XML Web Services in C and C++.

For Further Information

For further information regarding the use of the LPOS SDK, contact Voltage Support at POSSDK_Support@voltage.com.