Protegrity APIs, UDFs, and Commands Reference Guide 9.1.0.0

Created on: Nov 19, 2024

Copyright

Table of Contents

Chapter 1

Introduction to this Guide

This guide provides information about all the APIs, UDFs, and commands in all Protegrity Protectors.

1.1 Sections Contained in this Guide

The guide is broadly divided into the following sections:

- Section *Introduction to this Guide* defines the purpose and scope for this guide. In addition, it explains how information is organized in this guide.
- Section *Application Protector* provides information about the API elements and their parameters, including data types and usage.
- Section *Big Data Protector* describes about the APIs available in the Big Data Protector.
- Section *Database Protector* provides information about all the UDFs that are available in Database Protector.
- Section *z/OS Protector UDFs* describes about all the UDFs that are available for Mainframe z/OS.
- Section *Appendix A: DevOps REST APIs* provides information about using the policy management functions through the REST APIs.

1.2 Accessing the Protegrity documentation suite

This section describes the methods to access the *Protegrity Documentation Suite* using the *My.Protegrity* portal.

Chapter 2

Application Protector

This section describes the APIs that are supported by the Protegrity Application Protector.

The Application Protector contains APIs, which perform the following functions:

- Fetches the policy related information from the shared memory
- Applies the access control settings that are derived on the basis of policy settings
- Encrypts or tokenizes the data based on the policy settings
- Generates audit logs that are sent to the PEP server

Note:

To reduce performance issues that occur due to protection of data or casting of data, a general best practice is to protect the data and present the unprotect APIs, UDFs, or Commands, as applicable, to authorized users only. This eliminates access of the unauthorized users to the unprotection APIs, UDFs, or Commands as the data is in protected form only.

The unprotection of protected data is therefore limited to authorized users and does not cause a significant performance impact as the APIs, UDFs, or Commands are executed restrictively.

Note:

If you are using Format Preserving Encryption (FPE) with the reprotect API, then ensure that the *Plaintext* encoding used for FPE must be the same for both protecting and reprotecting the data. For example, if you have used the FPE-Numeric data element with UTF-8 encoding to protect the data, then you must use only the FPE-Numeric data element with UTF-8 encoding to reprotect the data.

If you are using Unicode Gen2 type tokenization Data-element with the reprotect API, then ensure that the *Default* encoding used for Unicode Gen2 type tokenization must be the same for protecting and reprotecting the data. For example, if you have used the Unicode Gen2 data element with UTF-8 encoding to protect the data, then you must use only the Unicode Gen2 data element with UTF-8 encoding to reprotect the data.

Note:

If you are using Format Preserving Encryption (FPE) and Byte APIs, then ensure that the encoding, which is used to convert the string input data to bytes, matches the encoding that is selected in the **Plaintext Encoding** drop-down for the required FPE data element.

If you are using Unicode Gen2 data element and Byte APIs, then ensure that the encoding, which is used to convert the string input data to bytes, matches the encoding that is selected in the **Default Encoding** drop-down for the required Unicode Gen2 data element.

2.1 Application Protector (AP) C APIs

The Protegrity Application Protector (AP) C provides APIs that integrate with the customer application to protect, unprotect, and reprotect sensitive data. A session must be created to run the AP C.

Note: The AP C APIs can be invoked by a valid *Policy User*.

Note:
The AP C supports only the *byte* data type.

The following diagram represents the basic flow of a session.

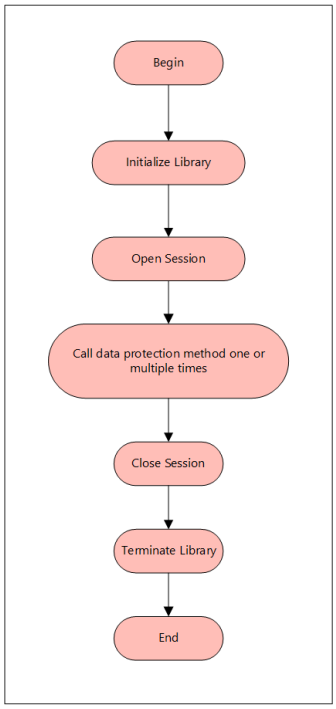

Figure 2-1: Flowchart for AP C

The following sections provide detailed information of the various structures and functions used by the Protegrity Application Protector C.

2.1.1 stXC_PARAM_EX

The *stXC_PARAM_EX* structure is used for the XC APIs that are used by the AP C. This structure contains metadata and information related to protect, unprotect, and reprotect operations.

The following is a sample of the *stXC_PARAM_EX* structure.

```
typedef struct stXC_PARAM_EX
{
  XC_CHAR   szVendor[XC_MAX_VENDOR_NAME_SIZE];       /* The vendor that is using the API,
should be set to XC_VENDOR_XC */
  XC_UINT8  ui8ScambledSessionHandle;                /* For internal use, Do not use ! */
  XC_CHAR   cUserIp[XC_MAX_IP_ADDRESS_SIZE];         /* The IP of the caller, to be used in
audit records */
  XC_UINT4  ui4Operation;                            /* The kind of operation that you want
to do.  */
  XC_UINT4  ui4DataType;                             /* The datatype of the data that will be
protected/unprotected  */
  XC_CHAR   cPlainTextEncoding[XC_ENCODING_SIZE];    /* Supported Encoding for FPE DE */
  XC_BYTE   bOldExternalTweak[XC_MAX_TWEAK_SIZE];    /* Old External tweak */
  XC_UINT4  ui4OldExternalTweakLength;               /* Old External tweak length */
  XC_BYTE   bNewExternalTweak[XC_MAX_TWEAK_SIZE];    /* New External tweak in case of
reprotect operation */
  XC_UINT4  ui4NewExternalTweakLength;               /* New External tweak length */
} XC_PARAM_EX;
```

2.1.2 stXC_DATA_ITEM

> **Note:**
> It is recommended to use *stXC_DATA_ITEM_EX* structure over *stXC_DATA_ITEM* structure.

The *stXC_DATA_ITEM* structure is used as a container to store the protected or unprotected data.

The following is a sample for the *stXC_DATA_ITEM* structure.

```
typedef struct stXC_DATA_ITEM
{
  XC_UINT4 ui4Capacity; /* Max number of bytes that can be stored in pvValue  */
  XC_UINT4 ui4Length;   /* Actual number of bytes stored in pvValue           */
  XC_VOID* pvValue;     /* The actual value of the data                       */
} XC_DATA_ITEM;
```

2.1.3 stXC_DATA_ITEM_EX

The *stXC_DATA_ITEM_EX* structure is typically used for bulk operations with the various APIs.

```
typedef struct stXC_DATA_ITEM_EX
{
  XC_UINT4 ui4Capacity;    /**< Max number of bytes that can be stored in pvValue  */
  XC_UINT4 ui4Length;      /**< Actual number of bytes stored in pvValue           */
  XC_VOID* pvValue;        /**< The actual value of the data                       */
  XC_BYTE  bNullIndicator; /**< Used to set if the data is null or not. XC_TRUE = Null.
XC_FALSE = not null  */
  XC_UINT4 ui4ErrorCode;   /**< The errorcode for the returning data struct, if an error
occured, otherwise it is PEP_LOG_PROTECT_SUCCESS or PEP_LOG_UNPROTECT_SUCCESS */
} XC_DATA_ITEM_EX;
```

2.1.4 stXC_ACTION_RESULT

For each bulk call, the *stXC_ACTION_RESULT* structure contains a summary status for the entire batch. This structure is not applicable to AP Lite which does not support bulk operations.

The following is a sample for the *stXC_ACTION_RESULT* structure.

```
typedef struct stXC_ACTION_RESULT
{
  XC_UINT4 ui4LogReturnCode; /* This is the returncode of the operation */
  XC_UINT4 ui4LogSeverity;   /* This is the severity,  SUCCESS, SUCCESS WITH WARNING or ERROR
*/
  XC_UINT4 ui4NoAccessOperation; /* What should we return if we do not have access to
unprotect data, NULL, EXCEPTION, PROTECTED VALUE or NOACCESS VALUE */
  XC_UINT4 ui4ProtectionAlgId; /* Algorithm that was used for protecting/unprotecting */
  XC_UINT4 ui4TokenType;       /* Type of data being tokenized */
  XC_UINT4 ui4OutputEncoding;  /* Encoding of output data */
  XC_MASK_SETTINGS stMaskSettings; /* Indicates type of masking used */
} stXC_ACTION_RESULT;
```

The following parameters are important for the bulk calls:

- **ui4LogReturnCode** contains only the most severe error code in the processed batch (for example, if there is one success with warning entry and one with severe error, the severe error entry will be logged).
- **ui4LogSeverity** reports on the status of the processed batch.

The **ui4LogSeverity** has three different types. The first type applies to successful operations. For erroneous operations, there are two different states (success with warning and error) pointing to what went wrong in a batch.

The following table describes the various ui4LogSeverity types.

ui4LogSeverity Type	**ui4LogSeverity Possible Errors**	
Success (operation completed successfully)	Operation completed successfully - no exception	
Success With Warning (operation completed successfully but some data failed)	Policy constraints: • User/DE/Time/Access • Load Key (Key ID) • Audit failed	Data constraints: • Integrity Check failed • Invalid format • Length • Token alphabet constraints
Error (Major error- batch failed)	System constraints: • PEP server not running • Fatal errors	Exceptions: • Policy locked • Policy not available • Unsupported algorithm • Input Parameter missing • Disk full • No access operation = exception (Policy set) • Out of memory • License expired

By turn, each data item in a batch contains its own log return code. It is defined by *ui4ErrorCode* for the AP C API.

2.1.5 stXC_MASK_SETTINGS

The *stXC_MASK_SETTINGS* structure holds information about the mask settings applied on the data element for access and audit purposes.

The following is a sample for the *stXC_MASK_SETTINGS* structure.

```
typedef struct stXC_MASK_SETTINGS
{
  XC_UINT4 bLeftIsMasked  : 1;
  XC_UINT4 bLeft          : 7;
  XC_UINT4 bRightIsMasked : 1;
  XC_UINT4 bRight         : 7;
  XC_UINT4 bFillerBits    : 16;
  XC_BYTE bMaskCharacter;
} XC_MASK_SETTINGS;
```

2.1.6 eXC_LOGRETURNTYPE

The *eXC_LOGRETURNTYPE* enum indicates the type of the log return code depending on the value returned.

The following is a sample for the *eXC_LOGRETURNTYPE* enum.

```
typedef enum eXC_LOGRETURNTYPE
{
  XC_LOGRETURNSUCCESS        = 0,  /* Success, no additional test            */
  XC_LOGRETURNSUCCESSWARNING = 1,  /* Success, with additional warning text  */
  XC_LOGRETURNERROR          = 2,  /* Error type of logreturn code           */
  XC_LOGRETURNEXCEPTION      = 3   /* If we want to throw exception if no acceess */
} XC_LOGRETURNTYPE;
```

2.1.7 eXC_FUNCTION

The *eXC_FUNCTION* enum indicates the type of algorithm used while protecting the data.

The following is a sample for the *eXC_FUNCTION* enum.

```
typedef enum eXC_FUNCTION
{
  XC_ANY_FUNCTION = 0,              /* This is the default - some kind of cipher function */
  XC_HMAC_FUNCTION = 1,             /* Hash message digest - one way cipher function      */
  XC_TYPE_PRESERVING_FUNCTION = 2,  /* Type Preserving, i.e. Token, NoEncryption        */
  XC_CRYPTO_FUNCTION = 3            /* Regular encryption algorithm           */
} XC_FUNCTION;
```

2.1.8 eXC_DATATYPE

The *eXC_DATATYPE* enum indicates the type of datatype you want to protect.

The following is a sample for the *eXC_DATATYPE* enum.

```
typedef enum eXC_DATATYPE
{
  XC_DATATYPE_BYTE = 0,     /* Byte */
  XC_DATATYPE_CHARACTER,    /* Character data */
  XC_DATATYPE_UNICODE,      /* Unicode data */
  XC_DATATYPE_DATE,         /* Date data */
  XC_DATATYPE_INTEGER,      /* Integer data */
  XC_DATATYPE_OTHER         /* For example real, blob, clob, time, timestamp */
} XC_DATATYPE;
```

2.1.9 XCInitLib Function

The *XCInitLib* function initializes the Protegrity AP library. This should be performed once in each application that uses the Protegrity AP API. When the application terminates, it should call the corresponding terminate function.

This function returns *XC_SUCCESS* on success.

The following is a sample for the *XCInitLib* function.

```
XCInitLib( XC_HANDLE* phXCHandle,
           const XC_CHAR* pcParameter );
```

The following table lists the various parameters used for the *XCInitLib* function.

Parameter	Description	Data Type
phXCHandle	[out] A handle for the library. It should be used for the other functions.	XC_HANDLE*
pcParameter	[in] NULL terminated string containing the parameter. **Caution:** This parameter is not used.	const XC_CHAR*

2.1.10 XCTerminateLib Function

The *XCTerminateLib* function terminates the Protegrity AP library. This should be performed when the application terminates. No other functions should be invoked after invoking the *XCTerminateLib* function. It should always be invoked after a call to the *XCInitLib* function for cleanup purposes.

This function returns *XC_SUCCESS* on success.

The following is a sample of the *XCTerminateLib* function.

```
XCTerminateLib( XC_HANDLE* phXCHandle );
```

The following table lists the parameter used for the *XCTerminateLib* function.

Parameter	Description	Data Type
phXCHandle	[in/out] A handle for the library that has been initialized.	XC_HANDLE*

2.1.11 XCGetVersion Function

The *XCGetVersion* function obtains a null terminated version string for the Protegrity AP.

This function returns *XC_SUCCESS* on success.

The following is a sample for the *XCGetVersion* function.

```
XCGetVersion( XC_CHAR* pszVersion,
              const XC_UINT4 ui4VersionLength );
```

The following table lists the various parameters used for the *XCGetVersion* function.

Parameter	Description	Data Type
pszVersion	[in/out] The null terminated version string is returned. The buffer needs to be allocated before it is sent to the function.	XC_CHAR*
ui4VersionLength	[in] The length of the buffer allocated for the version.	const XC_UINT4

2.1.12 XCGetVersionEx Function

The *XCGetVersionEx* function obtains a null terminated version string for the Protegrity AP. It returns the current version of the AP C and the core version.

The following is a sample for the *XCGetVersionEx* function.

```
XCGetVersionEx();
```

2.1.13 XCGetCoreVersion Function

The *XCGetCoreVersion* function obtains a null terminated version string for the Protegrity AP.

This function returns *XC_SUCCESS* on success.

The following is a sample for the *XCGetCoreVersion* function.

```
XCGetVersion( XC_CHAR* pszCoreVersion,
              const XC_UINT4 ui4VersionLength );
```

The following table lists the various parameters used for the *XCGetCoreVersion* function.

Parameter	Description	Data Type
pszCoreVersion	[in/out] The null terminated version string is returned. The buffer needs to be allocated before it is sent to the function.	XC_CHAR*
ui4VersionLength	[in] The length of the buffer allocated for the version.	const XC_UINT4

2.1.14 XCOpenSession Function

The *XCOpenSession* function opens a session and returns a handle for that session to be used in calls to the *XCProtect* and *XCUnprotect* functions. When the session is no longer needed, it should be closed by a call to *XCCloseSession* function.

This function returns *XC_SUCCESS* on success.

The following is a sample for the *XCOpenSession* function.

```
XCOpenSession( const XC_HANDLE    hXCHandle,
               const XC_CHAR*     pcUser,
               const XC_CHAR*     pcPassword,
               const XC_CHAR*     pcParameter,
               XC_SESSION*        phSession );
```

The following table lists the various parameters used for the *XCOpenSession* function.

Parameter	Description	Data Type
hXCHandle	[in] A handle for the library that has been initialized.	const XC_HANDLE
pcUser	[in] NULL terminated string for the user opening the session. **Important:** This parameter is only applicable for the AP Lite.	const XC_CHAR*
pcPassword	[in] NULL terminated string for the password for that user. **Important:** This parameter is only applicable for the AP Lite.	const XC_CHAR*
pcParameter	[in] NULL terminated string containing the parameter needed to create a session. For AP Client, the required parameter is *ipaddress;port;TCP* for the PEP server that is configured to handle the AP requests. For AP Lite, the required parameter is the *<export_keys_filename>.xml* file, which is the key export file from REST API service. For the *XCPep.plm* file, the parameter *0* needs to be set for the communicationid.	const XC_CHAR*
phSession	[out] On success, this will point to the handle for the established session.	XC_SESSION*

2.1.15 XCCloseSession Function

The *XCCloseSession* function will terminate an established session and reset the handle for it, after which no calls to *XCProtect* and *XCUnprotect* functions will work.

This function returns *XC_SUCCESS* on success.

The following is a sample for the *XCCloseSession* function.

```
XCCloseSession( const XC_HANDLE hXCHandle,
                XC_SESSION* phSession );
```

The following table lists the various parameters used for the *XCCloseSession* function.

Parameter	Description	Data Type
hXCHandle	[in] Handle for the library that has been initialized.	const XC_HANDLE
phSession	[in/out] Session to close.	XC_SESSION*

2.1.16 XCFlushPepAudits Function

The *XCFlushPepAudits* function is used for flushing the audit logs at any point within the application. This API is required for a short running process that lasts less than a second, to get the audit logs. It is recommended to invoke it at the point where the application exits, but this should be done before invoking the *XCTerminateLib* function.

The following is a sample of the *XCFlushPepAudits* function.

```
XCFlushPepAudits( XC_SESSION hSession );
```

The following table lists the parameter used for the *XCFlushPepAudits* function.

Parameter	Description	Data Type
hSession	[in/out] Handle to a previously opened provider.	XC_SESSION

2.1.17 XCProtect Function

The *XCProtect* function will take the supplied plaintext and send a request to the Protegrity AP that will protect the data. It will then read the response from the Protegrity AP and return the cipher text to the caller.

This function returns *XC_SUCCESS* on success.

Note:
You cannot move the data that is protected using encryption data elements with input as integers, long, or short data types and output as bytes, between platforms having different endianness.

For example, if the data is protected using encryption data elements with input as integers and output as bytes, then you cannot move the protected data from the AIX platform to the Linux or Windows platform and vice versa.

Note: If you are using Format Preserving Encryption (FPE) and Byte APIs, then ensure that the encoding, which is used to convert the *string* input data to *bytes*, matches the encoding that is selected in the **Plaintext Encoding** drop-down for the required FPE data element.

Warning:
For Date and DateTime type of data elements, the *XCProtect* API returns an invalid input data error if the input value falls between the non-existent date range from 05-OCT-1582 to 14-OCT-1582 of the Gregorian Calendar.

For more information about the tokenization and de-tokenization of the cutover dates of the Proleptic Gregorian Calendar, refer to the sections *Date Tokenization for cutover Dates of the Proleptic Gregorian Calendar* and *Datetime Tokenization for Cutover Dates of the Proleptic Gregorian Calendar* in the *Protection Method Reference Guide 9.1.0.0*.

The following is a sample of the *XCProtect* function.

```
XCProtect( const XC_HANDLE     hXCHandle,
           const XC_SESSION    hSession,
           const XC_UINT4      ui4EventType,
           const XC_CHAR*      pcPolicyUser,
           const XC_CHAR*      pcDataElement,
           const XC_BYTE*      pcExternalIV,
           const XC_UINT4      ui4ExternalIVLength,
           const XC_BYTE*      pcInputData,
```

```
            const XC_UINT4      ui4InputDataLength,
            const XC_BYTE       bNiInputData,
            XC_BYTE*            pcOutputData,
            XC_UINT4*           pui4OutputDataLength,
            XC_BYTE*            pbNiOutputData,
            const XC_PARAM_EX*  pXCParam,
            const XC_UINT4      ui4XCParamSize,
            stXC_ACTION_RESULT* pstActionResult );
```

The following table lists the various parameters used for the *XCProtect* function.

Parameter	Description	Data Type
hXCHandle	[in] Handle for the library that has been initialized.	const XC_HANDLE
hSession	[in] Handle for the established session.	const XC_SESSION

ui4EventType	[in] Used to identify whether the call type of the function is First Call or Normal Call.
	
	This parameter can be set to one of the following values:
	
	• **1**: Internally sets the XC event type to *XC_EVENT_FIRST_CALL*. This identifies the call type as First Call. If the call type is First Call, then data is written to the internal cache, which includes:
	  • SessionID
	  • RequestID
	  • DataElement
	  • UserName
	  • ProductID
	  • VendorID
	• **0**: Identifies the call type as Normal Call. If the call type is Normal Call, then the data is only read from the internal cache. Data is written to the internal cache only if it is different from the one that was written during the First Call.
	
	Important: An entry is generated in the *pepserver.log* file only if any new data is written to the internal cache.
	
	Note:
	If logging is not enabled, but the *XC_EVENT_FIRST_CALL* parameter is set, then logs are not generated.
	
	Similarly, if logging is enabled, but *XC_EVENT_FIRST_CALL* is not set, then logs are generated only if any new data is written to the cache.
		const XC_UINT4

pcPolicyUser	[in] NULL terminated string for the policy user.	const XC_CHAR *
pcDataElement	[in] NULL terminated string for the data element.	const XC_CHAR *
pcExternalIV	[in] Buffer containing data that will be used as external IV.	const XC_BYTE*

Parameter	Description	Data Type
ui4ExternalIVLength	[in] Number of bytes provided in the *pcExternalIv* buffer. The total amount of data to be protected should not exceed 256 bytes.	const XC_UINT4
pcInputData	[in] Buffer containing the plain data to encrypt.	const XC_BYTE *
ui4InputDataLength	[in] Number of bytes contained in *pcInputData*. The total amount of data to be protected should not exceed 2 GB.	const XC_UINT4
bNiInputData	[in] Flag to indicate NULL input.	const XC_BYTE
pcOutputData	[out] Buffer that will hold the resulting encrypted data. Needs to be larger than the input buffer.	XC_BYTE *
pui4OutputDataLength	[in/out] On input it should specify the max size of the output data buffer. When the function returns, it will contain the actual number of bytes stored in the output. This size needs to be big enough to hold the resulting data as well as the packed request. Consider having an output buffer that is *XC_BYTES_OVERHEAD* bytes larger than the input buffer for additional overhead. If the value is set to zero as **pui4OutputDataLength = 0;*, then the following value is returned. **pui4OutputDataLength = ui4InputDataLength + XC_BYTES_OVERHEAD;*	XC_UINT4 *
pbNiOutputData	[out] Flag indicating that the output is NULL.	XC_BYTE *
pXCParam_Ex	[in] Additional information associated with the protection, refer to *XC_Param_Ex* function.	const XC_PARAM_EX*
ui4XCParamSize	[in] Size of the structure passed in the *pXCParam* parameter.	const XC_UINT4
pstActionResult	[out] Structure containing the result of the operation. For more information refer to the *xcdefinitions.h* file.	stXC_ACTION_RESULT*

2.1.18 XCUnprotect Function

The *XCUnprotect* function will take the supplied ciphered data and send a request to the Protegrity AP that will unprotect the data. It will then read the response from the Protegrity AP and return the plain data to the caller.

This function returns *XC_SUCCESS* on success.

> **Note:** If you are using Format Preserving Encryption (FPE) and Byte APIs, then ensure that the encoding selected in the **Plaintext Encoding** drop-down for the required FPE data element matches the encoding that is used to convert the protected *byte* input data to *string*.

The following is a sample of the *XCUnprotect* function.

```
XCUnprotect( const XC_HANDLE     hXCHandle,
             const XC_SESSION    hSession,
```

```
            const XC_UINT4      ui4EventType,
            const XC_CHAR*      pcPolicyUser,
            const XC_CHAR*      pcDataElement,
            const XC_BYTE*      pcExternalIV,
            const XC_UINT4      ui4ExternalIVLength,
            const XC_BYTE*      pcInputData,
            const XC_UINT4      ui4InputDataLength,
            const XC_BYTE       bNiInputData,
            XC_BYTE*            pcOutputData,
            XC_UINT4*           pui4OutputDataLength,
            XC_BYTE*            pbNiOutputData,
            const XC_PARAM_EX*  pXCParam,
            const XC_UINT4      ui4XCParamSize,
            stXC_ACTION_RESULT* pstActionResult );
```

The following table lists the various parameters used for the *XCUnprotect* function.

Parameter	Description	Data Type
hXCHandle	[in] Handle for the library that has been initialized.	const XC_HANDLE
hSession	[in] Handle for the established session.	const XC_SESSION

ui4EventType	[in] Used to identify whether the call type of the function is First Call or Normal Call.
	
	This parameter can be set to one of the following values:
	
	• **1**: Internally sets the XC event type to *XC_EVENT_FIRST_CALL*. This identifies the call type as First Call. If the call type is First Call, then data is written to the internal cache, which includes:
	  • SessionID
	  • RequestID
	  • DataElement
	  • UserName
	  • ProductID
	  • VendorID
	• **0**: Identifies the call type as Normal Call. If the call type is Normal Call, then the data is only read from the internal cache. Data is written to the internal cache only if it is different from the one that was written during the First Call.
	
	Important: An entry is generated in the *pepserver.log* file only if any new data is written to the internal cache.
	
	Note:
	If logging is not enabled, but the *XC_EVENT_FIRST_CALL* parameter is set, then logs are not generated.
	
	Similarly, if logging is enabled, but *XC_EVENT_FIRST_CALL* is not set, then logs are generated only if any new data is written to the cache.
		const XC_UINT4

pcPolicyUser	[in] NULL terminated string for the policy user.	const XC_CHAR *

Parameter	Description	Data Type
pcDataElement	[in] NULL terminated string for the data element.	const XC_CHAR*
pcExternalIV	[in] Buffer containing data that will be used as external IV.	const XC_BYTE*
ui4ExternalIVLength	[in] Number of bytes provided in the *pcExternalIv* buffer. The total amount of data to be unprotected should not exceed 256 bytes.	const SD_UINT4
pcInputData	[in] Buffer containing the plain data to decrypt.	const XC_BYTE*
ui4InputDataLength	[in] Number of bytes contained in *pcInputData*. The total amount of data to be unprotected should not exceed 2 GB.	const XC_UINT4
bNiInputData	[in] Flag to indicate NULL input.	const XC_BYTE
pcOutputData	[out] Buffer that will hold the resulting decrypted data. Needs to be larger than the input buffer.	XC_BYTE*
pui4OutputDataLength	[in/out] On input it should specify the max size of the output data buffer. When the function returns, it will contain the actual number of bytes stored in the output. This size needs to be big enough to hold the resulting data as well as the packed request. Consider having an output buffer that is *XC_BYTES_OVERHEAD* bytes larger than the input buffer for additional overhead. If the value is set to zero as **pui4OutputDataLength = 0;*, then the following value is returned. **pui4OutputDataLength = ui4InputDataLength + XC_BYTES_OVERHEAD;*	XC_UINT4*
pbNiOutputData	[out] Flag indicating that the output is NULL.	XC_BYTE*
pXCParam	[in] Additional information associated with the protection, refer to the *XC_Param_Ex* function.	const XC_PARAM_EX*
ui4XCParamSize	[in] Size of the structure passed in the *pXCParam* parameter.	const XC_UINT4
pstActionResult	[out] Structure containing the result of the operation. For more information refer to the *xcdefinitions.h* file.	stXC_ACTION_RESULT*

2.1.19 XCReprotect Function

The *XCReprotect* function will take the supplied protected data and send a request to the Protegrity AP that will reprotect the data. It will then read the response from the Protegrity AP and return the cipher text to the caller.

This function returns *XC_SUCCESS* on success.

The following is a sample for the *XCReprotect* function.

```
XCReprotect( const XC_HANDLE     hXCHandle,
             const XC_SESSION    hSession,
```

```
            const XC_UINT4      ui4EventType,
            const XC_CHAR*      pcPolicyUser,
            const XC_CHAR*      pcOldDataElement,
            const XC_CHAR*      pcNewDataElement,
            const XC_BYTE*      pcOldExternalIV,
            const XC_UINT4      ui4OldExternalIVLength,
            const XC_BYTE*      pcNewExternalIV,
            const XC_UINT4      ui4NewExternalIVLength,
            const XC_BYTE*      pcInputData,
            const XC_UINT4      ui4InputDataLength,
            const XC_BYTE       bNiInputData,
            XC_BYTE*            pcOutputData,
            XC_UINT4*           pui4OutputDataLength,
            XC_BYTE*            pbNiOutputData,
            const XC_PARAM_EX*  pXCParam,
            const XC_UINT4      ui4XCParamSize,
            stXC_ACTION_RESULT* pstActionResult );
```

The following table lists the various parameters used for the *XCReprotect* function.

Parameter	Description	Data Type
hXCHandle	[in] Handle for the library that has been initialized.	const XC_HANDLE
hSession	[in] Handle for the established session.	const XC_SESSION

ui4EventType	[in] Used to identify whether the call type of the function is First Call or Normal Call.
	
	This parameter can be set to one of the following values:
	
	• **1**: Internally sets the XC event type to *XC_EVENT_FIRST_CALL*. This identifies the call type as First Call. If the call type is First Call, then data is written to the internal cache, which includes:
	  • SessionID
	  • RequestID
	  • DataElement
	  • UserName
	  • ProductID
	  • VendorID
	• **0**: Identifies the call type as Normal Call. If the call type is Normal Call, then the data is only read from the internal cache. Data is written to the internal cache only if it is different from the one that was written during the First Call.
	
	Important: An entry is generated in the *pepserver.log* file only if any new data is written to the internal cache.
	
	Note:
	If logging is not enabled, but the *XC_EVENT_FIRST_CALL* parameter is set, then logs are not generated.
	
	Similarly, if logging is enabled, but *XC_EVENT_FIRST_CALL* is not set, then logs are generated only if any new data is written to the cache.
		const XC_UINT4

Parameter	Description	Data Type
pcPolicyUser	[in] String for the policy user.	const XC_CHAR*
pcOldDataElement	[in] String for the old data element.	const XC_CHAR*
pcNewDataElement	[in] String for the new data element to be used.	const XC_CHAR*
pcOldExternalIV	[in] External IV that was used when data was protected.	const XC_BYTE*
ui4OldExternalIVLength	[in] Number of bytes in *pcOldExternalIV*. The total amount of data to be reprotected should not exceed 256 bytes.	const XC_UINT4
pcNewExternalIV	[in] External IV to use when data is protected with the new data element.	const XC_BYTE*
ui4NewExternalIVLength	[in] Number of bytes in *pcNewExternalIV*. The total amount of data to be reprotected should not exceed 256 bytes.	const XC_UINT4
pcInputData	[in] Buffer containing the data to be re-encrypted.	const XC_BYTE*
ui4InputDataLength	[in] Number of bytes contained in *pcInputData*. The total amount of data to be unprotected should not exceed 2 GB.	const XC_UINT4
bNiInputData	[in] Flag to indicate NULL input.	const XC_BYTE
pcOutputData	[out] Buffer that will hold the resulting re-encrypted data. Needs to be larger than the input buffer.	XC_BYTE*
pui4OutputDataLength	[in/out] On input it should specify the max size of the output data buffer. When the function returns, it will contain the actual number of bytes stored in the output. This size needs to be big enough to hold the resulting data as well as the packed request. Consider having an output buffer that is *XC_BYTES_OVERHEAD* bytes larger than the input buffer for additional overhead. If the value is set to zero as **pui4OutputDataLength = 0;*, then the following value is returned. **pui4OutputDataLength = ui4InputDataLength + XC_BYTES_OVERHEAD;*	XC_UINT4*
pbNiOutputData	[out] Flag indicating that the output is NULL.	XC_BYTE*
pXCParam	[in] Additional information associated with the protection, refer to the *XC_Param_Ex* function.	const XC_PARAM_EX*
ui4XCParamSize	[in] Size of the structure passed in the *pXCParam* parameter.	const XC_UINT4
pstActionResult	[out] Structure containing the result of the operation. For more information refer to the *xcdefinitions.h* file.	stXC_ACTION_RESULT*

2.1.20 XCBulkProtect Function

The *XCBulkProtect* function protects the data in bulk. It will take a list of data items and send them in a single request to the Protegrity AP, which will then process them in a batch. It will then read the response from the Protegrity AP and return a list of data items containing the protected data.

This function returns *XC_SUCCESS* on success.

> **Note:** If you are using Format Preserving Encryption (FPE) and Byte APIs, then ensure that the encoding, which is used to convert the *string* input data to *bytes*, matches the encoding that is selected in the **Plaintext Encoding** drop-down for the required FPE data element.

> **Warning:**
> For Date and DateTime type of data elements, the the *XCBulkProtect* API returns an invalid input data error if the input value falls between the non-existent date range from 05-OCT-1582 to 14-OCT-1582 of the Gregorian Calendar.
>
> For more information about the tokenization and de-tokenization of the cutover dates of the Proleptic Gregorian Calendar, refer to the sections *Date Tokenization for cutover Dates of the Proleptic Gregorian Calendar* and *Datetime Tokenization for Cutover Dates of the Proleptic Gregorian Calendar* in the *Protection Method Reference Guide 9.1.0.0*.

The following is a sample of the *XCBulkProtect* function.

```
XCBulkProtect( const XC_HANDLE        hXCHandle,
               const XC_SESSION       hSession,
               const XC_UINT4         ui4EventType,
               const XC_CHAR*         pcPolicyUser,
               const XC_CHAR*         pcDataElement,
               const XC_BYTE*         pcExternalIV,
               const XC_UINT4         ui4ExternalIVLength,
               const XC_DATA_ITEM_EX* pInDataItems,
               const XC_UINT4         ui4InDataItemCount,
               XC_DATA_ITEM_EX*       pOutDataItems,
               XC_UINT4*              pui4OutDataItemCount,
               XC_INT4*               pi4ErrorIndex,
               const XC_PARAM_EX*     pXCParam,
               const XC_UINT4         ui4XCParamSize,
               stXC_ACTION_RESULT*    pstActionResult );
```

The following table lists the various parameters used for the *XCBulkProtect* function.

Parameter	Description	Data Type
hXCHandle	[in] Handle for the library that has been initialized.	const XC_HANDLE
hSession	[in] Handle for the established session.	const XC_SESSION
ui4EventType	[in] Used to identify whether the call type of the function is First Call or Normal Call. This parameter can be set to one of the following values: • **1**: Internally sets the XC event type to *XC_EVENT_FIRST_CALL*. This identifies the call type as First Call. If the call type is First Call, then data is written to the internal cache, which includes:	const XC_UINT4

Parameter	Description	Data Type
	• SessionID • RequestID • DataElement • UserName • ProductID • VendorID • **0**: Identifies the call type as Normal Call. If the call type is Normal Call, then the data is only read from the internal cache. Data is written to the internal cache only if it is different from the one that was written during the First Call. **Important:** An entry is generated in the *pepserver.log* file only if any new data is written to the internal cache. **Note:** If logging is not enabled, but the *XC_EVENT_FIRST_CALL* parameter is set, then logs are not generated. Similarly, if logging is enabled, but *XC_EVENT_FIRST_CALL* is not set, then logs are generated only if any new data is written to the cache.	
pcPolicyUser	[in] NULL terminated string for the policy user.	const XC_CHAR*
pcDataElement	[in] NULL terminated string for the data element.	const XC_CHAR*
pcExternalIV	[in] Buffer containing data that will be used as external IV.	const XC_BYTE*
ui4ExternalIVLength	[in] Number of bytes provided in the *pcExternalIv* buffer. The total amount of data for bulk protection should not exceed 256 bytes.	const XC_UINT4
pInDataItems	[in] Buffer containing the plaintext to protect.	const XC_DATA_ITEM_EX*
ui4InDataItemCount	[in] Number of items contained in *pInDataItems*. The maximum number of data items should not exceed 1000 elements.	const XC_UINT4
pOutDataItems	[out] Buffer containing the protected data after a successful operation.	XC_DATA_ITEM_EX*
pui4OutDataItemCount	[in/out] On input the capacity of *pOutDataItems* which must be the same as *ui4InDataItemCount*. On return it will contain the number of items placed in *pOutDataItems*.	XC_UINT4*
pi4ErrorIndex	[out] Zero-based index of first item that failed (if any), otherwise it will be set to XC_ERROR_INDEX_NONE.	XC_INT4*

Parameter	Description	Data Type
pXCParam	[in] Additional parameters.	const XC_PARAM_EX*
ui4XCParamSize	[in] Size of the additional parameters.	const XC_UINT4
pstActionResult	[out] Structure containing the result of the operation. For more information refer to the *xcdefinitions.h* file.	stXC_ACTION_RESULT*

2.1.21 XCBulkUnProtect Function

The *XCBulkUnProtect* function decrypts the data in bulk. It will take a list of data items and send them in a single request to the Protegrity AP, which will then process them in a batch. It will then read the response from the Protegrity AP and return a list of data items containing the plain data.

This function returns *XC_SUCCESS* on success.

> **Note:** If you are using Format Preserving Encryption (FPE) and Byte APIs, then ensure that the encoding selected in the **Plaintext Encoding** drop-down for the required FPE data element matches the encoding that is used to convert the protected *byte* input data to *string*.

The following is a sample of the *XCBulkUnProtect* function.

```
XCBulkUnprotect( const XC_HANDLE          hXCHandle,
                 const XC_SESSION         hSession,
                 const XC_UINT4           ui4EventType,
                 const XC_CHAR*           pcPolicyUser,
                 const XC_CHAR*           pcDataElement,
                 const XC_BYTE*           pcExternalIV,
                 const XC_UINT4           ui4ExternalIVLength,
                 const XC_DATA_ITEM_EX*   pInDataItems,
                 const XC_UINT4           ui4InDataItemCount,
                 XC_DATA_ITEM_EX*         pOutDataItems,
                 XC_UINT4*                pui4OutDataItemCount,
                 XC_INT4*                 pi4ErrorIndex,
                 const XC_PARAM_EX*       pXCParam,
                 const XC_UINT4           ui4XCParamSize,
                 stXC_ACTION_RESULT*      pstActionResult );
```

The following table lists the various parameters used for the *XCBulkUnProtect* function.

Parameter	Description	Data Type
hXCHandle	[in] Handle for the library that has been initialized.	const XC_HANDLE
hSession	[in] Handle for the established session.	const XC_SESSION
ui4EventType	[in] Used to identify whether the call type of the function is First Call or Normal Call. This parameter can be set to one of the following values: • **1**: Internally sets the XC event type to *XC_EVENT_FIRST_CALL*. This identifies the call type as First Call. If the call type is First Call, then data is written to the internal cache, which includes:	const XC_UINT4

Parameter	Description	Data Type
	• SessionID • RequestID • DataElement • UserName • ProductID • VendorID • **0**: Identifies the call type as Normal Call. If the call type is Normal Call, then the data is only read from the internal cache. Data is written to the internal cache only if it is different from the one that was written during the First Call. **Important:** An entry is generated in the *pepserver.log* file only if any new data is written to the internal cache. **Note:** If logging is not enabled, but the *XC_EVENT_FIRST_CALL* parameter is set, then logs are not generated. Similarly, if logging is enabled, but *XC_EVENT_FIRST_CALL* is not set, then logs are generated only if any new data is written to the cache.	
pcPolicyUser	[in] NULL terminated string for the policy user.	const XC_CHAR*
pcDataElement	[in] NULL terminated string for the data element.	const XC_CHAR*
pcExternalIV	[in] Buffer containing data that will be used as external IV.	const XC_BYTE*
ui4ExternalIVLength	[in] Number of bytes provided in the *pcExternalIv* buffer. The total amount of data for bulk protection should not exceed 256 bytes.	const XC_UINT4
pInDataItems	[in] Buffer containing the plain data to protect.	const XC_DATA_ITEM_EX*
ui4InDataItemCount	[in] Number of items contained in *pInDataItems*. The maximum number of data items should not exceed 1000 elements.	const XC_UINT4
pOutDataItems	[out] Buffer containing the protected data after a successful operation.	XC_DATA_ITEM_EX*
pui4OutDataItemCount	[in/out] On input the capacity of *pOutDataItems* which must be the same as *ui4InDataItemCount*. On return it will contain the number of items placed in *pOutDataItems*.	XC_UINT4*
pi4ErrorIndex	[out] Zero-based index of first item that failed (if any), otherwise it will be set to *XC_ERROR_INDEX_NONE*.	XC_INT4*

Parameter	Description	Data Type
pXCParam	[in] Additional parameters.	const XC_PARAM_EX*
ui4XCParamSize	[in] Size of the additional parameters.	const XC_UINT4
pstActionResult	[out] Structure containing the result of the operation. For more information refer to the *xcdefinitions.h* file.	stXC_ACTION_RESULT*

2.1.22 XCCheckAccess Function

The *XCCheckAccess* function checks access permission to a specific Data Element.

This function returns *XC_SUCCESS* on success and returns *XC_ACCESS_DENIED* if no access permitted.

The following is a sample for the *XCCheckAccess* function.

```
XCCheckAccess( const XC_HANDLE  hXCHandle,
               const XC_SESSION hSession,
               const XC_UINT4   ui4FirstCall,
               const XC_CHAR*   pcPolicyUser,
               const XC_CHAR*   pcDataElement,
               const XC_BYTE    cAccessMask );
```

The following table lists the various parameters used for the *XCCheckAccess* function.

Parameter	Description	Data Type
hXCHandle	[in] Handle for the library that has been initialized.	const XC_HANDLE
hSession	[in] Handle for the established session.	const XC_SESSION
ui4FirstCall	[in] Flag for first call, use defined types XC_TRUE, XC_FALSE.	const XC_UINT4
pcPolicyUser	[in] String for the policy user.	const XC_CHAR *
pcDataElement	[in] String for the data element to be used.	const XC_CHAR *
cAccessMask	[in] Byte Mask of access right query.	const XC_BYTE

2.1.23 XCCheckAccessEx Function

The *XCCheckAccessEx* function checks access permission to a specific Data Element.

This function returns *XC_SUCCESS* on success and returns *XC_ACCESS_DENIED* if no access permitted.

The following is a sample for the *XCCheckAccessEx* function.

```
XCCheckAccessEx( const XC_HANDLE     hXCHandle,
                 const XC_SESSION    hSession,
                 const XC_UINT4      ui4EventType,
                 const XC_CHAR*      pcPolicyUser,
```

```
                    const XC_CHAR*      pcDataElement,
                    const XC_BYTE       cAccessMask,
                    const XC_CHAR*      pszVendorType,
                    const XC_PARAM_EX*  pXCParam,
                    const XC_UINT4      ui4XCParamSize );
```

The following table lists the various parameters used for the *XCCheckAccessEx* function.

Parameter	Description	Data Type
hXCHandle	[in] Handle for the library that has been initialized.	const XC_HANDLE
hSession	[in] Handle for the established session.	const XC_SESSION
ui4EventType	[in] Flag for first call, use defined types XC_TRUE, XC_FALSE.	const XC_UINT4
pcPolicyUser	[in] String for the policy user.	const XC_CHAR *
pcDataElement	[in] String for the data element to be used.	const XC_CHAR *
cAccessMask	[in] Byte Mask of access right query.	const XC_BYTE *
pszVendorType	[in] Type of vendor.	const XC_CHAR*
pXCParam	[in] Additional parameters.	const XC_PARAM_EX*
ui4XCParamSize	[in] Size of the additional parameters.	const XC_UINT4

2.1.24 XCGetDefaultDataElement Function

The *XCGetDefaultDataElement* function returns the default data element for a specific policy.

This function returns one of the following responses:

- XC_SUCCESS on success
- XC_NOT_FOUND if a policy with the specified name does not exist
- XC_NOT_DEFINED if no default data element has been configured in the policy
- XC_BUFFER_TOO_SMALL if the pui4DataElementLength is not big enough.

The following is a sample of the *XCGetDefaultDataElement* function.

```
XCGetDefaultDataElement( const XC_HANDLE  hXCHandle,
                         const XC_SESSION hSession,
                         const XC_CHAR*   pcPolicyName,
                         XC_CHAR*         pcDataElement,
                         XC_UINT4*        pui4DataElementLength );
```

The following table lists the various parameters used for the *XCGetDefaultDataElement* function.

Parameter	Description	Data Type
hCHandle	[in] Handle for the library that has been initialized.	const C_HANDLE

Parameter	Description	Data Type
hSession	[in] Handle for the established session.	const XC_SESSION
pcPolicyName	[in] String for the policy name for which to get the default data element.	const C_CHAR*
pcDataElement	[out] Buffer to hold the default data element name.	C_CHAR*
pui4DataElementLength	[in/out] On input, it should specify the max length of the pcDataElement buffer.	C_UINT4*

2.1.25 XCGetErrorDescription Function

The *XCGetErrorDescription* function gets error description when a previous call to a Protegrity AP function fails. To get the required size of the message buffer, you can call the function with *pszMessage* set to XC_NULL. The required buffer size will then be returned in parameter *pui4MessageLength*, and the return code set to *XC_BUFFER_TOO_SMALL*.

This function returns *XC_SUCCESS* if successful and it returns *XC_BUFFER_TOO_SMALL* if the pui4MessageLength is not big enough.

The following is a sample of the *XCGetErrorDescription* function.

```
XCGetErrorDescription( const XC_HANDLE  hXCHandle,
                       const XC_SESSION hSession,
                       XC_CHAR*         pszMessage,
                       XC_UINT4*        pui4MessageLength );
```

The following table lists the various parameters used for the *XCGetErrorDescription* function.

Parameter	Description	Data Type
hXCHandle	[in] Handle for the library that has been initialized.	const XC_HANDLE
hSession	[in] Handle for the session or NULL if no session exists (for example, when *XCOpenSession* failed).	const XC_SESSION
pszMessage	[in] The null terminated message string.	XC_CHAR*
pui4MessageLength	[in/out] On input, it should specify the max length of the pszMessage buffer. On return, it will contain the length of the message (not counting null).	XC_UINT4*

2.1.26 XCGetCurrentKeyID Function

The *XCGetCurrentKeyID* function gets the current KeyID for a data element.

This function returns *XC_SUCCESS* on success.

The following is a sample of the *XCGetCurrentKeyID* function.

```
XCGetCurrentKeyID( const XC_HANDLE   hXCHandle,
                   const XC_SESSION  hSession,
                   const XC_CHAR*    pcDataElement,
                   XC_UINT4*         pui4OutKeyID );
```

The following table lists the various parameters used for the *XCGetCurrentKeyID* function.

Parameter	Description	Data Type
hXCHandle	[in] Handle for the library that has been initialized.	const XC_HANDLE
hSession	[in] Handle for the established session.	const XC_SESSION
pcDataElement	[in] NULL terminated string for the data element.	const XC_CHAR*
pui4OutKeyID	[out] The KeyID is returned.	XC_UINT4*

2.1.27 XCGetKeyID Function

The *XCGetKeyID* function gets the KeyID from data protected with the data element *pcDataElement*.

This function returns *XC_SUCCESS* on success.

> **Note:** The KeyID is derived from the initial 2 bytes of the protected data.

The following is a sample of the *XCGetKeyID* function.

```
XCGetKeyID( const XC_HANDLE   hXCHandle,
            const XC_SESSION  hSession,
            const XC_CHAR*    pcDataElement,
            const XC_BYTE*    pcInputData,
            const XC_UINT4    ui4InputDataLength,
            XC_UINT4*         pui4OutKeyID );
```

The following table lists the various parameters used for the *XCGetKeyID* function.

Parameter	Description	Data Type
hXCHandle	[in] Handle for the library that has been initialized.	const XC_HANDLE
hSession	[in] Handle for the established session.	const XC_SESSION
pcDataElement	[in] NULL terminated string for the data element.	const XC_CHAR*
pcInputData	[in] Buffer containing the data to decrypt.	const XC_BYTE*
ui4InputDataLength	[in] Number of bytes contained in *pcInputData*. The total amount of data should not exceed 2 GB.	const XC_UINT4

Parameter	Description	Data Type
pui4OutKeyID	[out] The KeyID is returned.	XC_UINT4 *

2.2 Application Protector (AP) Golang APIs

A session must be created to run the AP Go. The AP Go accesses the information on the Trusted Application from the policy stored in the memory. If the application is trusted, then the protect or unprotect or reprotect method is called, one or many times, depending on the data. You can flush the audits after the operation is complete.

Note: The AP Go APIs can be invoked by a valid *Policy User* or a *Trusted Application* user.

Note:
When a short running application has completed its execution (in less than a second), the audit logs will not be seen. In such cases, the*FlushAudits()* API needs to be invoked.

It is recommended to invoke the *FlushAudits()* API at the point where the application exits.

For more information about flushAudits, refer to the section *FlushAudits API*.

The following diagram represents the basic flow of a session.

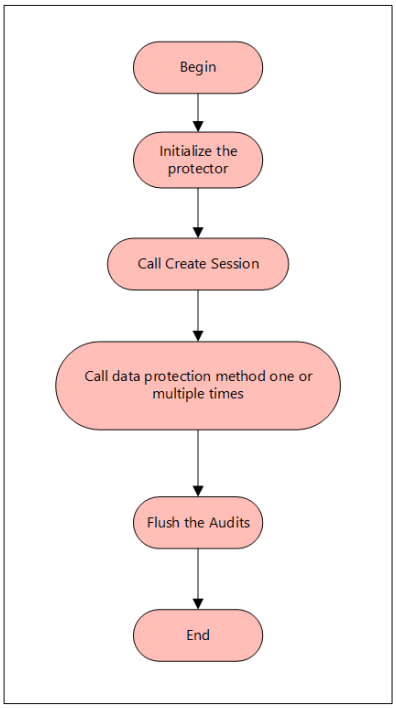

Figure 2-2: Flowchart for AP Go

The following sections provide detailed information of the various APIs used by the Protegrity Application Protector Golang (Go).

Note:

The user must call the *defer terminate()* method after initializing the AP Go package to see the logs for applications running for less than a second.

Note:

For the AP Go APIs, the *apgo.UsingExtIV* is an optional parameter. It is a buffer containing data that is used as an initialization vector, and accepts input in byte format. When the external IV is empty, its value is ignored.

Note:

The *apgo.UsingExtTweak* parameter is an optional parameter that is used only for the FPE data elements. It is a buffer containing data that is used as an external Tweak, and accepts input in byte format. When the external Tweak is empty, its value is ignored.

2.2.1 Supported Data Types for AP Go

This section lists the data types supported by the AP Go.

The AP Go supports the following data types:

- String
- Integer
- Short
- Long
- Float
- Double
- Bytes
- Date

2.2.2 GetVersion API

The *GetVersion* API returns the version of the Application Protector Go in use.

Note:

You do not need to call the *Init* API before invoking the *GetVersion* API.

Note:

You do not need to create a session for invoking the *GetVersion* API.

func GetVersion() string

Returns

String: Product version of the installed AP Go.

2.2.3 GetVersionEx API

The *GetVersionEx* API returns the extended version of the Application Protector Go in use. The extended version consists of product version number and core version number. Core version number can be communicated to the Protegrity Support while troubleshooting issues related to AP Go.

> **Note:**
> You do not need to call the *Init* API before invoking the *GetVersion* API.

> **Note:**
> You do not need to create a session for invoking the *GetVersion* API.

func GetVersionEx() string

Returns

`String`: Extended product version of the installed AP Go.

2.2.4 Init API

The *Init* API initializes the Application Protector Go. This should be performed only once in the lifecycle of each application that uses the AP Go. The protection operations can be performed only on the successful initialization of the AP Go. This API returns a terminate function, which should be called at the end of the life of the application.

func Init(opts ...InitOptions) error

Optional Parameters

`WithCommID`: Helper function of type *InitOption* to configure the *Communication ID* used by the PEP server. This value must match the value specified by the *Communication ID* parameter in the PEP server configuration file, *pepserver.cfg*. This parameter is optional.

> **Note:** Ensure that the *WithCommID* parameter value is equal to the *Communication ID* parameter value in the *pepserver.cfg* file.

Returns

`func()`: Returns the *terminate* function to be called at the end of the application lifecycle. The *terminate* function releases the resources acquired by the AP Go library. After this function is invoked, you cannot invoke any other function in the AP Go.

`Error`: Returns an error if there is an issue while creating the session.

The *InitOptions* parameter is a self-referencing function in Golang which is used for passing the *Communication ID* if required. This is optional and the default value is *0*.

For example, if you want to configure the *Communication ID* value as 120, then you can invoke the *Init* API listed in the following sample:

```
terminate, err := apgo.Init(apgo.WithCommID(120))
       defer terminate()
```

2.2.5 NewSession API

The *NewSession* API creates a session. Sessions that are created using this API, automatically time out after the session timeout value has been reached. The default session timeout value is *15* minutes. However, you can also pass the session timeout value as a parameter to this API.

func NewSession(usr string, options ...SessionOptions) (*Session, error)

Parameters

policyUser: User name defined in the policy, as a *string* value.

SessionOptions: Passing the optional session timeout value.

Returns

Session: Object of the *Session* type.

Error: Returns an error if the session creation fails.

You can use a helper function to configure the timeout value. The following sample is a signature for the helper function used for configuring the timeout value.

```
func Timeout(t int) SessionOptions
```

For example, if you want to change the timeout value from 15 minutes to 20 minutes, then you can invoke the *NewSession* API listed in the following sample.

```
// With default timeout
session, err := apgo.NewSession("user1")

// With explicit timeout of 20 mins
session, err := apgo.NewSession("user1", apgo.Timeout(20))
```

2.2.6 CheckAccess API

The *CheckAccess* API checks the access permission status of the user for a specified data element and the access type, for example, protect, unprotect, or reprotect.

func (s *Session) CheckAccess(dataElement string, accessType byte) (bool, error)

Parameters

dataElement: String containing the data element name defined in the policy.

accessType: Type of the access permission of the user for the specified data element. You can specify a value for this parameter from the constants, such as, protect, unprotect, or reportect.

Returns

bool: Returns *true* if the user has access to the data element and *false* if the user does not have access to the data element.

error: Error message is returned if the *CheckAccess* API fails.

Example:

```
if _, err := apgo.Init(); err != nil {
  log.Println(err)
}
apSession, err := apgo.NewSession("user")
ok, err := apSession.CheckAccess(de, apgo.PROTECT_ACCESS)
ok, err = apSession.CheckAccess(de, apgo.UNPROTECT_ACCESS)
ok, err = apSession.CheckAccess(de, apgo.REPROTECT_ACCESS)
```

2.2.7 GetCurrentKeyIdForDataElement API

The *GetCurrentKeyIdForDataElement* API returns the key ID of the data element provided as an input parameter.

func (s *Session) GetCurrentKeyIdForDataElement(dataElement string) (int, error)

Parameters

dataElement: The data element name for which the key ID needs to be returned.

Returns

`int`: Returns the current key ID in *int* format data.

`error`: Error message if the *GetCurrentKeyIdForDataElement* API fails.

Example:

The following is a sample of the *GetCurrentKeyIdForDataElement* API.

```
if terminate, err := apgo.Init(); err != nil {
    panic(err)
}
defer terminate()
apSession, err := apgo.NewSession("user")
getkeyid, err := apSession.GetCurrentKeyIdForDataElement(dataElement)
```

2.2.8 ExtractKeyIdFromData API

The *ExtractKeyIdFromData* API extracts key ID from input data when the data element is provided as an input parameter.

func (s *Session) ExtractKeyIdFromData(data []byte, dataElement string) (int, error)

Parameters

`data`: The data for which the key ID needs to be returned.

`dataElement`: The data element name using which the Key ID for the data is returned.

Returns

`int`: Returns the key ID in *int* format data.

`error`: Error message if the *ExtractKeyIdFromData* API fails.

Example

The following is a sample for the *ExtractKeyIdFromData* API.

```
if _, err := apgo.Init(); err != nil {
log.Println(err)
}
apSession, err := apgo.NewSession("user")
output, rc, err := apSession.EncryptStr(data, dataElement)

getkeyid, err := apSession.ExtractKeyIdFromData(output, dataElement)
```

2.2.9 GetDefaultDataElement API

The *GetDefaultDataElement* API returns the default data element for the policy name provided as an input parameter. A data element becomes default for a policy if you select it as default during policy creation.

func (s *Session) GetDefaultDataElement(policyName string) (string, error)

Parameters

`policyName`: The policy name for which the default data element needs to be returned.

Returns

`string`: Returns the default data element name in *string* format data.

`error`: Error message if the *GetDefaultDataElement* API fails.

Example

The following is a sample for the *GetDefaultDataElement* API.

```
if _, err := apgo.Init(); err != nil {
log.Println(err)
}
apSession, err := apgo.NewSession("user")
defaultDE, err := apSession.GetDefaultDataElement("PolicyName")
```

2.2.10 FlushAudits API

The *FlushAudits* API is used for flushing the audit logs at any given point within the application. This API is required for a short running process that lasts less for than a second, to get the audit logs. It is recommended to invoke it at the point where the application exits.

func (s *Session) FlushAudits() error

Parameters

None

Returns

`error`: Error message if the *FlushAudits* API fails.

Example

In the following example, the *FlushAudits* API is used to flush the audit logs.

```
if _, err := apgo.Init(); err != nil {
log.Println(err)
}
session, err := apgo.NewSession("user")
output, rc, err := session.ProtectStr(inputData, dataElement)
session.FlushAudits()
```

2.2.11 ProtectBytes API

The *ProtectBytes* API protects the data in *byte* format using data type preservation or a No Encryption data element.

func (s *Session) ProtectBytes(data []byte, dataElement string, options ...ProtectionOptions) ([]byte, int16, error)

Parameters

`data`: Input containing the data to be protected in *byte* format.

`dataElement`: Input containing the data element name defined in the policy in *string* format.

`options`:

- `apgo.UsingExtIV`: External IV is an optional parameter. It is a buffer containing data that is used as an initialization vector and accepts input in byte format. When the External IV is empty, its value is ignored.
- `apgo.UsingExtTweak`: External Tweak is an optional parameter that is used only for the FPE data elements. It is a buffer containing data that is used as an External Tweak and accepts input in byte format. When the External Tweak is empty, its value is ignored.

Returns

`byte`: Returns the protected data in *byte* format.

`int16`: Returns the return code of the protect operation in *int16* format.

`error`: Error message if the protect operation fails.

Example

The following is a sample for the *ProtectBytes* API.

```
output, rc, err := session.ProtectBytes(input, dataElement,
apgo.UsingExtIV([]byte(externalIv)), apgo.UsingExtTweak([]byte(externaltweak)))
```

2.2.12 ProtectStr API

The *ProtectStr* API protects the data in *string* format using data type preservation or the No Encryption data element.

func (s *Session) ProtectStr(data string, dataElement string, options ...ProtectionOptions) (string, int16, error)

Parameters

data: Input containing the data to be protected in *string* format.

dataElement: Input containing the data element name defined in the policy in *string* format.

options:

- ***apgo.UsingExtIV***: External IV is an optional parameter. It is a buffer containing data that is used as an initialization vector and accepts input in byte format. When the External IV is empty, its value is ignored.
- ***apgo.UsingExtTweak***: External Tweak is an optional parameter that is used only for the FPE data elements. It is a buffer containing data that is used as an External Tweak and accepts input in byte format. When the External Tweak is empty, its value is ignored.

Returns

string: Returns the protected data in *string* format.

int16: Returns the return code of the protect operation in *int16* format.

error: Error message if the protect operation fails.

Example

The following is a sample for the *ProtectStr* API.

```
output, rc, err := session.ProtectStr(input, dataElement,
apgo.UsingExtIV([]byte(externalIv)), apgo.UsingExtTweak([]byte(externaltweak)))
```

2.2.13 ProtectShort API

The *ProtectShort* API protects data in the *short* format using data type preservation or a No Encryption data element.

func (s *Session) ProtectShort(data int16, dataElement string, options ...ProtectionOptions) (int16, int16, error)

Parameters

data: Input containing the data to be protected in the *int16* format.

dataElement: Input containing the data element name defined in the policy in *string* format.

options:

- ***apgo.UsingExtIV***: External IV is an optional parameter. It is a buffer containing data that is used as an initialization vector and accepts input in byte format. When the External IV is empty, its value is ignored.

Returns

int16: Returns the protected data in *int16* format.

int16: Returns the return code of the protect operation in *int16* format.

error: Error message if the protect operation fails.

Example

The following is a sample for the *ProtectShort* API.

```
output, rc, err := session.ProtectShort(input, dataElement,
apgo.UsingExtIV([]byte(externalIv)))
```

2.2.14 ProtectInt API

The *ProtectInt* API protects the data in the *integer* format using data type preservation or a No Encryption data element.

func (s *Session) ProtectInt(data int32, dataElement string, options ...ProtectionOptions) (int32, int16, error)

Parameters

`data`: Input containing the data to be protected in *int32* format.

`dataElement`: Input containing the data element name defined in the policy in *string* format.

`options`:

- `apgo.UsingExtIV`: External IV is an optional parameter. It is a buffer containing data that is used as an initialization vector and accepts input in byte format. When the External IV is empty, its value is ignored.

Returns

`int32`: Returns the protected data in *int32* format.

`int16`: Returns the return code of the protect operation in *int16* format.

`error`: Error message if the protect operation fails.

Example

The following is a sample for the *ProtectInt* API.

```
output, rc, err := session.ProtectInt(input, dataElement,
apgo.UsingExtIV([]byte(externalIv)))
```

2.2.15 ProtectLong API

The *ProtectLong* API protects the data in the *long* format using data type preservation or a No Encryption data element.

func (s *Session) ProtectLong(data int64, dataElement string, options ...ProtectionOptions) (int64, int16, error)

Parameters

`data`: Input containing the data to be protected in *int64* format.

`dataElement`: Input containing the data element name defined in the policy in *string* format.

`options`:

- `apgo.UsingExtIV`: External IV is an optional parameter. It is a buffer containing data that is used as an initialization vector and accepts input in byte format. When the External IV is empty, its value is ignored.

Returns

`int64`: Returns the protected data in *int64* format.

`int16`: Returns the return code of the protect operation in *int16* format.

`error`: Error message if the protect operation fails.

Example

The following is a sample for the *ProtectLong* API.

```
output, rc, err := session.ProtectLong(input, dataElement,
apgo.UsingExtIV([]byte(externalIv)))
```

2.2.16 ProtectFloat API

The *ProtectFloat* API protects the data in the *float* format using a No Encryption data element.

func (s *Session) ProtectFloat(data float32, dataElement string) (float32, int16, error)

Parameters

data: Input containing the data to be protected in *float32* format.

dataElement: Input containing the data element name defined in the policy in *string* format.

Returns

float32: Returns the protected data in *float32* format.

int16: Returns the return code of the protect operation in *int16* format.

error: Error message if the protect operation fails.

Example

The following is a sample for the *ProtectFloat* API.

```
output, rc, err := session.ProtectFloat(input, dataElement,
apgo.UsingExtIV([]byte(externalIv)))
```

2.2.17 ProtectDouble API

The *ProtectDouble* API protects the data in the *double* format using a No Encryption data element.

func (s *Session) ProtectDouble(data float64, dataElement string) (float64, int16, error)

Parameters

data: Input containing the data to be protected in *float64* format.

dataElement: Input containing the data element name defined in the policy in *string* format.

Returns

float64: Returns the protected data in *float64* format.

int16: Returns the return code of the protect operation in *int16* format.

error: Error message if the protect operation fails.

Example

The following is a sample for the *ProtectDouble* API.

```
output, rc, err := session.ProtectDouble(input, dataElement,
apgo.UsingExtIV([]byte(externalIv)))
```

2.2.18 EncryptStr API

The *EncryptStr* API encrypts the data in the *string* format using an encryption data element.

func (s *Session) EncryptStr(data string, dataElement string, options ...ProtectionOptions) ([]byte, int16, error)

Parameters

data: Input containing the data to be protected in *string* format.

dataElement: Input containing the data element name defined in the policy in *string* format.

options:

- **`apgo.UsingExtIV`**: External IV is an optional parameter. It is a buffer containing data that is used as an initialization vector and accepts input in byte format. When the External IV is empty, its value is ignored.

  > **Note:**
  > Encryption data elements do not support external IV.

- **`apgo.UsingExtTweak`**: External Tweak is an optional parameter that is used only for the FPE data elements. It is a buffer containing data that is used as an External Tweak, and accepts input in byte format. When the External Tweak is empty, its value is ignored.

Returns

`byte`: Returns the encrypted data in *byte* format.

`int16`: Returns the return code of the encrypt operation in *int16* format.

`error`: Error message if the encrypt operation fails.

Example

The following is a sample for the *EncryptStr* API.

```
output, rc, err := session.EncryptStr(input, dataElement,
apgo.UsingExtIV([]byte(externalIv)), apgo.UsingExtTweak([]byte(externaltweak)
```

2.2.19 EncryptShort API

The *EncryptShort* API encrypts the data in the *short* format using an encryption data element.

func (s *Session) EncryptShort(data int16, dataElement string, options ...ProtectionOptions) ([]byte, int16, error)

Parameters

`data`: Input containing the data to be protected in *int16* format.

`dataElement`: Input containing the data element name defined in the policy in *string* format.

`options`:

- **`apgo.UsingExtIV`**: External IV is an optional parameter. It is a buffer containing data that is used as an initialization vector and accepts input in byte format. When the External IV is empty, its value is ignored.

  > **Note:**
  > Encryption data elements do not support external IV.

Returns

`byte`: Returns the encrypted data in *byte* format.

`int16`: Returns the return code of the encrypt operation in *int16* format.

`error`: Error message if the encrypt operation fails.

Example

The following is a sample for the *EncryptShort* API.

```
output, rc, err := session.EncryptShort(input, dataElement,
apgo.UsingExtIV([]byte(externalIv)))
```

2.2.20 EncryptInt API

The *EncryptInt* API encrypts the data in the *integer* format using an encryption data element.

func (s *Session) EncryptInt(data int32, dataElement string, options ...ProtectionOptions) ([]byte, int16, error)

Parameters

data: Input containing the data to be protected in *int32* format.

dataElement: Input containing the data element name defined in the policy in *string* format.

options:

- ***apgo.UsingExtIV***: External IV is an optional parameter. It is a buffer containing data that is used as an initialization vector and accepts input in byte format. When the External IV is empty, its value is ignored.

> **Note:**
> Encryption data elements do not support external IV.

Returns

byte: Returns the encrypted data in *byte* format.

int16: Returns the return code of the encrypt operation in *int16* format.

error: Error message if the encrypt operation fails.

Example

The following is a sample for the *EncryptInt* API.

```
output, rc, err := session.EncryptInt(input, dataElement,
apgo.UsingExtIV([]byte(externalIv)))
```

2.2.21 EncryptLong API

The *EncryptLong* API encrypts the data in the *long* format using an encryption data element.

func (s *Session) EncryptLong(data int64, dataElement string, options ...ProtectionOptions) ([]byte, int16, error)

Parameters

data: Input containing the data to be protected in *int64* format.

dataElement: Input containing the data element name defined in the policy in *string* format.

options:

- ***apgo.UsingExtIV***: External IV is an optional parameter. It is a buffer containing data that is used as an initialization vector and accepts input in byte format. When the External IV is empty, its value is ignored.

> **Note:**
> Encryption data elements do not support external IV.

Returns

byte: Returns the encrypted data in *byte* format.

int16: Returns the return code of the encrypt operation in *int16* format.

error: Error message if the encrypt operation fails.

Example

The following is a sample for the *EncryptLong* API.

```
output, rc, err := session.EncryptLong(input, dataElement,
apgo.UsingExtIV([]byte(externalIv)))
```

2.2.22 EncryptFloat API

The *EncryptFloat* API encrypts the data in the *float* format using an encryption data element.

func (s *Session) EncryptFloat(data float32, dataElement string) ([]byte, int16, error)

Parameters

data: Input containing the data to be protected in *float32* format.

dataElement: Input containing the data element name defined in the policy in *string* format.

Returns

byte: Returns the encrypted data in *byte* format.

int16: Returns the return code of the encrypt operation in *int16* format.

error: Error message if the encrypt operation fails.

Example

The following is a sample for the *EncryptFloat* API.

```
output, rc, err := session.EncryptFloat(input, dataElement,
apgo.UsingExtIV([]byte(externalIv)))
```

2.2.23 EncryptDouble API

The *EncryptDouble* API encrypts the data in the *double* format using an encryption data element.

func (s *Session) EncryptDouble(data float64, dataElement string) ([]byte, int16, error)

Parameters

data: Input containing the data to be protected in *float64* format.

dataElement: Input containing the data element name defined in the policy in *string* format.

Returns

byte: Returns the encrypted data in *byte* format.

int16: Returns the return code of the encrypt operation in *int16* format.

error: Error message if the encrypt operation fails.

Example

The following is a sample for the *EncryptDouble* API.

```
output, rc, err := session.EncryptDouble(input, dataElement,
apgo.UsingExtIV([]byte(externalIv)))
```

2.2.24 UnprotectBytes API

The *UnprotectBytes* API unprotects the data in the *byte* format using data type preservation or a No Encryption data element.

func (s *Session) UnprotectBytes(data []byte, dataElement string, options ...ProtectionOptions) ([]byte, int16, error)

Parameters

data: Input containing the data to be unprotected in *byte* format.

dataElement: Input containing the data element defined in the policy in *string* format.

options:

- ***apgo.UsingExtIV***: External IV is an optional parameter. It is a buffer containing data that is used as an initialization vector and accepts input in byte format. When the External IV is empty, its value is ignored.

- **`apgo.UsingExtTweak`**: External Tweak is an optional parameter that is used only for the FPE data elements. It is a buffer containing data that is used as an External Tweak and accepts input in byte format. When the External Tweak is empty, its value is ignored.

Returns

`byte`: Returns the unprotected data in *byte* format

`int16`: Returns the return code of the unprotect operation in *int16* format.

`error`: Error message if the unprotect operation fails.

Example

The following is a sample for the *UnprotectBytes* API.

```
output, rc, err := session.UnprotectBytes(input, dataElement,
apgo.UsingExtIV([]byte(externalIv)), apgo.UsingExtTweak([]byte(externaltweak)))
```

2.2.25 UnprotectStr API

The *UnprotectStr* API unprotects the data in the *string* format using data type preservation or a No Encryption data element.

func (s *Session) UnprotectStr(data string, dataElement string, options ...ProtectionOptions) (string, int16, error)

Parameters

`data`: Input containing the data to be unprotected in *string* format.

`dataElement`: Input containing the data element defined in the policy in *string* format.

`options`:

- **`apgo.UsingExtIV`**: External IV is an optional parameter. It is a buffer containing data that is used as an initialization vector and accepts input in byte format. When the External IV is empty, its value is ignored.
- **`apgo.UsingExtTweak`**: External Tweak is an optional parameter that is used only for the FPE data elements. It is a buffer containing data that is used as an External Tweak and accepts input in byte format. When the External Tweak is empty, its value is ignored.

Returns

`string`: Returns the unprotected data in *string* format.

`int16`: Returns the return code of the unprotect operation in *int16* format.

`error`: Error message if the unprotect operation fails.

Example

The following is a sample for the *UnprotectStr* API.

```
output, rc, err := session.UnprotectStr(input, dataElement,
apgo.UsingExtIV([]byte(externalIv)), apgo.UsingExtTweak([]byte(externaltweak)))
```

2.2.26 UnprotectShort API

The *UnprotectShort* API unprotects the data in the *short* format using data type preservation or a No Encryption data element.

func (s *Session) UnprotectShort(data int16, dataElement string, options ...ProtectionOptions) (int16, int16, error)

Parameters

`data`: Input containing the data to be unprotected in *int16* format.

`dataElement`: Input containing the data element defined in the policy in *string* format.

`options`:

- **`apgo.UsingExtIV`**: External IV is an optional parameter. It is a buffer containing data that is used as an initialization vector and accepts input in byte format. When the External IV is empty, its value is ignored.

Returns

int16: Returns the unprotected data in *int16* format.

int16: Returns the return code of the unprotect operation in *int16* format.

error: Error message if the unprotect operation fails.

Example

The following is a sample for the *UnprotectShort* API.

```
output, rc, err := session.UnprotectShort(input, dataElement,
apgo.UsingExtIV([]byte(externalIv)))
```

2.2.27 UnprotectInt API

The *UnprotectInt* API unprotects the data in the *integer* format using data type preservation or a No Encryption data element.

func (s *Session) UnprotectInt(data int32, dataElement string, options ...ProtectionOptions) (int32, int16, error)

Parameters

data: Input containing the data to be unprotected in *int32* format.

dataElement: Input containing the data element defined in the policy in *string* format.

options:

- ***apgo.UsingExtIV***: External IV is an optional parameter. It is a buffer containing data that is used as an initialization vector and accepts input in byte format. When the External IV is empty, its value is ignored.

Returns

int32: Returns the unprotected data in *int32* format.

int16: Returns the return code of the unprotect operation in *int16* format.

error: Error message if the unprotect operation fails.

Example

The following is a sample for the *UnprotectInt* API.

```
output, rc, err := session.UnprotectInt(input, dataElement,
apgo.UsingExtIV([]byte(externalIv)))
```

2.2.28 UnprotectLong API

The *UnprotectLong* API unprotects the data in the *long* format using data type preservation or a No Encryption data element.

func (s *Session) UnprotectLong(data int64, dataElement string, options ...ProtectionOptions) (int64, int16, error)

Parameters

data: Input containing the data to be unprotected in *int64* format.

dataElement: Input containing the data element defined in the policy in *string* format.

options:

- ***apgo.UsingExtIV***: External IV is an optional parameter. It is a buffer containing data that is used as an initialization vector and accepts input in byte format. When the External IV is empty, its value is ignored.

Returns

int64: Returns the unprotected data in *int64* format.

int16: Returns the return code of the unprotect operation in *int16* format.

error: Error message if the unprotect operation fails.

Example

The following is a sample for the *UnprotectLong* API.

```
output, rc, err := session.UnprotectLong(input, dataElement,
apgo.UsingExtIV([]byte(externalIv)))
```

2.2.29 UnprotectFloat API

The *UnprotectFloat* API unprotects the data in the *float* format using data type preservation or a No Encryption data element.

func (s *Session) UnprotectFloat(data float32, dataElement string, options ...ProtectionOptions) (float32, int16, error)

Parameters

data: Input containing the data to be unprotected in *float32* format.

dataElement: Input containing the data element defined in the policy in *string* format.

options:

- ***apgo.UsingExtIV***: External IV is an optional parameter. It is a buffer containing data that is used as an initialization vector and accepts input in byte format. When the External IV is empty, its value is ignored.

Returns

float32: Returns the unprotected data in *float32* format.

int16: Returns the return code of the unprotect operation in *int16* format.

error: Error message if the unprotect operation fails.

Example

The following is a sample for the *UnprotectFloat* API.

```
output, rc, err := session.UnprotectFloat(input, dataElement,
apgo.UsingExtIV([]byte(externalIv)))
```

2.2.30 UnprotectDouble API

The *UnprotectDouble* API unprotects the data in the *double* format using data type preservation or a No Encryption data element.

func (s *Session) UnprotectDouble(data float64, dataElement string, options ...ProtectionOptions) (float64, int16, error)

Parameters

data: Input containing the data to be unprotected in *float64* format.

dataElement: Input containing the data element defined in the policy in *string* format.

options:

- ***apgo.UsingExtIV***: External IV is an optional parameter. It is a buffer containing data that is used as an initialization vector and accepts input in byte format. When the External IV is empty, its value is ignored.

Returns

float64: Returns the unprotected data in *float64* format.

int16: Returns the return code of the unprotect operation in *int16* format.

error: Error message if the unprotect operation fails.

Example

The following is a sample for the *UnprotectDouble* API.

```
output, rc, err := session.UnprotectDouble(input, dataElement,
apgo.UsingExtIV([]byte(externalIv)))
```

2.2.31 DecryptStr API

The *DecryptStr* API decrypts the data in the *string* format using an encryption data element.

func (s *Session) DecryptStr(data []byte, dataElement string, options ...ProtectionOptions) (string, int16, error)

Parameters

data: Input containing the data to be protected in *byte* format.

dataElement: String containing the data element name defined in the policy in *string* format.

options:

- ***apgo.UsingExtIV***: External IV is an optional parameter. It is a buffer containing data that is used as an initialization vector and accepts input in byte format. When the External IV is empty, its value is ignored.

  > **Note:**
  > Encryption data elements do not support external IV.

- ***apgo.UsingExtTweak***: External Tweak is an optional parameter that is used only for the FPE data elements. It is a buffer containing data that is used as an External Tweak and accepts input in byte format. When the External Tweak is empty, its value is ignored.

Returns

string: Returns the decrypted data in *string* format.

int16: Returns the return code of the decrypt operation in *int16* format.

error: Error message if the decrypt operation fails.

Example

The following is a sample for the *DecryptStr* API.

```
output, rc, err := session.DecryptStr(input, dataElement,
apgo.UsingExtIV([]byte(externalIv)), apgo.UsingExtTweak([]byte(externaltweak)))
```

2.2.32 DecryptShort API

The *DecryptShort* API decrypts the data in the *short* format using an encryption data element.

func (s *Session) DecryptShort(data []byte, dataElement string, options ...ProtectionOptions) (int16, int16, error)

Parameters

data: Input containing the data to be protected in *byte* format.

dataElement: Input containing the data element name defined in the policy in *string* format.

options:

- ***apgo.UsingExtIV***: External IV is an optional parameter. It is a buffer containing data that is used as an initialization vector and accepts input in byte format. When the External IV is empty, its value is ignored.

  > **Note:**
  > Encryption data elements do not support external IV.

Returns

int16: Returns the decrypted data in *int16* format.

int16: Returns the return code of the decrypt operation in *int16* format.

error: Error message if the decrypt operation fails.

Example

The following is a sample for the *DecryptShort* API.

```
output, rc, err := session.DecryptShort(input, dataElement,
apgo.UsingExtIV([]byte(externalIv)))
```

2.2.33 DecryptInt API

The *DecryptInt* API decrypts the data in the *integer* format using an encryption data element.

func (s *Session) DecryptInt(data []byte, dataElement string, options ...ProtectionOptions) (int32, int16, error)

Parameters

data: Input containing the data to be protected in *byte* format.

dataElement: Input containing the data element name defined in the policy in *string* format.

options:

- ***apgo.UsingExtIV***: External IV is an optional parameter. It is a buffer containing data that is used as an initialization vector and accepts input in byte format. When the External IV is empty, its value is ignored.

  > **Note:**
  > Encryption data elements do not support external IV.

Returns

int32: Returns the decrypted data in *int32* format.

int16: Returns the return code of the decrypt operation in *int16* format.

error: Error message if the decrypt operation fails.

Example

The following is a sample for the *DecryptInt* API.

```
output, rc, err := session.DecryptInt(input, dataElement,
apgo.UsingExtIV([]byte(externalIv)))
```

2.2.34 DecryptLong API

The *DecryptLong* API decrypts the data in the *long* format using an encryption data element.

func (s *Session) DecryptShort(data []byte, dataElement string, options ...ProtectionOptions) (int16, int16, error)

Parameters

data: Input containing the data to be protected in *byte* format.

dataElement: Input containing the data element name defined in the policy in *string* format.

options:

- ***apgo.UsingExtIV***: External IV is an optional parameter. It is a buffer containing data that is used as an initialization vector and accepts input in byte format. When the External IV is empty, its value is ignored.

  > **Note:**
  > Encryption data elements do not support external IV.

Returns

int16: Returns the decrypted data in *int16* format.

int16: Returns the return code of the decrypt operation in *int16* format.

error: Error message if the decrypt operation fails.

Example

The following is a sample for the *DecryptLong* API.

```
output, rc, err := session.DecryptLong(input, dataElement,
apgo.UsingExtIV([]byte(externalIv)))
```

2.2.35 DecryptFloat API

The *DecryptFloat* API decrypts the data in the *float* format using an encryption data element.

func (s *Session) DecryptFloat(data []byte, dataElement string) (float32, int16, error)

Parameters

data: Input containing the data to be protected in *byte* format.

dataElement: Input containing the data element name defined in the policy in *string* format.

Returns

float32: Returns the protected data in *float32* format.

int16: Returns the return code of the protect operation in *int16* format.

error: Error message if the decrypt operation fails.

Example

The following is a sample for the *DecryptFloat* API.

```
output, rc, err := session.DecryptFloat(input, dataElement,
apgo.UsingExtIV([]byte(externalIv)))
```

2.2.36 DecryptDouble API

The *DecryptDouble* API decrypts the data in the *double* format using an encryption data element.

func (s *Session) DecryptDouble(data []byte, dataElement string) (float64, int16, error)

Parameters

data: Input containing the data to be protected in *byte* format.

dataElement: Input containing the data element name defined in the policy in *string* format.

Returns

float64: Returns the decrypted data in *float64* format.

int16: Returns the return code of the decrypt operation in *int16* format.

error: Error message if the decrypt operation fails.

Example

The following is a sample for the *DecryptDouble* API.

```
output, rc, err := session.DecryptDouble(input, dataElement,
apgo.UsingExtIV([]byte(externalIv)))
```

2.2.37 ReProtectStr API

The *ReProtectStr* API reprotects the data in the *string* format using data type preservation or an No Encryption data element.

func (s *Session) ReprotectStr(data string, oldDataElement, newDataElement string, options ...ProtectionOptions) (string, int16, error)

Parameters

data: Input containing the data to be re-protected in *string* format.

oldDataElement: Input containing the older data element defined in the policy in *string* format.

newDataElement: Input containing the new data element defined in the policy in *string* format.

options:

- ***apgo.UsingExtIV***: External IV is an optional parameter. It is a buffer containing data that is used as an initialization vector and accepts input in byte format. When the External IV is empty, its value is ignored.
- ***apgo.UsingExtTweak***: External Tweak is an optional parameter that is used only for the FPE data elements. It is a buffer containing data that is used as an External Tweak and accepts input in byte format. When the External Tweak is empty, its value is ignored.
- ***apgo.UsingNewExtIV***: This parameter lets user specify New External IV for the reprotect operation.
- ***apgo.UsingNewExtTweak***: This parameter lets user specify New External Tweak for the reprotect operation.

Returns

string: Returns the re-protected data in *string* format.

int16: Returns the return code of the re-protect operation in *int16* format.

error: Error message if the re-protect operation fails.

Example

The following is a sample for the *ReProtectStr* API.

```
output, rc, err := session.ReProtectStr(input,
dataElement, apgo.UsingExtIV([]byte(externalIv)),
apgo.UsingNewExtIV([]byte(newExternalIv)), apgo.UsingExtTweak([]byte(externaltweak)),
apgo.UsingNewExtTweak([]byte(newExternaltweak)))
```

2.2.38 ReProtectShort API

The *ReProtectShort* API reprotects the data in the *short* format using data type preservation or an No Encryption data element.

func (s *Session) ReprotectShort(data int16, oldDataElement, newDataElement string, options ...ProtectionOptions) (int16, int16, error)

Parameters

data: Input containing the data to be re-protected in *int16* format.

oldDataElement: Input containing the older data element defined in the policy in *string* format.

newDataElement: Input containing the new data element defined in the policy in *string* format.

options:

- ***apgo.UsingExtIV***: External IV is an optional parameter. It is a buffer containing data that is used as an initialization vector and accepts input in byte format. When the External IV is empty, its value is ignored.
- ***apgo.UsingNewExtIV***: This parameter lets user specify New External IV for the reprotect operation.

Returns

int16: Returns the re-protected data in *int16* format.

int16: Returns the return code of the re-protect operation in *int16* format.

error: Error message if the re-protect operation fails.

Example

The following is a sample for the *ReProtectShort* API.

```
output, rc, err := session.ReProtectShort(input, dataElement,
apgo.UsingExtIV([]byte(externalIv)), apgo.UsingNewExtIV([]byte(newExternalIv)))
```

2.2.39 ReprotectInt API

The *ReprotectInt* API reprotects the data in the *integer* format using data type preservation or a No Encryption data element.

func (s *Session) ReprotectInt(data int32, oldDataElement, newDataElement string, options ...ProtectionOptions) (int32, int16, error)

Parameters

data: Input containing the data to be re-protected in *int32* format.

oldDataElement: Input containing the older data element defined in the policy in *string* format.

newDataElement: Input containing the new data element defined in the policy in *string* format.

options:

- ***apgo.UsingExtIV***: External IV is an optional parameter. It is a buffer containing data that is used as an initialization vector and accepts input in byte format. When the External IV is empty, its value is ignored.
- ***apgo.UsingNewExtIV***: This parameter lets user specify New External IV for the reprotect operation.

Returns

int32: Returns the reprotected data in *int32* format.

int16: Returns the return code of the reprotect operation in *int16* format.

error: Error message if the re-protect operation fails.

Example

The following is a sample for the *ReprotectInt* API.

```
output, rc, err := session.ReprotectInt(input, dataElement,
apgo.UsingExtIV([]byte(externalIv)), apgo.UsingNewExtIV([]byte(newExternalIv)))
```

2.2.40 ReProtectLong API

The *ReProtectLong* API reprotects the data in the *long* format using data type preservation or a No Encryption data element.

func (s *Session) ReprotectLong(data int64, oldDataElement, newDataElement string, options ...ProtectionOptions) (int64, int16, error)

Parameters

data: Input containing the data to be re-protected in *int64* format

oldDataElement: Input containing the older data element defined in the policy in *string* format

newDataElement: Input containing the new data element defined in the policy in *string* format

options:

- ***apgo.UsingExtIV***: External IV is an optional parameter. It is a buffer containing data that is used as an initialization vector, and accepts input in byte format. When the External IV is empty, its value is ignored.
- ***apgo.UsingNewExtIV***: This parameter lets user specify New External IV for the reprotect operation.

Returns

int64: Returns the re-protected data in *int64* format.

int16: Returns the return code of the re-protect operation in *int16* format.

error: Error message if the re-protect operation fails.

Example

The following is a sample for the *ReProtectLong* API.

```
output, rc, err := session.ReProtectLong(input, dataElement,
apgo.UsingExtIV([]byte(externalIv)), apgo.UsingNewExtIV([]byte(newExternalIv)))
```

2.2.41 ReProtectFloat API

The *ReProtectFloat* API reprotects the data in the *float* format using a No Encryption data element.

func (s *Session) ReprotectFloat(data float32, oldDataElement, newDataElement string, options ...ProtectionOptions) (float32, int16, error)

Parameters

data: Input containing the data to be reprotected in *float32* format.

oldDataElement: Input containing the older data element defined in the policy in *string* format.

newDataElement: Input containing the new data element defined in the policy in *string* format.

options:

- ***apgo.UsingExtIV***: External IV is an optional parameter. It is a buffer containing data that is used as an initialization vector and accepts input in byte format. When the External IV is empty, its value is ignored.
- ***apgo.UsingNewExtIV***: This parameter lets user specify New External IV for the reprotect operation.

Returns

float32: Returns the re-protected data in *float32* format.

int16: Returns the return code of the reprotect operation in *int16* format.

error: Error message if the reprotect operation fails.

Example

The following is a sample for the *ReProtectFloat* API.

```
output, rc, err := session.ReProtectFloat(input, dataElement,
apgo.UsingExtIV([]byte(externalIv)), apgo.UsingNewExtIV([]byte(newExternalIv)))
```

2.2.42 ReProtectDouble API

The *ReProtectDouble* API reprotects the data in the *double* format using a No Encryption data element.

func (s *Session) ReprotectDouble(data float64, oldDataElement, newDataElement string, options ...ProtectionOptions) (float64, int16, error)

Parameters

data: Input containing the data to be re-protected in *float64* format.

oldDataElement: Input containing the older data element defined in the policy in *string* format.

newDataElement: Input containing the new data element defined in the policy in *string* format.

options:

- ***apgo.UsingExtIV***: External IV is an optional parameter. It is a buffer containing data that is used as an initialization vector and accepts input in byte format. When the External IV is empty, its value is ignored.
- ***apgo.UsingNewExtIV***: This parameter lets user specify New External IV for the reprotect operation.

Returns

float64: Returns the re-protected data in *float64* format.

int16: Returns the return code of the re-protect operation in *int16* format.

error: Error message if the reprotect operation fails.

Example

The following is a sample for the *ReProtectDouble* API.

```
output, rc, err := session.ReProtectDouble(input, dataElement,
apgo.UsingExtIV([]byte(externalIv)), apgo.UsingNewExtIV([]byte(newExternalIv)))
```

2.2.43 ReEncryptBytes API

The *ReEncryptBytes* API re-encrypts the data in the *byte* format using an encryption data element.

func (s *Session) ReEncryptBytes(data []byte, oldDataElement, newDataElement string, options ...ProtectionOptions) ([]byte, int16, error)

Parameters

data: Input containing the data to be re-encrypted in *byte* format.

oldDataElement: String containing the older data element name defined in the policy in *string* format.

newDataElement: String containing the new data element name defined in the policy in *string* format.

options:

- ***apgo.UsingExtIV***: External IV is an optional parameter. It is a buffer containing data that is used as an initialization vector and accepts input in byte format. When the External IV is empty, its value is ignored.

  > **Note:**
  > Encryption data elements do not support external IV.

- ***apgo.UsingExtTweak***: External Tweak is an optional parameter that is used only for the FPE data elements. It is a buffer containing data that is used as an external tweak and accepts input in byte format. When the external tweak is empty, its value is ignored.
- ***apgo.UsingNewExtIV***: This parameter lets user specify New External IV for the reprotect operation.

  > **Note:**
  > Encryption data elements do not support external IV.

- ***apgo.UsingNewExtTweak***: This parameter lets user specify New External Tweak for the reprotect operation.

Returns

byte: Returns the re-encrypted data in *byte* format.

int16: Returns the return code of the re-encrypt operation in *int16* format.

error: Error message if the re-encrypt operation fails.

Example

The following is a sample for the *ReEncryptBytes* API.

```
output, rc, err := session.ReEncryptBytes(input,
dataElement, apgo.UsingExtIV([]byte(externalIv)),
apgo.UsingNewExtIV([]byte(newExternalIv)), apgo.UsingExtTweak([]byte(externaltweak)),
apgo.UsingNewExtTweak([]byte(newExternaltweak)))
```

2.2.44 ProtectBytesBulk API

The *ProtectBytesBulk* API protects the data of the *bytes* format in *bulk* using data type preservation or a No Encryption data element.

func (session *Session) ProtectStrBulk(data [][]byte, dataElement string, options ...ProtectionOptions) ([][]byte, []int16, error)

Parameters

data: Input containing the data to be protected in *byte* format.

dataElement: Input containing the data element defined in the policy in *string* format.

options:

- ***apgo.UsingExtIV***: External IV is an optional parameter. It is a buffer containing data that is used as an initialization vector and accepts input in byte format. When the External IV is empty, its value is ignored.
- ***apgo.UsingNewExtIV***: This parameter lets user specify New External IV for the reprotect operation.

Returns

byte: Returns output array in *byte* format.

int16: Returns the return code array of the bulk protect operation in *int16* format.

error: Error message if the bulk protect operation fails.

Example

The following is a sample for the *ProtectBytesBulk* API.

```
output, rc, err := session.ProtectBytesBulk(input, dataElement,
apgo.UsingExtIV([]byte(externalIv)), apgo.UsingExtTweak([]byte(externaltweak)))
```

2.2.45 ProtectStrBulk API

The *ProtectStrBulk* API protects the data in the *string* format in *bulk* using data type preservation or a No Encryption data element.

func (s *Session) ProtectStrBulk(data []string, dataElement string, options ...ProtectionOptions) ([]string, []int16, error)

Parameters

data: Input containing the data to be protected in *string* format.

dataElement: Input containing the data element defined in the policy in *string* format.

options:

- ***apgo.UsingExtIV***: External IV is an optional parameter. It is a buffer containing data that is used as an initialization vector and accepts input in byte format. When the External IV is empty, its value is ignored.
- ***apgo.UsingExtTweak***: External Tweak is an optional parameter that is used only for the FPE data elements. It is a buffer containing data that is used as an External Tweak and accepts input in byte format. When the External Tweak is empty, its value is ignored.

Returns

string: Returns output array in *string* format.

int16: Returns the return code array of the bulk protect operation in *int16* format.

error: Error message if the bulk protect operation fails.

Example

The following is a sample for the *ProtectStrBulk* API.

2.2.46 ProtectShortBulk API

The *ProtectShortBulk* API protects the data in the *short* format in *bulk* using data type preservation or a No Encryption data element.

func (s *Session) ProtectShortBulk(data []int16, dataElement string, options ...ProtectionOptions) ([]int16, []int16, error)

Parameters

data: Input containing the data to be protected in *int16* format.

dataElement: Input containing the data element defined in the policy in *string* format.

options:

- ***apgo.UsingExtIV***: External IV is an optional parameter. It is a buffer containing data that is used as an initialization vector and accepts input in byte format. When the External IV is empty, its value is ignored.

Returns

int16: Returns output array in *int16* format.

int16: Returns the return code array of the bulk protect operation in *int16* format.

error: Error message if the bulk protect operation fails.

Example

The following is a sample for the *ProtectShortBulk* API.

```
output, rc, err := session.ProtectShortBulk(input, dataElement,
apgo.UsingExtIV([]byte(externalIv)))
```

2.2.47 ProtectIntBulk API

The *ProtectIntBulk* API protects the data in the *integer* format in *bulk* using data type preservation or a No Encryption data element.

func (s *Session) ProtectIntBulk(data []int32, dataElement string, options ...ProtectionOptions) ([]int32, []int16, error)

Parameters

data: Input containing the data to be protected in *int32* format

dataElement: Input containing the data element defined in the policy in *string* format.

options:

- ***apgo.UsingExtIV***: External IV is an optional parameter. It is a buffer containing data that is used as an initialization vector and accepts input in byte format. When the External IV is empty, its value is ignored.

Returns

int32: Returns output array in *int32* format.

int16: Returns the return code array of the bulk protect operation in *int16* format.

error: Error message if the bulk protect operation fails.

Example

The following is a sample for the *ProtectIntBulk* API.

```
output, rc, err := session.ProtectIntBulk(input, dataElement,
apgo.UsingExtIV([]byte(externalIv)))
```

2.2.48 ProtectLongBulk API

The *ProtectLongBulk* API protects the data in the *long* format in *bulk* using data type preservation or a No Encryption data element.

func (s *Session) ProtectLongBulk(data []int64, dataElement string, options ...ProtectionOptions) ([]int64, []int16, error)

Parameters

data: Input containing the data to be protected in *int64* format.

dataElement: Input containing the data element defined in the policy in *string* format.

options:

- ***apgo.UsingExtIV***: External IV is an optional parameter. It is a buffer containing data that is used as an initialization vector and accepts input in byte format. When the External IV is empty, its value is ignored.

Returns

int64: Returns output array in *int64* format.

int16: Returns the return code array of the bulk protect operation in *int16* format.

error: Error message if the bulk protect operation fails.

Example

The following is a sample for the *ProtectLongBulk* API.

```
output, rc, err := session.ProtectLongBulk(input, dataElement,
apgo.UsingExtIV([]byte(externalIv))
```

2.2.49 ProtectFloatBulk API

The *ProtectFloatBulk* API protects the data in the *float* format in *bulk* using a No Encryption data element.

func (s *Session) ProtectFloatBulk(data []float32, dataElement string, options ...ProtectionOptions) ([]float32, []int16, error)

Parameters

data: Input containing the data to be protected in *float32* format.

dataElement: Input containing the data element defined in the policy in *string* format.

options:

- ***apgo.UsingExtIV***: External IV is an optional parameter. It is a buffer containing data that is used as an initialization vector and accepts input in byte format. When the External IV is empty, its value is ignored.

Returns

float32: Returns output array in *float32* format.

int16: Returns the return code array of the bulk protect operation in *int16* format.

error: Error message if the bulk protect operation fails.

Example

The following is a sample for the *ProtectFloatBulk* API.

```
output, rc, err := session.ProtectFloatBulk(input, dataElement,
apgo.UsingExtIV([]byte(externalIv)))
```

2.2.50 ProtectDoubleBulk API

The *ProtectDoubleBulk* API protects the data in the *double* format in *bulk* using a No Encryption data element.

func (s *Session) ProtectDoubleBulk(data []float64, dataElement string, options ...ProtectionOptions) ([]float64, []int16, error)

Parameters

data: Input containing the data to be protected in *float64* format.

dataElement: Input containing the data element defined in the policy in *string* format.

options:

- ***apgo.UsingExtIV***: External IV is an optional parameter. It is a buffer containing data that is used as an initialization vector and accepts input in byte format. When the External IV is empty, its value is ignored.

Returns

float64: Returns output array in *float64* format.

int16: Returns the return code array of the bulk protect operation in *int16* format.

error: Error message if the bulk protect operation fails.

Example

The following is a sample for the *ProtectDoubleBulk* API.

```
output, rc, err := session.ProtectDoubleBulk(input, dataElement,
apgo.UsingExtIV([]byte(externalIv)))
```

2.2.51 EncryptStrBulk API

The *EncryptStrBulk* API encrypts the data in the *string* format in *bulk* using an encryption data element.

func (s *Session) EncryptStrBulk(data []string, dataElement string, options ...ProtectionOptions) ([][]byte, []int16, error)

Parameters

data: Input containing the data to be protected in *string* format.

dataElement: Input containing the data element name defined in the policy in *string* format.

options:

- ***apgo.UsingExtIV***: External IV is an optional parameter. It is a buffer containing data that is used as an initialization vector and accepts input in byte format. When the External IV is empty, its value is ignored.

  > **Note:**
  > Encryption data elements do not support external IV.

- ***apgo.UsingExtTweak***: External Tweak is an optional parameter that is used only for the FPE data elements. It is a buffer containing data that is used as an External Tweak and accepts input in byte format. When the External Tweak is empty, its value is ignored.

Returns

byte: Returns the encrypted data in *byte* format.

int16: Returns the return code array of the encrypt operation in *int16* format.

error: Error message if the encrypt operation fails.

Example

The following is a sample for the *EncryptStrBulk* API.

```
output, rc, err := session.EncryptStrBulk(input, dataElement,
apgo.UsingExtIV([]byte(externalIv)), apgo.UsingExtTweak([]byte(externaltweak)))
```

2.2.52 EncryptShortBulk API

The *EncryptShortBulk* API encrypts the data in the *short* format in *bulk* using an encryption data element.

func (s *Session) EncryptShortBulk(data []int16, dataElement string, options ...ProtectionOptions) ([][]byte, []int16, error)

Parameters

data: Input containing the data to be protected in *int16* format.

dataElement: Input containing the data element name defined in the policy in *string* format.

options:

- ***apgo.UsingExtIV***: External IV is an optional parameter. It is a buffer containing data that is used as an initialization vector and accepts input in byte format. When the External IV is empty, its value is ignored.

  > **Note:**

> Encryption data elements do not support external IV.

Returns

byte: Returns the encrypted data in *byte* format.

int16: Returns the return code array of the encrypt operation in *int16* format.

error: Error message if the encrypt operation fails.

Example

The following is a sample for the *EncryptShortBulk* API.

```
output, rc, err := session.EncryptShortBulk(input, dataElement,
apgo.UsingExtIV([]byte(externalIv)))
```

2.2.53 EncryptIntBulk API

The *EncryptIntBulk* API encrypts the data in the *integer* format in *bulk* using an encryption data element.

func (s *Session) EncryptIntBulk(data []int32, dataElement string, options ...ProtectionOptions) ([][]byte, []int16, error)

Parameters

data: Input containing the data to be protected in *int32* format.

dataElement: Input containing the data element name defined in the policy in *string* format.

options:

- ***apgo.UsingExtIV***: External IV is an optional parameter. It is a buffer containing data that is used as an initialization vector and accepts input in byte format. When the External IV is empty, its value is ignored.

> **Note:**
> Encryption data elements do not support external IV.

Returns

byte: Returns the encrypted data in *byte* format.

int16: Returns the return code array of the encrypt operation in *int16* format.

error: Error message if the encrypt operation fails.

Example

The following is a sample for the API.

```
output, rc, err := session.EncryptIntBulk(input, dataElement,
apgo.UsingExtIV([]byte(externalIv)))
```

2.2.54 EncryptLongBulk API

The *EncryptLongBulk* API encrypts the data in the *long* format in *bulk* using an encryption data element.

func (s *Session) EncryptLongBulk(data []int64, dataElement string, options ...ProtectionOptions) ([][]byte, []int16, error)

Parameters

data: Input containing the data to be protected in *int64* format.

dataElement: Input containing the data element name defined in the policy in *string* format.

options:

- ***apgo.UsingExtIV***: External IV is an optional parameter. It is a buffer containing data that is used as an initialization vector and accepts input in byte format. When the External IV is empty, its value is ignored.

  > **Note:**
  > Encryption data elements do not support external IV.

Returns

byte: Returns the encrypted data in *byte* format.

int16: Returns the return code array of the encrypt operation in *int16* format.

error: Error message if the encrypt operation fails.

Example

The following is a sample for the *EncryptLongBulk* API.

```
output, rc, err := session.EncryptLongBulk(input, dataElement,
apgo.UsingExtIV([]byte(externalIv)))
```

2.2.55 EncryptFloatBulk API

The *EncryptFloatBulk* API encrypts the data in the *float* format in *bulk* using an encryption data element.

func (s *Session) EncryptFloatBulk(data []float32, dataElement string) ([][]byte, []int16, error)

Parameters

data: Input containing the data to be protected in *float32* format.

dataElement: Input containing the data element name defined in the policy in *string* format.

Returns

byte: Returns the encrypted data in *byte* format.

int16: Returns the return code array of the encrypt operation in *int16* format.

error: Error message if the encrypt operation fails.

Example

The following is a sample for the *EncryptFloatBulk* API.

```
output, rc, err := session.EncryptFloatBulk(input, dataElement,
apgo.UsingExtIV([]byte(externalIv)))
```

2.2.56 EncryptDoubleBulk API

The *EncryptDoubleBulk* API encrypts the data in the *double* format in *bulk* using an encryption data element.

func (s *Session) EncryptDoubleBulk(data []float64, dataElement string) ([][]byte, []int16, error)

Parameters

data: Input containing the data to be protected in *float64* format.

dataElement: Input containing the data element name defined in the policy in *string* format.

Returns

byte: Returns the encrypted data in *byte* format.

int16: Returns the return code array of the encrypt operation in *int16* format.

error: Error message if the encrypt operation fails.

Example

The following is a sample for the *EncryptDoubleBulk* API.

```
output, rc, err := session.EncryptDoubleBulk(input, dataElement,
apgo.UsingExtIV([]byte(externalIv)))
```

2.2.57 UnprotectBytesBulk API

The *UnprotectBytesBulk* API unprotects the data in the *byte* format in *bulk* using data type preservation or a No Encryption data element.

func (s *Session) UnprotectBytesBulk(data [][]byte, dataElement string, options ...ProtectionOptions) ([][]byte, []int16, error)

Parameters

data: Input containing the data to be unprotected in *byte* format.

dataElement: Input containing the data element defined in the policy in *string* format.

options:

- ***apgo.UsingExtIV***: External IV is an optional parameter. It is a buffer containing data that is used as an initialization vector and accepts input in byte format. When the External IV is empty, its value is ignored.
- ***apgo.UsingExtTweak***: External Tweak is an optional parameter that is used only for the FPE data elements. It is a buffer containing data that is used as an External Tweak and accepts input in byte format. When the External Tweak is empty, its value is ignored.

Returns

byte: Returns the unprotected data in *byte* format.

int16: Returns the return code array of the bulk unprotect operation in *int16* format.

error: Error message if the bulk unprotect operation fails.

Example

The following is a sample for the *UnprotectBytesBulk* API.

```
output, rc, err := session.UnprotectBytesBulk(input, dataElement,
apgo.UsingExtIV([]byte(externalIv)), apgo.UsingExtTweak([]byte(externaltweak)))
```

2.2.58 UnprotectStrBulk API

The *UnprotectStrBulk* API unprotects the data in the *string* format in *bulk* using data type preservation or a No Encryption data element.

func (session *Session) UnprotectStrBulk(data []string, dataElement string, options ...ProtectionOptions) ([]string, []int16, error)

Parameters

data: Input containing the data to be unprotected in *string* format.

dataElement: Input containing the data element defined in the policy in *string* format.

options:

- ***apgo.UsingExtIV***: External IV is an optional parameter. It is a buffer containing data that is used as an initialization vector and accepts input in byte format. When the External IV is empty, its value is ignored.
- ***apgo.UsingExtTweak***: External Tweak is an optional parameter that is used only for the FPE data elements. It is a buffer containing data that is used as an External Tweak and accepts input in byte format. When the External Tweak is empty, its value is ignored.

Returns

string: Returns the unprotected data in *string* format.

int16: Returns the return code array of the bulk unprotect operation in *int16* format.

error: Error message if the bulk unprotect operation fails.

Example

The following is a sample for the *UnprotectStrBulk* API.

```
output, rc, err := session.UnprotectStrBulk(input, dataElement,
apgo.UsingExtIV([]byte(externalIv)), apgo.UsingExtTweak([]byte(externaltweak)))
```

2.2.59 UnprotectShortBulk API

The *UnprotectShortBulk* API unprotects the data in the *short* format in *bulk* using data type preservation or a No Encryption data element.

func (s *Session) UnprotectShortBulk(data []int16, dataElement string, options ...ProtectionOptions) ([]int16, []int16, error)

Parameters

data: Input containing the data to be unprotected in *int16* format.

dataElement: Input containing the data element defined in the policy in *string* format.

options:

- ***apgo.UsingExtIV***: External IV is an optional parameter. It is a buffer containing data that is used as an initialization vector and accepts input in byte format. When the External IV is empty, its value is ignored.

Returns

int16: Returns the unprotected data in *int16* format.

int16: Returns the return code array of the unprotect operation in *int16* format.

error: Error message if the unprotect operation fails.

Example

The following is a sample for the *UnprotectShortBulk* API.

```
output, rc, err := session.UnprotectShortBulk(input, dataElement,
apgo.UsingExtIV([]byte(externalIv)))
```

2.2.60 UnprotectIntBulk API

The *UnprotectIntBulk* API unprotects the data in the *integer* format in *bulk* using data type preservation or a No Encryption data element.

func (s *Session) UnprotectIntBulk(data []int32, dataElement string, options ...ProtectionOptions) ([]int32, []int16, error)

Parameters

data: Input containing the data to be unprotected in *int32* format.

dataElement: Input containing the data element defined in the policy in *string* format.

options:

- ***apgo.UsingExtIV***: External IV is an optional parameter. It is a buffer containing data that is used as an initialization vector and accepts input in byte format. When the External IV is empty, its value is ignored.

Returns

int32: Returns the unprotected data in *int32* format.

int16: Returns the return code array of the unprotect operation in *int16* format.

error: Error message if the unprotect operation fails.

Example

The following is a sample for the *UnprotectIntBulk* API.

```
output, rc, err := session.UnprotectIntBulk(input, dataElement,
apgo.UsingExtIV([]byte(externalIv)))
```

2.2.61 UnprotectLongBulk API

The *UnprotectLongBulk* API unprotects the data in the *long* format in *bulk* using data type preservation or a No Encryption data element.

func (s *Session) UnprotectLongBulk(data []int64, dataElement string, options ...ProtectionOptions) ([]int64, []int16, error)

Parameters

data: Input containing the data to be unprotected in *int64* format.

dataElement: Input containing the data element defined in the policy in *string* format.

options:

- ***apgo.UsingExtIV***: External IV is an optional parameter. It is a buffer containing data that is used as an initialization vector and accepts input in byte format. When the External IV is empty, its value is ignored.

Returns

int64: Returns the unprotected data in *int64* format.

int16: Returns the return code array of the unprotect operation in *int16* format.

error: Error message if the unprotect operation fails.

Example

The following is a sample for the *UnprotectLongBulk* API.

```
output, rc, err := session.UnprotectLongBulk(input, dataElement,
apgo.UsingExtIV([]byte(externalIv)))
```

2.2.62 UnprotectFloatBulk API

The *UnprotectFloatBulk* API unprotects the data in the *float* format in *bulk* using a No Encryption data element.

func (s *Session) UnprotectFloatBulk(data []float32, dataElement string, options ...ProtectionOptions) ([]float32, []int16, error)

Parameters

data: Input containing the data to be unprotected in *float32* format.

dataElement: Input containing the data element defined in the policy in *string* format.

options:

- ***apgo.UsingExtIV***: External IV is an optional parameter. It is a buffer containing data that is used as an initialization vector and accepts input in byte format. When the External IV is empty, its value is ignored.

Returns

float32: Returns the unprotected data in *float32* format.

int16: Returns the return code array of the unprotect operation in *int16* format.

error: Error message if the unprotect operation fails.

Example

The following is a sample for the *UnprotectFloatBulk* API.

```
output, rc, err := session.UnprotectFloatBulk(input, dataElement,
apgo.UsingExtIV([]byte(externalIv)))
```

2.2.63 UnprotectDoubleBulk API

The *UnprotectDoubleBulk* API unprotects the data in the *double* format in *bulk* using a Encryption data element.

func (s *Session) UnprotectDoubleBulk(data []float64, dataElement string, options ...ProtectionOptions) ([]float64, []int16, error)

Parameters

`data`: Input containing the data to be unprotected in *float64* format.

`dataElement`: Input containing the data element defined in the policy in *string* format.

`options`:

- `apgo.UsingExtIV`: External IV is an optional parameter. It is a buffer containing data that is used as an initialization vector and accepts input in byte format. When the External IV is empty, its value is ignored.

Returns

`float64`: Returns the unprotected data in *float64* format.

`int16`: Returns the return code array of the unprotect operation in *int16* format.

`error`: Error message if the unprotect operation fails.

Example

The following is a sample for the *UnprotectDoubleBulk* API.

```
output, rc, err := session.UnprotectDoubleBulk(input, dataElement,
apgo.UsingExtIV([]byte(externalIv)))
```

2.2.64 DecryptStrBulk API

The *DecryptStrBulk* API decrypts the data in the *string* format in *bulk* using a decryption data element.

func (s *Session) DecryptStrBulk(data [][]byte, dataElement string, options ...ProtectionOptions) ([]string, []int16, error)

Parameters

`data`: Input containing the data to be decrypted in *byte* format.

`dataElement`: Input containing the data element name defined in the policy in *string* format.

`options`:

- `apgo.UsingExtIV`: External IV is an optional parameter. It is a buffer containing data that is used as an initialization vector and accepts input in byte format. When the External IV is empty, its value is ignored.

  > **Note:**
  > Encryption data elements do not support external IV.

- `apgo.UsingExtTweak`: External Tweak is an optional parameter that is used only for the FPE data elements. It is a buffer containing data that is used as an initialization vector, and accepts input in byte format. When the External Tweak is empty, its value is ignored.

Returns

`string`: Returns the decrypted data in *string* format.

`int16`: Returns the return code array of the decrypt operation in *int16* format.

`error`: Error message if the decrypt operation fails.

Example

The following is a sample for the *DecryptStrBulk* API.

```
output, rc, err := session.DecryptStrBulk(input, dataElement,
apgo.UsingExtIV([]byte(externalIv)), apgo.UsingExtTweak([]byte(externaltweak)))
```

2.2.65 DecryptShortBulk API

The *DecryptShortBulk* API decrypts the data in the *short* format in *bulk* using a decryption data element.

func (s *Session) DecryptShortBulk(data [][]byte, dataElement string, options ...ProtectionOptions) ([]int16, []int16, error)

Parameters

data: Input containing the data to be decrypted in *byte* format.

dataElement: Input containing the data element name defined in the policy in *string* format.

options:

- ***apgo.UsingExtIV***: External IV is an optional parameter. It is a buffer containing data that is used as an initialization vector and accepts input in byte format. When the External IV is empty, its value is ignored.

> **Note:**
> Encryption data elements do not support external IV.

Returns

int16: Returns the decrypted data in *int16* format.

int16: Returns the return code array of the decrypt operation in *int16* format.

error: Error message if the decrypt operation fails.

Example

The following is a sample for the *DecryptShortBulk* API.

```
output, rc, err := session.DecryptShortBulk(input, dataElement,
apgo.UsingExtIV([]byte(externalIv)))
```

2.2.66 DecryptIntBulk API

The *DecryptIntBulk* API decrypts the data in the *int* format in *bulk* using a decryption data element.

func (s *Session) DecryptIntBulk(data [][]byte, dataElement string, options ...ProtectionOptions) ([]int32, []int16, error)

Parameters

data: Input containing the data to be decrypted in *byte* format.

dataElement: Input containing the data element name defined in the policy in *string* format.

options:

- ***apgo.UsingExtIV***: External IV is an optional parameter. It is a buffer containing data that is used as an initialization vector and accepts input in byte format. When the External IV is empty, its value is ignored.

> **Note:**
> Encryption data elements do not support external IV.

Returns

int32: Returns the decrypted data in *int32* format.

int16: Returns the return code array of the decrypt operation in *int16* format.

error: Error message if the decrypt operation fails.

Example

The following is a sample for the *DecryptIntBulk* API.

```
output, rc, err := session.DecryptIntBulk(input, dataElement,
apgo.UsingExtIV([]byte(externalIv)))
```

2.2.67 DecryptLongBulk API

The *DecryptLongBulk* API decrypts the data in the *long* format in *bulk* using a decryption data element.

func (s *Session) DecryptLongBulk(data [][]byte, dataElement string, options ...ProtectionOptions) ([]int64, []int16, error)

Parameters

data: Input containing the data to be decrypted in *byte* format.

dataElement: Input containing the data element name defined in the policy in *string* format.

options:

- ***apgo.UsingExtIV***: External IV is an optional parameter. It is a buffer containing data that is used as an initialization vector and accepts input in byte format. When the External IV is empty, its value is ignored.

  > **Note:**
  > Encryption data elements do not support external IV.

Returns

int64: Returns the decrypted data in *int64* format.

int16: Returns the return code array of the decrypt operation in *int16* format.

error: Error message if the decrypt operation fails.

Example

The following is a sample for the *DecryptLongBulk* API.

```
output, rc, err := session.DecryptLongBulk(input, dataElement,
apgo.UsingExtIV([]byte(externalIv)))
```

2.2.68 DecryptFloatBulk API

The *DecryptFloatBulk* API decrypts the data in the *float* format in *bulk* using a decryption data element.

func (s *Session) DecryptFloatBulk(data [][]byte, dataElement string) ([]float32, []int16, error)

Parameters

data: Input containing the data to be decrypted in *byte* format.

dataElement: Input containing the data element name defined in the policy in *string* format.

Returns

float32: Returns the decrypted data in *float32* format data.

int16: Returns the return code array of the decrypt operation in *int16* format.

error: Error message if the decrypt operation fails.

Example

The following is a sample for the *DecryptFloatBulk* API.

```
output, rc, err := session.DecryptFloatBulk(input, dataElement,
apgo.UsingExtIV([]byte(externalIv)))
```

2.2.69 DecryptDoubleBulk API

The *DecryptDoubleBulk* API decrypts the data in the *double* format in *bulk* using a decryption data element.

func (s *Session) DecryptDoubleBulk(data [][]byte, dataElement string) ([]float64, []int16, error)

Parameters

data: Input containing the data to be decrypted in *byte* format.

dataElement: Input containing the data element name defined in the policy in *string* format.

Returns

float64: Returns the decrypted data in *float64* format.

int16: Returns the return code array of the decrypt operation in *int16* format.

error: Error message if the decrypt operation fails.

Example

The following is a sample for the *DecryptDoubleBulk* API.

```
output, rc, err := session.DecryptDoubleBulk(input, dataElement,
apgo.UsingExtIV([]byte(externalIv)))
```

2.2.70 ReProtectStrBulk API

The *ReProtectStrBulk* API reprotects the data in the *string* format in *bulk* using data type preservation or a No Encryption data element.

func (s *Session) ReProtectStrBulk(data []string, oldDataElement, newDataElement string, options ...ProtectionOptions) ([]string, []int16, error)

Parameters

data: Input containing the data to be re-protected in *string* format.

oldDataElement: Input containing the older data element defined in the policy in *string* format.

newDataElement: Input containing the new data element defined in the policy in *string* format.

options:

- ***apgo.UsingExtIV***: External IV is an optional parameter. It is a buffer containing data that is used as an initialization vector and accepts input in byte format. When the External IV is empty, its value is ignored.
- ***apgo.UsingExtTweak***: External Tweak is an optional parameter that is used only for the FPE data elements. It is a buffer containing data that is used as an External Tweak and accepts input in byte format. When the External Tweak is empty, its value is ignored.
- ***apgo.UsingNewExtIV***: This parameter lets user specify New External IV for the reprotect operation.
- ***apgo.UsingNewExtTweak***: This parameter lets user specify New External Tweak for the reprotect operation.

Returns

string: Returns the re-protected data in *string* format.

int16: Returns the return code array of the reprotect operation in *int16* format.

error: Error message if the re-protect operation fails.

Example

The following is a sample for the *ReProtectStrBulk* API.

```
output, rc, err := session.ReProtectStrBulk(input,
dataElement, apgo.UsingExtIV([]byte(externalIv)),
apgo.UsingNewExtIV([]byte(newExternalIv)), apgo.UsingExtTweak([]byte(externaltweak)),
apgo.UsingNewExtTweak([]byte(newExternaltweak)))
```

2.2.71 ReProtectShortBulk API

The *ReProtectShortBulk* API reprotects the data in the *float* format in *bulk* using data type preservation or a No Encryption data element.

func (s *Session) ReProtectShortBulk(data []int16, oldDataElement, newDataElement string, options ...ProtectionOptions) ([]int16, []int16, error)

Parameters

data: Input containing the data to be re-protected in *int16* format.

oldDataElement: Input containing the older data element defined in the policy in *string* format.

newDataElement: Input containing the new data element defined in the policy in *string* format.

options:

- ***apgo.UsingExtIV***: External IV is an optional parameter. It is a buffer containing data that is used as an initialization vector and accepts input in byte format. When the External IV is empty, its value is ignored.
- ***apgo.UsingNewExtIV***: This parameter lets user specify New External IV for the reprotect operation.

Returns

int16: Returns the re-protected data in *int16* format.

int16: Returns the return code array of the re-protect operation in *int16* format.

error: Error message if the re-protect operation fails.

Example

The following is a sample for the *ReProtectShortBulk* API.

```
output, rc, err := session.ReProtectShortBulk(input, dataElement,
apgo.UsingExtIV([]byte(externalIv)), apgo.UsingNewExtIV([]byte(newExternalIv)))
```

2.2.72 ReProtectIntBulk API

The *ReProtectIntBulk* API reprotects the data in the *float* format in *bulk* using data type preservation or a No Encryption data element.

func (s *Session) ReProtectIntBulk(data []int32, oldDataElement, newDataElement string, options ...ProtectionOptions) ([]int32, []int16, error)

Parameters

data: Input containing the data to be re-protected in *int32* format.

oldDataElement: Input containing the older data element defined in the policy in *string* format.

newDataElement: Input containing the new data element defined in the policy in *string* format.

options:

- ***apgo.UsingExtIV***: External IV is an optional parameter. It is a buffer containing data that is used as an initialization vector and accepts input in byte format. When the External IV is empty, its value is ignored.
- ***apgo.UsingNewExtIV***: This parameter lets user specify New External IV for the reprotect operation.

Returns

int32: Returns the re-protected data in *int32* format.

int16: Returns the return code array of the re-protect operation in *int16* format.

error: Error message if the reprotect operation fails.

Example

The following is a sample for the *ReProtectIntBulk* API.

```
output, rc, err := session.ReProtectIntBulk(input, dataElement,
apgo.UsingExtIV([]byte(externalIv)), apgo.UsingNewExtIV([]byte(newExternalIv)))
```

2.2.73 ReProtectLongBulk API

The *ReProtectLongBulk* API reprotects the data in the *long* format in *bulk* using data type preservation or a No Encryption data element.

func (s *Session) ReProtectLongBulk(data []int64, oldDataElement, newDataElement string, options ...ProtectionOptions) ([]int64, []int16, error)

Parameters

data: Input containing the data to be reprotected in *int64* format.

oldDataElement: Input containing the older data element defined in the policy in *string* format.

newDataElement: Input containing the new data element defined in the policy in *string* format.

options:

- ***apgo.UsingExtIV***: External IV is an optional parameter. It is a buffer containing data that is used as an initialization vector and accepts input in byte format. When the External IV is empty, its value is ignored.
- ***apgo.UsingNewExtIV***: This parameter lets user specify New External IV for the reprotect operation.

Returns

int64: Returns the re-protected data in *int64* format.

int16: Returns the return code array of the reprotect operation in *int16* format.

error: Error message if the reprotect operation fails.

Example

The following is a sample for the *ReProtectLongBulk* API.

```
output, rc, err := session.ReProtectLongBulk(input, dataElement,
apgo.UsingExtIV([]byte(externalIv)), apgo.UsingNewExtIV([]byte(newExternalIv)))
```

2.2.74 ReProtectFloatBulk API

The *ReProtectFloatBulk* API reprotects *float* data in *bulk* using a No Encryption data element.

func (s *Session) ReProtectFloatBulk(data []float32, oldDataElement, newDataElement string, options ...ProtectionOptions) ([]float32, []int16, error)

Parameters

data: Input containing the data to be reprotected in *float32* format.

oldDataElement: Input containing the older data element defined in the policy in *string* format.

newDataElement: Input containing the new data element defined in the policy in *string* format.

options:

- ***apgo.UsingExtIV***: External IV is an optional parameter. It is a buffer containing data that is used as an initialization vector and accepts input in byte format. When the External IV is empty, its value is ignored.
- ***apgo.UsingNewExtIV***: This parameter lets user specify New External IV for the reprotect operation.

Returns

float32: Returns the reprotected data in *float32* format.

int16: Returns the return code array of the reprotect operation in *int16* format.

error: Error message if the re-protect operation fails.

Example

The following is a sample for the *ReProtectFloatBulk* API.

```
output, rc, err := session.ReProtectFloatBulk(input, dataElement,
apgo.UsingExtIV([]byte(externalIv)), apgo.UsingNewExtIV([]byte(newExternalIv)))
```

2.2.75 ReProtectDoubleBulk API

The *ReProtectDoubleBulk* API reprotects the data in the *double* format in *bulk* using a No Encryption data element.

func (s *Session) ReProtectDoubleBulk(data []float64, oldDataElement, newDataElement string, options ...ProtectionOptions) ([]float64, []int16, error)

Parameters

data: Input containing the data to be re-protected in *float64* format.

oldDataElement: Input containing the older data element defined in the policy in *string* format.

newDataElement: Input containing the new data element defined in the policy in *string* format.

options:

- ***apgo.UsingExtIV***: External IV is an optional parameter. It is a buffer containing data that is used as an initialization vector and accepts input in byte format. When the External IV is empty, its value is ignored.
- ***apgo.UsingNewExtIV***: This parameter lets user specify New External IV for the reprotect operation.

Returns

float64: Returns the reprotected data in *float64* format.

int16: Returns the return code array of the reprotect operation in *int16* format.

error: Error message if the re-protect operation fails.

Example

The following is a sample for the *ReProtectDoubleBulk* API.

```
output, rc, err := session.ReProtectDoubleBulk(input, dataElement,
apgo.UsingExtIV([]byte(externalIv)), apgo.UsingNewExtIV([]byte(newExternalIv)))
```

2.2.76 ReEncryptBytesBulk API

The *ReEncryptBytesBulk* API re-encrypts the data in the *byte* format in *bulk* using an encryption data element.

func (s *Session) ReEncryptBytesBulk(data [][]byte, oldDataElement, newDataElement string, options ...ProtectionOptions) ([][]byte, []int16, error)

Parameters

data: Input containing the data to be re-encrypt in *byte* format.

oldDataElement: Input containing the older data element defined in the policy in *string* format.

newDataElement: Input containing the new data element defined in the policy in *string* format.

options:

- ***apgo.UsingExtIV***: External IV is an optional parameter. It is a buffer containing data that is used as an initialization vector and accepts input in byte format. When the External IV is empty, its value is ignored.

> **Note:**

> Encryption data elements do not support external IV.

- ***apgo.UsingExtTweak***: External Tweak is an optional parameter that is used only for the FPE data elements. It is a buffer containing data that is used as an External Tweak and accepts input in byte format. When the External Tweak is empty, its value is ignored.
- ***apgo.UsingNewExtIV***: This parameter lets user specify New External IV for the reprotect operation.

  > **Note:**
  > Encryption data elements do not support external IV.

- ***apgo.UsingNewExtTweak***: This parameter lets user specify New External Tweak for the reprotect operation.

Returns

byte: Returns the re-encrypted data in *byte* format.

int16: Returns the return code array of the re-encrypt operation in *int16* format.

error: Error message if the re-encrypt operation fails.

Example

The following is a sample for the *ReEncryptBytesBulk* API.

```
output, rc, err := session.ReEncryptBytesBulk(input,
dataElement, apgo.UsingExtIV([]byte(externalIv)),
apgo.UsingNewExtIV([]byte(newExternalIv)), apgo.UsingExtTweak([]byte(externaltweak)),
apgo.UsingNewExtTweak([]byte(newExternaltweak)))
```

2.3 Application Protector (AP) Java APIs

A session must be created to run the AP Java. The AP Java accesses the information on the Trusted Application from the policy stored in the memory. If the application is trusted, then the protect, unprotect, or reprotect method is called, one or many times, depending on the data. You can flush the audits after the operation is complete.

> **Note:** The AP Java APIs can be invoked by a valid *Policy User* or a *Trusted Application* user.

> **Note:**
> When a short running application has completed its execution (in less than a second), the audit logs will not be seen. In such cases, the *protector.flushAudits()* API needs to be invoked.
>
> It is recommended to invoke the *flushAudits* API at the point where the application exits.
>
> For more information about flushAudits, refer to the section *flushAudits*.

The following diagram represents the basic flow of a session.

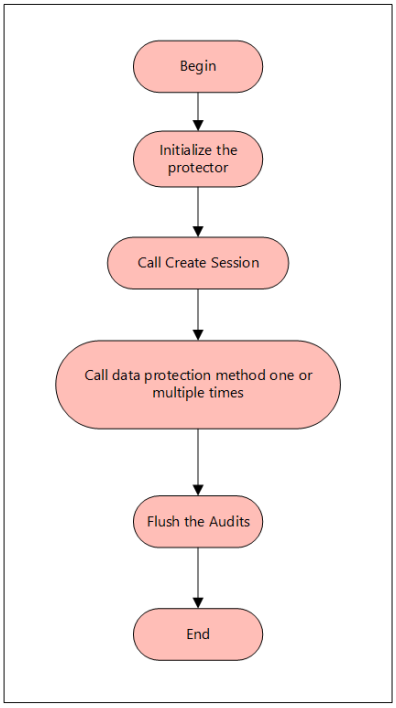

Figure 2-3: Flowchart for AP Java

Warning:

The Protegrity AP Java protector only supports *bytes* converted from the *string* data type.

If any other data type is directly converted to *bytes* and passed as an input to the API that supports *byte* as an input and provides *byte* as an output, then data corruption might occur.

The following sections provide detailed information for the various methods used by the Protegrity Application Protector Java.

2.3.1 Supported Data Types for AP Java

This section lists the data types supported by the AP Java.

The AP Java supports the following data types:

- String
- Integer
- Short
- Long
- Float
- Double
- Bytes
- java.util.Date

2.3.2 getProtector

The *getProtector* method returns the Protector object associated with the Protegrity Application Protector API. After instantiation, this object is used to create a session. The session is passed as a parameter to protect, unprotect, or reprotect methods.

static Protector getProtector()

Parameters

None

Returns

Protector Object: Object associated with the Protegrity Application Protector API

Exception (and Error Codes)

ProtectorException: If the configuration is invalid

2.3.3 getVersion

The *getVersion* method returns the version of the Application Protector Java in use.

public java.lang.String getVersion()

Parameters

None

Returns

String: Product version

2.3.4 getVersionEx

The *getVersionEx* method returns the extended version of the Application Protector Java in use. The extended version consists of the Product version number and the Core version number.

public java.lang.String getVersionEx()

Parameters

None

Returns

String: Product version and Core version

2.3.5 getLastError

The *getLastError* method returns the last error and a description of why this error was returned. When the methods used for protecting, unprotecting, or reprotecting data return an exception or a Boolean false, the *getLastError* method is called to provide a description of why the method failed.

public java.lang.String getLastError(SessionObject session)

Parameters

Session: Session ID that is obtained by calling the *createSession* method

Returns

String: Error message

Exception (and Error Codes)

ProtectorException: If the *SessionObject* is null, then an exception is thrown

SessionTimeoutException: If the session is invalid or has timed out, then an exception is thrown

For more information about the error codes, refer to the section *Application Protectors API Return Codes* in the *Protegrity Troubleshooting Guide 9.1.0.0*.

2.3.6 createSession

The *createSession* method creates a new session. The sessions that have not been utilized for a while, are automatically removed according to the session timeout parameter defined in the *ApplicationProtectorJava.properties* file.

> **Note:** The methods in the Protector API that take the *SessionObject* as a parameter, might throw an exception *SessionTimeoutException* if the session is invalid or has timed out. The application developers can handle the *SessionTimeoutException* and create a new session with a new *SessionObject*.

public SessionObject createSession(java.lang.String policyUser)

Parameters

`policyUser`: User name defined in the policy, as a string value

Returns

`SessionObject`: Object of the *SessionObject* class

Exception (and Error Codes)

`ProtectionException`: If input is null or empty, then an exception is thrown

2.3.7 closeSession

> **Note:**
> Starting from version 9.0.0.0, the *closeSession* method is deprecated.

The *closeSession* method closes a session.

public void closeSession (SessionObject session)

Parameters

`session`: Session ID that is obtained by calling the *createSession* method

Exception (and Error Codes)

`Protector Exception`: If the input is null or empty, then an exception is thrown

2.3.8 getCurrentKeyIdForDataElement

The *getCurrentKeyIdForDataElement* method returns the key ID for a data element that is passed as an input parameter. The data elements could be of the following encryption method types, such as, 3DES, AES-128, or AES-256.

public int getCurrentKeyIdForDataElement (SessionObject sessionObj, java.lang.String dataElementName)

Parameters

`sessionObj`: Object of the *SessionObject* class obtained by calling the *createSession* method

`dataElementName`: The data element for which the Key ID is required

Returns

`int`: Current Key ID

`-1`: Use the *getLastError* method to identify the cause

Exception (and Error Codes)

`Protector Exception`: If input is null, then an exception is thrown

`SessionTimeoutException`: If the session is invalid or has timed out, then an exception is thrown

2.3.9 getDefaultDataelementName

The *getDefaultDataelementName* method returns the default data element for the policy provided as an input parameter and is used transparently by the AP Java methods. A data element becomes a default for a policy when you select it as default during policy creation.

public java.lang.String getDefaultDataelementName (SessionObject sessionObj, java.lang.String policyName)

Parameters

sessionObj: *SessionObject* that is obtained by calling the *createSession* method

policyName: Policy name for which the default data element needs to be returned

Returns

String: Default data element name

NULL: Use the *getLastError* method to identify the cause

Exception (and Error Codes)

Protector Exception: If the *SessionObject* is null, then an exception is thrown

SessionTimeoutException: If the session is invalid or has timed out, then an exception is thrown

2.3.10 extractKeyIdFromData

The *extractKeyIdFromData* method returns the key ID for the data that has already been protected and is passed in this method as the input parameter. Protection is possible for data types 3DES, AES-128, or AES-256. For example, when you need to rotate the key using the latest key and do not know the key ID used for the previous protection, run this method and the *getCurrentKeyId* method. If the output key IDs differ for these two methods, then it implies that the data is protected with the previously generated key ID. For this method, only the byte data type is supported.

> **Note:** Key IDs can only be obtained from data that has been encrypted using a data element with a key ID.

public int extractKeyIdFromData (SessionObject sessionObj, java.lang.String dataElementName, byte[] input)

Parameters

sessionObj: *SessionObject* that is obtained by calling the *createSession* method

dataElementName: The data element for which the Key ID is required

input: Protected data

Returns

int: Key ID

-1: Use the *getLastError* method to identify the cause

Exception (and Error Codes)

Protector Exception: If the *SessionObject* is null, then an exception is thrown

SessionTimeoutException: If the session is invalid or has timed out, then an exception is thrown

2.3.11 flushAudits

The *flushAudits* method is used for flushing the audit logs at any point in the application. This API is required for a short running process that lasts less than a second, to get the audit logs. It is recommended to invoke it at the point where the application exits.

protector.flushAudits()

Parameters

None

Returns

None

Exception (and Error Codes)

Protector Exception: If the API is unable to flush the audit logs, then an exception is thrown

2.3.12 protect - Short array data

Protects the data provided as a short array that uses the preservation data type or No Encryption data element. It supports the bulk protection. There is no maximum data limit. However, you are recommended to pass not more than 1 MB of input data for each protection call.

> **Note:** When the data type preservation methods, such as, Tokenization and No Encryption, are used to protect data, the output of data protection (protected data) can be stored in the same data type that was used for the input data.

public boolean protect(SessionObject sessionObj, java.lang.String dataElementName, short[] input, short[] output, byte[] externalIv)

Parameters

sessionObj: *SessionObject* that is obtained by calling the *createSession* method

dataElementName: String containing the data element name defined in policy

input: Input array with *short* format data

output: Resultant output array with *short* format data

externalIv: Buffer containing data that will be used as external IV, when externalIv = null, the value is ignored

Result

True: The data is successfully protected

False: The parameters passed are accurate, but the method failed when:

- The protection methods failed to perform the required action
- The data element is null or empty

> **Note:** For more information, such as, a text explanation and reason for the failure, call *getLastError(session)*.

Exception

Protector Exception: If the *SessionObject* is null or if policy is configured to throw exception, then an exception is thrown

SessionTimeoutException

2.3.13 protect - Short array data for encryption

Protects the data provided as a short array that uses an encryption data element. It supports the bulk protection. There is no maximum data limit. However, you are recommended to pass not more than 1 MB of input data for each protection call.

> **Note:** When the encryption method is used to protect data, the output of data protection (protected data) should be stored in *byte[]*.

public boolean protect(SessionObject sessionObj, java.lang.String dataElementName, short[] input, byte[][] output, byte[] externalIv)

Parameters

sessionObj: *SessionObject* that is obtained by calling the *createSession* method

dataElementName: String containing the data element name defined in policy

input: Input array with *short* format data

output: Resultant output array with *byte* format data

externalIv: This is optional. Buffer containing data that will be used as external IV, when externalIv = null, the value is ignored

> **Note:**
> Encryption data elements do not support external IV.

Result

True: The data is successfully protected

False: The parameters passed are accurate, but the method failed when:

- The protection methods failed to perform the required action
- The data element is null or empty

> **Note:** For more information, such as, a text explanation and reason for the failure, call *getLastError(session)*.

Exception

Protector Exception: If the *SessionObject* is null or if policy is configured to throw exception, then an exception is thrown

SessionTimeoutException: If the session is invalid or has timed out, then an exception is thrown

2.3.14 protect - Int array data

Protects the data provided as an int array that uses the preservation data type or No Encryption data element. It supports the bulk protection. However, you are recommended to pass not more than 1 MB of input data for each protection call.

> **Note:** When the data type preservation methods, such as, Tokenization and No Encryption, are used to protect data, the output of data protection (protected data) can be stored in the same data type that was used for the input data.

public boolean protect(SessionObject sessionObj, java.lang.String dataElementName, int[] input, int[] output, byte[] externalIv)

Parameters

sessionObj: *SessionObject* that is obtained by calling the *createSession* method

dataElementName : String containing the data element name defined in policy

input: Input array with *int* data

output: Resultant output array with *int* data

externalIv: Buffer containing data that will be used as external IV, when externalIv = null, the value is ignored

Result

True: The data is successfully protected

False: The parameters passed are accurate, but the method failed when:

- The protection methods failed to perform the required action
- The data element is null or empty

> **Note:** For more information, such as, a text explanation and reason for the failure, call *getLastError(session)*.

Exception

ProtectorException: If the *SessionObject* is null or if the policy is configured to throw exception, then an exception is thrown

SessionTimeoutException: If the session is invalid or has timed out, then an exception is thrown

2.3.15 protect - Int array data for encryption

Protects the data provided as an int array that uses an encryption data element. It supports the bulk protection. There is no maximum data limit. However, you are recommended to pass not more than 1 MB of input data for each protection call.

> **Note:**
> You cannot move the data that is protected using encryption data elements with input as integers, long, or short data types and output as bytes, between platforms having different endianness.
>
> For example, if the data is protected using encryption data elements with input as integers and output as bytes, then you cannot move the protected data from the AIX platform to the Linux or Windows platform and vice versa.

> **Note:** When the encryption method is used to protect data, the output of data protection (protected data) should be stored in byte[].

public boolean protect(SessionObject sessionObj, java.lang.String dataElementName, int[] input, byte[][] output, byte[] externalIv)

Parameters

sessionObj: *SessionObject* that is obtained by calling the *createSession* method

dataElementName: String containing the data element name defined in policy

input: Input array with *int* format data

output: Resultant output array with *byte* format data

externalIv: This is optional. Buffer containing data that will be used as external IV, when externalIv = null, the value is ignored

> **Note:**
> Encryption data elements do not support external IV.

Result

True: The data is successfully protected

False: The parameters passed are accurate, but the method failed when:

- The protection methods failed to perform the required action
- The data element is null or empty

> **Note:** For more information, such as, a text explanation and reason for the failure, call *getLastError(session)*.

Exception

ProtectorException: If the *SessionObject* is null or if policy is configured to throw exception, then an exception is thrown

SessionTimeoutException: If the session is invalid or has timed out, then an exception is thrown

2.3.16 protect - Long array data

Protects the data provided as a long array that uses the preservation data type or No Encryption data element. It supports the bulk protection. There is no maximum data limit. However, you are recommended to pass not more than 1 MB of input data for each protection call.

> **Note:** When the data type preservation methods, such as, Tokenization and No Encryption, are used to protect data, the output of data protection (protected data) can be stored in the same data type that was used for the input data.

public boolean protect(SessionObject sessionObj, java.lang.String dataElementName, long[] input, long[] output, byte[] externalIv)

Parameters

sessionObj: *SessionObject* that is obtained by calling the *createSession* method

dataElementName: String containing the data element name defined in policy

input: Input array with *long* format data

output: Resultant output array with *long* format data

externalIv: This is optional. Buffer containing data that will be used as external IV, when externalIv = null, the value is ignored

Result

True: The data is successfully protected

False: The parameters passed are accurate, but the method failed when:

- The protection methods failed to perform the required action
- The data element is null or empty

> **Note:** For more information, such as, a text explanation and reason for the failure, call *getLastError(session)*.

Exception

ProtectorException: If the *SessionObject* is null or if policy is configured to throw exception, then an exception is thrown

SessionTimeoutException: If the session is invalid or has timed out, then an exception is thrown

2.3.17 protect - Long array data for encryption

Protects the data provided as a long array that uses an encryption data element. It supports bulk protection. There is no maximum data limit. However, you are recommended to pass not more than 1 MB of input data for each protection call.

> **Note:** When the encryption method is used to protect data, the output of data protection (protected data) should be stored in *byte[]*.

protect(SessionObject sessionObj, java.lang.String dataElementName, long[] input, byte[][] output)

Parameters

sessionObj: *SessionObject* that is obtained by calling the *createSession* method

dataElementName: String containing the data element name defined in policy

input: Input array with *long* format data

output: Resultant output array with *byte* format data

externalIv: This is optional. Buffer containing data that will be used as external IV, when externalIv = null, the value is ignored

> **Note:**

> Encryption data elements do not support external IV.

Result

`True`: The data is successfully protected

`False`: The parameters passed are accurate, but the method failed when:

- The protection methods failed to perform the required action
- The data element is null or empty

> **Note:** For more information, such as, a text explanation and reason for the failure, call *getLastError(session)*.

Exception (and Error Codes)

`ProtectorException`: If the *SessionObject* is null or if policy is configured to throw exception, then an exception is thrown

`SessionTimeoutException`: If the session is invalid or has timed out, then an exception is thrown

2.3.18 protect - Float array data

Protects the data provided as a float array that uses the No Encryption data element. It supports bulk protection. There is no maximum data limit. However, you are recommended to pass not more than 1 MB of input data for each protection call.

> **Note:** When the data type preservation methods, such as, Tokenization and No Encryption, are used to protect data, the output of data protection (protected data) can be stored in the same data type that was used for input data.

public boolean protect(SessionObject sessionObj, java.lang.String dataElementName, float[] input, float[] output, byte[] externalIv)

Parameters

`sessionObj`: *SessionObject* that is obtained by calling the *createSession* method

`dataElementName`: String containing the data element name defined in policy

`input`: Input array with *float* format data

`output`: Resultant output array with *float* format data

Result

`True`: The data is successfully protected

`False`: The parameters passed are accurate, but the method failed when:

- The protection methods failed to perform the required action
- The data element is null or empty

> **Note:** For more information, such as, a text explanation and reason for the failure, call *getLastError(session)*.

Exception

`ProtectorException`: If the *SessionObject* is null or if policy is configured to throw exception, then an exception is thrown

`SessionTimeoutException`: If the session is invalid or has timed out, then an exception is thrown

2.3.19 protect - Float array data for encryption

Protects the data provided as a float array that uses an encryption data element. It supports bulk protection. There is no maximum data limit. However, you are recommended to pass not more than 1 MB of input data for each protection call.

> **Note:** When the encryption method is used to protect data, the output of data protection (protected data) should be stored in *byte[]*.

public boolean protect(SessionObject sessionObj, java.lang.String dataElementName, float[] input, byte[][] output)

Parameters

sessionObj: *SessionObject* that is obtained by calling the *createSession* method

dataElementName: String containing the data element name defined in policy

input: Input array with *float* format data

output: Resultant output array with *byte* format data

Result

True: The data is successfully protected

False: The parameters passed are accurate, but the method failed when:

- The protection methods failed to perform the required action
- The data element is null or empty

> **Note:** For more information, such as, a text explanation and reason for the failure, call *getLastError(session)*.

Exception

ProtectorException: If the *SessionObject* is null or if policy is configured to throw exception, then an exception is thrown

SessionTimeoutException: If the session is invalid or has timed out, then an exception is thrown

2.3.20 protect - Double array data

Protects the data provided as a double array that uses the No Encryption data element. It supports bulk protection. There is no maximum data limit. However, you are recommended to pass not more than 1 MB of input data for each protection call.

> **Note:** When the data type preservation methods, such as, Tokenization and No Encryption, are used to protect data, the output of data protection (protected data) can be stored in the same data type that was used for input data.

public boolean protect(SessionObject sessionObj, java.lang.String dataElementName, double[] input, double[] output)

Parameters

sessionObj: *SessionObject* that is obtained by calling the *createSession* method

dataElementName: String containing the data element name defined in policy

input: Input array with *double* format data

output: Resultant output array with *double* format data

Result

True: The data is successfully protected

False: The parameters passed are accurate, but the method failed when:

- The protection methods failed to perform the required action

- The data element is null or empty

> **Note:** For more information, such as, a text explanation and reason for the failure, call *getLastError(session)*.

Exception

Protector Exception: If the *SessionObject* is null or if policy is configured to throw exception, then an exception is thrown

SessionTimeoutException: If the session is invalid or has timed out, then an exception is thrown

2.3.21 protect - Double array data for encryption

Protects the data provided as a double array that uses an encryption data element. It supports bulk protection. There is no maximum data limit. However, you are recommended to pass not more than 1 MB of input data for each protection call.

> **Note:** When the encryption method is used to protect data, the output of data protection (protected data) should be stored in *byte[]*.

public boolean protect(SessionObject sessionObj, java.lang.String dataElementName, double[] input, byte[][] output)

Parameters

sessionObj: *SessionObject* that is obtained by calling the *createSession* method

dataElementName: String containing the data element name defined in policy

input: Input array with *double* format data

output: Resultant output array with *byte* format data

Result

True: The data is successfully protected

False: The parameters passed are accurate, but the method failed when:

- The protection methods failed to perform the required action
- The data element is null or empty

> **Note:** For more information, such as, a text explanation and reason for the failure, call *getLastError(session)*.

Exception

ProtectorException: If the *SessionObject* is null or if policy is configured to throw exception, then an exception is thrown

SessionTimeoutException: If the session is invalid or has timed out, then an exception is thrown

2.3.22 protect - Date array data

Protects the data provided as a java.util.Data array that uses a preservation data type. It supports bulk protection. There is no maximum data limit. However, you are recommended to pass not more than 1 MB of input data for each protection call.

> **Note:** When the data type preservation methods, such as, Tokenization and No Encryption, are used to protect data, the output of data protection (protected data) can be stored in the same data type that was used for the input data.

public boolean protect(SessionObject sessionObj, java.lang.String dataElementName, java.util.Date[] input, java.util.Date[] output)

Parameters

sessionObj: *SessionObject* that is obtained by calling the *createSession* method

dataElementName: String containing the data element name defined in policy

input: Input array with *date* format data

output: Resultant output array with *date* format data

Result

True: The data is successfully protected

False: The parameters passed are accurate, but the method failed when:

- The protection methods failed to perform the required action
- The data element is null or empty

> **Note:** For more information, such as, a text explanation and reason for the failure, call *getLastError(session)*.

Exception

ProtectorException: If the *SessionObject* is null or if policy is configured to throw exception, then an exception is thrown

SessionTimeoutException: If the session is invalid or has timed out, then an exception is thrown

2.3.23 protect - String array data

Protects the data provided as a string array that uses a preservation data type or the No Encryption data element. It supports the bulk protection. There is no maximum data limit. However, you are recommended to pass not more than 1 MB of input data for each protection call.

For String and Byte data types, the maximum length for tokenization is 4096 bytes, while for encryption there is no maximum length defined.

> **Note:** When the data type preservation methods, such as, Tokenization and No Encryption, are used to protect the data, the output of data protection (protected data) can be stored in the same data type that was used for the input data.

> **Warning:**
> For Date and DateTime type of data elements, the protect API returns an invalid input data error if the input value falls between the non-existent date range from 05-OCT-1582 to 14-OCT-1582 of the Gregorian Calendar.
>
> For more information about the tokenization and de-tokenization of the cutover dates of the Proleptic Gregorian Calendar, refer to the sections *Date Tokenization for cutover Dates of the Proleptic Gregorian Calendar* and *Datetime Tokenization for Cutover Dates of the Proleptic Gregorian Calendar* in the *Protection Method Reference Guide 9.1.0.0*.

public boolean protect(SessionObject sessionObj, java.lang.String dataElementName, java.lang.String[] input, java.lang.String[] output, byte[] externalIv)

Parameters

sessionObj: *SessionObject* that is obtained by calling the *createSession* method

dataElementName: String containing the data element name defined in policy

input: Input array with *string* format data

output: Resultant output array with *string* format data

externalIv: This is optional. Buffer containing data that will be used as external IV, when externalIv = null, the value is ignored

Returns

True: The data is successfully protected

False: The parameters passed are accurate, but the method failed when:

- The protection methods failed to perform the required action
- The data element is null or empty

> **Note:** For more information, such as, a text explanation and reason for the failure, call *getLastError(session)*.

Exception

ProtectorException: If the *SessionObject* is null or if policy is configured to throw exception, then an exception is thrown

SessionTimeoutException: If the session is invalid or has timed out, then an exception is thrown

2.3.24 protect - String array data for encryption

Protects the data provided as s string array that uses an encryption data element. It supports the bulk protection. There is no maximum data limit. However, you are recommended to pass not more than 1 MB of input data for each protection call.

For String and byte data types, the maximum length for tokenization is 4096 bytes, while for encryption there is no maximum length defined.

> **Note:** When encryption method is used to protect data or Format Preserving Encryption (FPE) for data type preservation methods for String and Char APIs is used, the output of data protection (protected data) should be stored in *byte[]*.

> **Note:** The string as an input and byte as an output API is unsupported by Unicode Gen2 and FPE data elements for the AP Java.

public boolean protect(SessionObject sessionObj, java.lang.String dataElementName, java.lang.String[] input, byte[][] output, byte[] externalIv)

Parameters

sessionObj: *SessionObject* that is obtained by calling the *createSession* method

dataElementName: String containing the data element name defined in policy

input: Input array with *string* format data

output: Resultant output array with *byte* format data

externalIv: This is optional. Buffer containing data that will be used as external IV, when externalIv = null, the value is ignored

> **Note:**
> Encryption data elements do not support external IV.

Returns

True: The data is successfully protected

False: The parameters passed are accurate, but the method failed when:

- The protection methods failed to perform the required action
- The data element is null or empty

> **Note:** For more information, such as, a text explanation and reason for the failure, call *getLastError(session)*.

Exception

ProtectorException: If the *SessionObject* is null or if policy is configured to throw exception, then an exception is thrown

SessionTimeoutException: If the session is invalid or has timed out, then an exception is thrown

2.3.25 protect - Char array data

Protects the data provided as a char array that uses a preservation data type or the No Encryption data element. It supports the bulk protection. There is no maximum data limit. However, you are recommended to pass not more than 1 MB of input data for each protection call.

> **Note:** When the data type preservation methods, such as, Tokenization and No Encryption, are used to protect the data, the output of data protection (protected data) can be stored in the same data type that was used for the input data.

> **Warning:**
> For Date and DateTime type of data elements, the protect API returns an invalid input data error if the input value falls between the non-existent date range from 05-OCT-1582 to 14-OCT-1582 of the Gregorian Calendar.
>
> For more information about the tokenization and de-tokenization of the cutover dates of the Proleptic Gregorian Calendar, refer to the sections *Date Tokenization for cutover Dates of the Proleptic Gregorian Calendar* and *Datetime Tokenization for Cutover Dates of the Proleptic Gregorian Calendar* in the *Protection Method Reference Guide 9.1.0.0*.

public boolean protect(SessionObject sessionObj, java.lang.String dataElementName, char[][] input, char[][] output, byte[] externalIv)

Parameters

sessionObj: *SessionObject* that is obtained by calling the *createSession* method

dataElementName: String containing the data element name defined in policy

input: Input array with *char* format data

output: Resultant output array with *char* format data

externalIv: This is optional. Buffer containing data that will be used as external IV, when externalIv = null, the value is ignored

Returns

True: The data is successfully protected

False: The parameters passed are accurate, but the method failed when:

- The protection methods failed to perform the required action
- The data element is null or empty

> **Note:** For more information, such as, a text explanation and reason for the failure, call *getLastError(session)*.

Exception

ProtectorException: If the *SessionObject* is null or if policy is configured to throw exception, then an exception is thrown

SessionTimeoutException: If the session is invalid or has timed out, then an exception is thrown

2.3.26 protect - Char array data for encryption

Protects the data provided as a char array that uses an encryption data element. It supports the bulk protection. There is no maximum data limit. However, you are recommended to pass not more than 1 MB of input data for each protection call.

> **Note:** When the encryption method is used to protect data or Format Preserving Encryption (FPE) for data type preservation methods for String and Char APIs is used, the output of data protection (protected data) should be stored in *byte[]*.

public boolean protect(SessionObject sessionObj, java.lang.String dataElementName, char[][] input, byte[][] output, byte[] externalIv)

Parameters

sessionObj: *SessionObject* that is obtained by calling the *createSession* method

dataElementName: String containing the data element name defined in policy

input: Input array with *char* format data

output: Resultant output array with *byte* format data

externalIv: This is optional. Buffer containing data that will be used as external IV, when externalIv = null, the value is ignored

> **Note:**
> Encryption data elements do not support external IV.

Result

True: The data is successfully protected

False: The parameters passed are accurate, but the method failed when:

- The protection methods failed to perform the required action
- The data element is null or empty

> **Note:** For more information, such as, a text explanation and reason for the failure, call *getLastError(session)*.

Exception

ProtectorException: If the *SessionObject* is null or if policy is configured to throw exception, then an exception is thrown

SessionTimeoutException: If the session is invalid or has timed out, then an exception is thrown

2.3.27 protect - Byte array data

Protects the data provided as a byte array that uses the encryption data element, No Encryption data element, and preservation data type. It supports the bulk protection. There is no maximum data limit. However, you are recommended to pass not more than 1 MB of input data for each protection call.

For String and byte data types, the maximum length for tokenization is 4096 bytes, while for encryption there is no maximum length defined.

> **Note:** When the data type preservation methods, such as, Tokenization and No Encryption, are used to protect data, the output of data protection (protected data) can be stored in the same data type that was used for input data.

> **Warning:**
> For Date and DateTime type of data elements, the protect API returns an invalid input data error if the input value falls between the non-existent date range from 05-OCT-1582 to 14-OCT-1582 of the Gregorian Calendar.
>
> For more information about the tokenization and de-tokenization of the cutover dates of the Proleptic Gregorian Calendar, refer to the sections *Date Tokenization for cutover Dates of the Proleptic Gregorian Calendar* and *Datetime Tokenization for Cutover Dates of the Proleptic Gregorian Calendar* in the *Protection Method Reference Guide 9.1.0.0*.

public boolean protect(SessionObject sessionObj, java.lang.String dataElementName, byte[][] input, byte[][] output, byte[] externalIv)

Parameters

`sessionObj`: *SessionObject* that is obtained by calling the *createSession* method

`dataElementName`: String containing the data element name defined in policy

`input`: Input array with *byte* format data

`output`: Resultant output array with *byte* format data

`externalIv`: This is optional. Buffer containing data that will be used as external IV, when externalIv = null, the value is ignored

> **Note:** The Protegrity AP Java protector only supports bytes converted from the *string* data type.
>
> If any other data type is directly converted to *bytes* and passed as input to the API that supports *byte* as input and provides *byte* as output, then data corruption might occur.

Result

`True`: The data is successfully protected

`False`: The parameters passed are accurate, but the method failed when:

- The protection methods failed to perform the required action
- The data element is null or empty

> **Note:** For more information, such as, a text explanation and reason for the failure, call *getLastError(session)*.

Exception

`ProtectorException`: If the *SessionObject* is null or if policy is configured to throw exception, then an exception is thrown

`SessionTimeoutException`: If the session is invalid or has timed out, then an exception is thrown

2.3.28 protect - String array data with External Tweak

Protects the data provided as a string array using the FPE (FF1) that uses a preservation data type with FPE data elements. It supports the bulk protection. There is no maximum data limit. However, you are recommended to pass not more than 1 MB of input data for each protection call.

> **Note:** When the data type preservation method, such as FPE, is used with FPE data elements to protect data, the output of data protection (protected data) can be stored in the same data type that was used for the input data.

public boolean protect(SessionObject sessionObj, java.lang.String dataElementName, java.lang.String[] input, java.lang.String[] output, byte[] externalIv, byte[] externalTweak)

Parameters

`sessionObj`: *SessionObject* that is obtained by calling the *createSession* method

`dataElementName`: String containing the data element name defined in policy

`input`: Input array with *string* format data

`output`: Resultant output array with *string* format data

`externalIv`: This is optional. Buffer containing data that will be used as external IV, when externalIv = null, the value is ignored

`externalTweak`: This is optional. Buffer containing data that will be used as Tweak, when *externalTweak* = null, the value is ignored

Result

True: The data is successfully protected

False: The parameters passed are accurate, but the method failed when:

- The protection methods failed to perform the required action
- The data element is null or empty

> **Note:** For more information, such as, a text explanation and reason for the failure, call *getLastError(session)*.

Exception

ProtectorException: If the *SessionObject* is null or if policy is configured to throw exception, then an exception is thrown

SessionTimeoutException: If the session is invalid or has timed out, then an exception is thrown

2.3.29 unprotect - Short array data

Unprotects the data provided as a short array that uses the preservation data type or the No Encryption data element. It supports the bulk unprotection. There is no maximum data limit. However, you are recommended to pass not more than 1 MB of input data for each unprotection call.

public boolean unprotect(SessionObject sessionObj, java.lang.String dataElementName, short[] input, short[] output, byte[] externalIv)

Parameters

sessionObj: *SessionObject* that is obtained by calling the *createSession* method

dataElementName: String containing the data element name defined in policy

input: Input array with *short* format data

output: Resultant output array with *short* format data

externalIv: This is optional. Buffer containing data that will be used as external IV, when externalIv = null, the value is ignored

Result

True: The data is successfully unprotected

False: The parameters passed are accurate, but the method failed to perform the required action

> **Note:** For more information, such as, a text explanation and reason for the failure, call *getLastError(session)*.

Exception

ProtectorException: If the *SessionObject* is null or if policy is configured to throw exception, then an exception is thrown

SessionTimeoutException: If the session is invalid or has timed out, then an exception is thrown

2.3.30 unprotect - Short array data for encryption

Unprotect the data provided as a short array that uses an encryption data element. It supports the bulk unprotection. There is no maximum data limit. However, you are recommended to pass not more than 1 MB of input data for each unprotection call.

public boolean unprotect(SessionObject sessionObj, java.lang.String dataElementName, byte[][] input, short[] output, byte[] externalIv)

Parameters

sessionObj: *SessionObject* that is obtained by calling the *createSession* method

dataElementName: String containing the data element name defined in policy

input: Input array with *byte* format data

output: Resultant output array with *short* format data

externalIv: This is optional. Buffer containing data that will be used as external IV, when externalIv = null, the value is ignored

> **Note:**
> Encryption data elements do not support external IV.

Result

True: The data is successfully unprotected

False: The parameters passed are accurate, but the method failed to perform the required action

> **Note:** For more information, such as, a text explanation and reason for the failure, call *getLastError(session)*.

Exception

ProtectorException: If the *SessionObject* is null or if policy is configured to throw exception, then an exception is thrown

SessionTimeoutException: If the session is invalid or has timed out, then an exception is thrown

2.3.31 unprotect - Int array data

Unprotects the data provided as an int array that uses a preservation data type or a No Encryption data element. It supports the bulk unprotection. There is no maximum data limit. However, you are recommended to pass not more than 1 MB of input data for each unprotection call.

public boolean unprotect(SessionObject sessionObj, java.lang.String dataElementName, int[] input, int[] output, byte[] externalIv)

Parameters

sessionObj: *SessionObject* that is obtained by calling the *createSession* method

dataElementName: String containing the data element name defined in policy

input: Input array with *int* format data

output: Resultant output array with *int* format data

externalIv: This is optional. Buffer containing data that will be used as external IV, when externalIv = null, the value is ignored

Result

True: The data is successfully unprotected

False: The parameters passed are accurate, but the method failed to perform the required action.

> **Note:** For more information, such as, a text explanation and reason for the failure, call *getLastError(session)*.

Exception

ProtectorException: If the *SessionObject* is null or if policy is configured to throw exception, then an exception is thrown

SessionTimeoutException: If the session is invalid or has timed out, then an exception is thrown

2.3.32 unprotect - Int array data for encryption

Unprotects the data provided as an int array that uses an encryption data element. It supports the bulk unprotection. There is no maximum data limit. However, you are recommended to pass not more than 1 MB of input data for each unprotection call.

public boolean unprotect(SessionObject sessionObj, java.lang.String dataElementName, byte[][] input, int[] output, byte[] externalIv)

Parameters

sessionObj: *SessionObject* that is obtained by calling the *createSession* method

dataElementName: String containing the data element name defined in policy

input: Input array with *byte* format data

output: Resultant output array with *int* format data

externalIv: This is optional. Buffer containing data that will be used as external IV, when externalIv = null, the value is ignored

> **Note:**
> Encryption data elements do not support external IV.

Result

True: The data is successfully unprotected

False: The parameters passed are accurate, but the method failed to perform the required action

> **Note:** For more information, such as, a text explanation and reason for the failure, call *getLastError(session)*.

Exception

ProtectorException: If the *SessionObject* is null or if policy is configured to throw exception, then an exception is thrown

SessionTimeoutException: If the session is invalid or has timed out, then an exception is thrown

2.3.33 unprotect - Long array data

Unprotects the data provided as a long array that uses the preservation data type or the No Encryption data element. It supports the bulk unprotection. However, you are recommended to pass not more than 1 MB of input data for each unprotection call.

public boolean unprotect(SessionObject sessionObj, java.lang.String dataElementName, long[] input, long[] output, byte[] externalIv)

Parameters

sessionObj: *SessionObject* that is obtained by calling the *createSession* method

dataElementName: String containing the data element name defined in policy

input: Input array with *long* format data

output: Resultant output array with *long* format data

externalIv: This is optional. Buffer containing data that will be used as external IV, when externalIv = null, the value is ignored

Result

True: The data is successfully unprotected

False: The parameters passed are accurate, but the method failed to perform the required action

> **Note:** For more information, such as, a text explanation and reason for the failure, call *getLastError(session)*.

Exception

ProtectorException: If the *SessionObject* is null or if policy is configured to throw exception, then an exception is thrown

SessionTimeoutException: If the session is invalid or has timed out, then an exception is thrown

2.3.34 unprotect - Long array data for encryption

Unprotects the data provided as a long array that uses an encryption data element. It supports the bulk unprotection. There is no maximum data limit. However, you are recommended to pass not more than 1 MB of input data for each unprotection call.

public boolean unprotect(SessionObject sessionObj, java.lang.String dataElementName, byte[][] input, long[] output, byte[] externalIv)

Parameters

sessionObj: *SessionObject* that is obtained by calling the *createSession* method

dataElementName: String containing the data element name defined in policy

input: Input array with *byte* format data

output: Resultant output array with *long* format data

externalIv: This is optional. Buffer containing data that will be used as external IV, when externalIv = null, the value is ignored

> **Note:**
> Encryption data elements do not support external IV.

Result

True: The data is successfully unprotected

False: The parameters passed are accurate, but the method failed to perform the required action

> **Note:** For more information, such as, a text explanation and reason for the failure, call *getLastError(session)*.

Exception

ProtectorException: If the *SessionObject* is null or if policy is configured to throw exception, then an exception is thrown

SessionTimeoutException: If the session is invalid or has timed out, then an exception is thrown

2.3.35 unprotect - Float array data

Unprotects the data provided as a float array that uses a No Encryption data element. It supports the bulk unprotection. There is no maximum data limit. However, you are recommended to pass not more than 1 MB of input data for each unprotection call.

public boolean unprotect(SessionObject sessionObj, java.lang.String dataElementName, float[] input, float[] output)

Parameters

sessionObj: *SessionObject* that is obtained by calling the *createSession* method

dataElementName: String containing the data element name defined in policy

input: Input array with *float* format data

output: Resultant output array with *float* format data

Result

True: The data is successfully unprotected

False: The parameters passed are accurate, but the method failed to perform the required action

> **Note:** For more information, such as, a text explanation and reason for the failure, call *getLastError(session)*.

Exception

ProtectorException: If the *SessionObject* is null or if policy is configured to throw exception, then an exception is thrown

SessionTimeoutException: If the session is invalid or has timed out, then an exception is thrown

2.3.36 unprotect - Float array data for encryption

Unprotects the data provided as a float array that uses an encryption data element. It supports the bulk unprotection. There is no maximum data limit. However, you are recommended to pass not more than 1 MB of input data for each unprotection call.

public boolean unprotect(SessionObject sessionObj, java.lang.String dataElementName, byte[][] input, float[] output)

Parameters

sessionObj: *SessionObject* that is obtained by calling the *createSession* method

dataElementName: String containing the data element name defined in policy

input: Input array with *byte* format data

output: Resultant output array with *float* format data

Result

True: The data is successfully unprotected

False: The parameters passed are accurate, but the method failed to perform the required action

> **Note:** For more information, such as, a text explanation and reason for the failure, call *getLastError(session)*.

Exception

ProtectorException: If the *SessionObject* is null or if policy is configured to throw exception, then an exception is thrown

SessionTimeoutException: If the session is invalid or has timed out, then an exception is thrown

2.3.37 unprotect - Double array data

Unprotects the data provided as a double array that uses the No Encryption data element. It supports the bulk unprotection. There is no maximum data limit. However, you are recommended to pass not more than 1 MB of input data for each unprotection call.

public boolean unprotect(SessionObject sessionObj, java.lang.String dataElementName, double[] input, double[] output)

Parameters

sessionObj: *SessionObject* that is obtained by calling the *createSession* method

dataElementName: String containing the data element name defined in policy

input: Input array with *double* format data

output: Resultant output array with *double* format data

Result

True: The data is successfully unprotected

False: The parameters passed are accurate, but the method failed to perform the required action

> **Note:** For more information, such as, a text explanation and reason for the failure, call *getLastError(session)*.

Exception

ProtectorException: If the *SessionObject* is null or if policy is configured to throw exception, then an exception is thrown

SessionTimeoutException: If the session is invalid or has timed out, then an exception is thrown

2.3.38 unprotect - Double array data for encryption

Unprotects the data provided as a double array that uses an encryption data element. It supports the bulk unprotection. There is no maximum data limit. However, you are recommended to pass not more than 1 MB of input data for each unprotection call.

public boolean unprotect(SessionObject sessionObj, java.lang.String dataElementName, byte[][] input, double[] output)

Parameters

sessionObj: *SessionObject* that is obtained by calling the *createSession* method

dataElementName: String containing the data element name defined in policy

input: Input array with *byte* format data

output: Resultant output array with *double* format data

Result

True: The data is successfully unprotected

False: The parameters passed are accurate, but the method failed to perform the required action

> **Note:** For more information, such as, a text explanation and reason for the failure, call *getLastError(session)*.

Exception

ProtectorException: If the *SessionObject* is null or if policy is configured to throw exception, then an exception is thrown

SessionTimeoutException: If the session is invalid or has timed out, then an exception is thrown

2.3.39 unprotect - Date array data

Unprotects the data provided as a java.util.Date array using the preservation data type. It supports the bulk unprotection. There is no maximum data limit. However, you are recommended to pass not more than 1 MB of input data for each unprotection call.

public boolean unprotect(SessionObject sessionObj, java.lang.String dataElementName, java.util.Date[] input, java.util.Date[] output)

Parameters

sessionObj: *SessionObject* that is obtained by calling the *createSession* method

dataElementName: String containing the data element name defined in policy

input: Input array with *date* format data

output: Resultant output array with *date* format data

Result

True: The data is successfully unprotected

False: The parameters passed are accurate, but the method failed to perform the required action

> **Note:** For more information, such as, a text explanation and reason for the failure, call *getLastError(session)*.

Exception

ProtectorException: If the *SessionObject* is null or if policy is configured to throw exception, then an exception is thrown

SessionTimeoutException: If the session is invalid or has timed out, then an exception is thrown

2.3.40 unprotect - String array data

Unprotects the data provided as a string array that uses a preservation data type or a No Encryption data element. It supports the bulk unprotection. There is no maximum data limit. However, you are recommended to pass not more than 1 MB of input data for each unprotection call.

public boolean unprotect(SessionObject sessionObj, java.lang.String dataElementName, java.lang.String[] input, java.lang.String[] output, byte[] externalIv)

Parameters

sessionObj: *SessionObject* that is obtained by calling the *createSession* method

dataElementName: String containing the data element name defined in policy

input: Input array with *string* format data

output: Resultant output array with *string* format data

externalIv: This is optional. Buffer containing data that will be used as external IV, when externalIv = null, the value is ignored

Result

True: The data is successfully unprotected

False: The parameters passed are accurate, but the method failed to perform the required action

> **Note:** For more information, such as, a text explanation and reason for the failure, call *getLastError(session)*.

Exception

ProtectorException: If the *SessionObject* is null or if policy is configured to throw exception, then an exception is thrown

SessionTimeoutException: If the session is invalid or has timed out, then an exception is thrown

2.3.41 unprotect - String array data for encryption

Unprotects the data provided as a string array that uses an encryption data element. It supports the bulk unprotection. There is no maximum data limit. However, you are recommended to pass not more than 1 MB of input data for each unprotection call.

public boolean unprotect(SessionObject sessionObj, java.lang.String dataElementName, byte[][] input, java.lang.String[] output, byte[] externalIv)

Parameters

sessionObj: *SessionObject* that is obtained by calling the *createSession* method

dataElementName: String containing the data element name defined in policy

input: Input array with *byte* format data

output: Resultant output array with *string* format data

externalIv: This is optional. Buffer containing data that will be used as external IV, when externalIv = null, the value is ignored

> **Note:**
> Encryption data elements do not support external IV.

Result

True: The data is successfully unprotected

False: The parameters passed are accurate, but the method failed to perform the required action.

> **Note:** For more information, such as, a text explanation and reason for the failure, call *getLastError(session)*.

Exception

ProtectorException: If the *SessionObject* is null or if policy is configured to throw exception, then an exception is thrown

SessionTimeoutException: If the session is invalid or has timed out, then an exception is thrown

2.3.42 unprotect - Char array data

Unprotects the data provided as a char array that uses a preservation data type or a No Encryption data element. It supports the bulk unprotection. There is no maximum data limit. However, you are recommended to pass not more than 1 MB of input data for each unprotection call.

public boolean unprotect(SessionObject sessionObj, java.lang.String dataElementName, char[][] input, char[][] output, byte[] externalIv)

Parameters

sessionObj: *SessionObject* that is obtained by calling the *createSession* method

dataElementName: String containing the data element name defined in policy

input: Input array with *char* format data

output: Resultant output array with *char* data

externalIv: This is optional. Buffer containing data that will be used as external IV, when externalIv = null, the value is ignored

Result

True: The data is successfully unprotected

False: The parameters passed are accurate, but the method failed to perform the required action

> **Note:** For more information, such as, a text explanation and reason for the failure, call *getLastError(session)*.

Exception

ProtectorException: If the *SessionObject* is null or if policy is configured to throw exception, then an exception is thrown

SessionTimeoutException: If the session is invalid or has timed out, then an exception is thrown

2.3.43 unprotect - Char array data for encryption

Unprotects the data provided as a char array that uses an encryption data element. It supports the bulk unprotection. There is no maximum data limit. However, you are recommended to pass not more than 1 MB of input data for each unprotection call.

public boolean unprotect(SessionObject sessionObj, java.lang.String dataElementName, byte[][] input, char[][] output, byte[] externalIv)

Parameters

sessionObj: *SessionObject* that is obtained by calling the *createSession* method

dataElementName: String containing the data element name defined in policy

input: Input array with *byte* format data

output: Resultant output array with *char* format data

externalIv: This is optional. Buffer containing data that will be used as external IV, when externalIv = null, the value is ignored

> **Note:**
> Encryption data elements do not support external IV.

Result

True: The data is successfully unprotected

False: The parameters passed are accurate, but the method failed to perform the required action.

> **Note:** For more information, such as, a text explanation and reason for the failure, call *getLastError(session)*.

Exception

ProtectorException: If the *SessionObject* is null or if policy is configured to throw exception, then an exception is thrown

SessionTimeoutException: If the session is invalid or has timed out, then an exception is thrown

2.3.44 unprotect - Byte array data

Unprotects the data provided as a byte array that uses an encryption data element or a No Encryption data element, or a preservation data type. It supports the bulk unprotection. There is no maximum data limit. However, you are recommended to pass not more than 1 MB of input data for each unprotection call.

> **Note:**
> The Protegrity AP Java protector only supports *bytes* converted from the *string* data type.
>
> If any other data type is directly converted to *bytes* and passed as input to the API that supports *byte* as input and provides *byte* as output, then data corruption might occur.

public boolean unprotect(SessionObject sessionObj, java.lang.String dataElementName, byte[][] input, byte[][] output, byte[] externalIv)

Parameters

sessionObj: *SessionObject* that is obtained by calling the *createSession* method

dataElementName: String containing the data element name defined in policy

input: Input array with *byte* format data

output: Resultant output array with *byte* format data

externalIv: This is optional. Buffer containing data that will be used as external IV, when externalIv = null, the value is ignored

Result

True: The data is successfully unprotected

False: The parameters passed are accurate, but the method failed to perform the required action

> **Note:** For more information, such as, a text explanation and reason for the failure, call *getLastError(session)*.

Exception

ProtectorException: If the *SessionObject* is null or if policy is configured to throw exception, then an exception is thrown

SessionTimeoutException: If the session is invalid or has timed out, then an exception is thrown

2.3.45 unprotect - String array data with External Tweak

Unprotects the data provided as a string array using the FPE (FF1) that uses a preservation data type with FPE data elements. It supports the bulk unprotection. There is no maximum data limit. However, you are recommended to pass not more than 1 MB of input data for each unprotection call.

public boolean unprotect(SessionObject sessionObj, java.lang.String dataElementName, java.lang.String[] input, java.lang.String[] output, byte[] externalIv, byte[] externalTweak)

Parameters

`sessionObj`: *SessionObject* that is obtained by calling the *createSession* method

`dataElementName`: String containing the data element name defined in policy

`input`: Input array with *string* format data

`output`: Resultant output array with *string* format data

`externalIv`: This is optional. Buffer containing data that will be used as external IV, when *externalIv* = null, the value is ignored

`externalTweak`: This is optional. Buffer containing data that will be used as Tweak, when *externalTweak* = null, the value is ignored

Result

`True`: The data is successfully unprotected

`False`: The parameters passed are accurate, but the method failed to perform the required action

> **Note:** For more information, such as, a text explanation and reason for the failure, call *getLastError(session)*.

Exception

`ProtectorException`: If the *SessionObject* is null or if policy is configured to throw exception, then an exception is thrown

`SessionTimeoutException`: If the session is invalid or has timed out, then an exception is thrown

2.3.46 reprotect - Short array data

Reprotects the data provided as a short array that uses a preservation data type or a No Encryption data element. The protected data is first unprotected and then protected again with a new data element. It supports the bulk reprotection. There is no maximum data limit. However, you are recommended to pass not more than 1 MB of input data for each reprotection call.

> **Warning:** If you are using the reprotect API, then the old data element and the new data element must have the same data type. For example, if you have used Alpha-Numeric data element to protect the data, then you must use only Alpha-Numeric data element to reprotect the data.

public boolean reprotect(SessionObject sessionObj, java.lang.String newDataElementName, java.lang.String oldDataElementName, short[] input, short[] output, byte[] newExternalIv, byte[] oldExternalIv)

Parameters

`sessionObj`: *SessionObject* that is obtained by calling the *createSession* method

`newdataElementName`: String containing the data element name defined in policy to create the output data.

`olddataElementName`: String containing the data element name defined in policy for the input data.

`input`: Input array with *short* format data

`output`: Resultant output array with *short* format data

`newExternalIV`: This is optional. Buffer containing data that will be used as external IV, when *newExternalIv* = null, the value is ignored

`oldExternalIV`: This is optional. Buffer containing data that will be used as external IV on the old data element, when *oldExternalIv* = null, the value is ignored

Result

`True`: The data is successfully reprotected

`False`: The parameters passed are accurate, but the method failed to perform the required action

> **Note:** For more information, such as, a text explanation and reason for the failure, call *getLastError(session)*.

Exception

`ProtectorException`: If the *SessionObject* is null or if policy is configured to throw exception, then an exception is thrown

`SessionTimeoutException`: If the session is invalid or has timed out, then an exception is thrown

2.3.47 reprotect - Int array data

Reprotects the data provided as an int array that uses a preservation data type or a No Encryption data element. The protected data is first unprotected and then protected again with a new data element. It supports the bulk reprotection. There is no maximum data limit. However, you are recommended to pass not more than 1 MB of input data for each reprotection call.

Warning: If you are using the reprotect API, then the old data element and the new data element must have the same data type. For example, if you have used Alpha-Numeric data element to protect the data, then you must use only Alpha-Numeric data element to reprotect the data.

public boolean reprotect(SessionObject sessionObj, java.lang.String newDataElementName, java.lang.String oldDataElementName, int[] input, int[] output, byte[] newExternalIv, byte[] oldExternalIv)

Parameters

`sessionObj`: *SessionObject* that is obtained by calling the *createSession* method

`newdataElementName`: String containing the data element name defined in policy to create the output data

`olddataElementName`: String containing the data element name defined in policy for the input data

`input`: Input array with *int* format data

`output`: Resultant output array with *int* format data

`newExternalIV`: This is optional. Buffer containing data that will be used as external IV, when *newExternalIv* = null, the value is ignored

`oldExternalIV`: This is optional. Buffer containing data that will be used as external IV on the old data element, when *oldExternalIv* = null, the value is ignored

Result

`True`: The data is successfully reprotected

`False`: The parameters passed are accurate, but the method failed to perform the required action

Note: For more information, such as, a text explanation and reason for the failure, call *getLastError(session)*.

Exception

`ProtectorException`: If the *SessionObject* is null or if policy is configured to throw exception, then an exception is thrown

`SessionTimeoutException`: If the session is invalid or has timed out, then an exception is thrown

2.3.48 reprotect - Long array data

Reprotects the data provided as a long array that uses a preservation data type or a No Encryption data element. The protected data is first unprotected and then protected again with a new data element. It supports the bulk reprotection. There is no maximum data limit. However, you are recommended to pass not more than 1 MB of input data for each reprotection call.

Warning: If you are using the reprotect API, then the old data element and the new data element must have the same data type. For example, if you have used Alpha-Numeric data element to protect the data, then you must use only Alpha-Numeric data element to reprotect the data.

public boolean reprotect(SessionObject sessionObj, java.lang.String newDataElementName, java.lang.String oldDataElementName, long[] input, long[] output, byte[] newExternalIv, byte[] oldExternalIv)

Parameters

`sessionObj`: *SessionObject* that is obtained by calling the *createSession* method

`newdataElementName`: String containing the data element name defined in policy to create the output data.

`olddataElementName`: String containing the data element name defined in policy for the input data.

`input`: Input array with *long* format data

`output`: Resultant output array with *long* format data

`newExternalIV`: This is optional. Buffer containing data that will be used as external IV, when *newExternalIv* = null, the value is ignored

`oldExternalIV`: This is optional. Buffer containing data that will be used as external IV on the old data element, when *oldExternalIv* = null, the value is ignored

Result

`True`: The data is successfully reprotected

`False`: The parameters passed are accurate, but the method failed to perform the required action.

> **Note:** For more information, such as, a text explanation and reason for the failure, call *getLastError(session)*.

Exception

`ProtectorException`: If the *SessionObject* is null or if policy is configured to throw exception, then an exception is thrown

`SessionTimeoutException`: If the session is invalid or has timed out, then an exception is thrown

2.3.49 reprotect - Float array data

Reprotects the data provided as a float array that uses a No Encryption data element. The protected data is first unprotected and then protected again with a new data element. It supports the bulk reprotection. There is no maximum data limit. However, you are recommended to pass not more than 1 MB of input data for each reprotection call.

> **Warning:** If you are using the reprotect API, then the old data element and the new data element must have the same data type. For example, if you have used Alpha-Numeric data element to protect the data, then you must use only Alpha-Numeric data element to reprotect the data.

public boolean reprotect(SessionObject sessionObj, java.lang.String newDataElementName, java.lang.String oldDataElementName, float[] input, float[] output)

Parameters

`sessionObj`: *SessionObject* that is obtained by calling the *createSession* method

`newdataElementName`: String containing the data element name defined in policy to create the output data.

`olddataElementName`: String containing the data element name defined in policy for the input data.

`input`: Input array with *float* format data

`output`: Resultant output array with *float* format data

Result

`True`: The data is successfully reprotected

`False`: The parameters passed are accurate, but the method failed to perform the required action

> **Note:** For more information, such as, a text explanation and reason for the failure, call *getLastError(session)*.

Exception

`ProtectorException`: If the *SessionObject* is null or if policy is configured to throw exception, then an exception is thrown

`SessionTimeoutException`: If the session is invalid or has timed out, then an exception is thrown

2.3.50 reprotect - Double array data

Reprotects the data provided as a double array that uses a No Encryption data element. The protected data is first unprotected and then protected again with a new data element. It supports the bulk reprotection. There is no maximum data limit. However, you are recommended to pass not more than 1 MB of input data for each reprotection call.

Warning: If you are using the reprotect API, then the old data element and the new data element must have the same data type. For example, if you have used Alpha-Numeric data element to protect the data, then you must use only Alpha-Numeric data element to reprotect the data.

public boolean reprotect(SessionObject sessionObj, java.lang.String newDataElementName, java.lang.String oldDataElementName, double[] input, double[] output)

Parameters

sessionObj: *SessionObject* that is obtained by calling the *createSession* method

newdataElementName: String containing the data element name defined in policy to create the output data.

olddataElementName: String containing the data element name defined in policy for the input data.

input: Input array with *double* format data

output: Resultant output array with *double* format data

Result

True: The data is successfully reprotected

False: The parameters passed are accurate, but the method failed to perform the required action

Note: For more information, such as, a text explanation and reason for the failure, call *getLastError(session)*.

Exception

ProtectorException: If the *SessionObject* is null or if policy is configured to throw exception, then an exception is thrown

SessionTimeoutException: If the session is invalid or has timed out, then an exception is thrown

2.3.51 reprotect - Date array data

Reprotects the data provided as a date array that uses a preservation data type. The protected data is first unprotected and then protected again with a new data element. It supports the bulk reprotection. There is no maximum data limit. However, you are recommended to pass not more than 1 MB of input data for each reprotection call.

Warning: If you are using the reprotect API, then the old data element and the new data element must have the same data type. For example, if you have used Alpha-Numeric data element to protect the data, then you must use only Alpha-Numeric data element to reprotect the data.

public boolean reprotect(SessionObject sessionObj, java.lang.String newDataElementName, java.lang.String oldDataElementName, java.util.Date[] input, java.util.Date[] output)

Parameters

sessionObj: *SessionObject* that is obtained by calling the *createSession* method

newdataElementName: String containing the data element name defined in policy to create the output data

olddataElementName: String containing the data element name defined in policy for the input data

input: Input array with *date* format data

output: Resultant output array with *date* format data

Result

True: The data is successfully reprotected

False: The parameters passed are accurate, but the method failed to perform the required action

> **Note:** For more information, such as, a text explanation and reason for the failure, call *getLastError(session)*.

Exception

ProtectorException: If the *SessionObject* is null or if policy is configured to throw exception, then an exception is thrown

SessionTimeoutException: If the session is invalid or has timed out, then an exception is thrown

2.3.52 reprotect - Date array data

Reprotects the data provided as a date array that uses a preservation data type. The protected data is first unprotected and then protected again with a new data element. It supports the bulk reprotection. There is no maximum data limit. However, you are recommended to pass not more than 1 MB of input data for each reprotection call.

> **Warning:** If you are using the reprotect API, then the old data element and the new data element must have the same data type. For example, if you have used Alpha-Numeric data element to protect the data, then you must use only Alpha-Numeric data element to reprotect the data.

public boolean reprotect(SessionObject sessionObj, java.lang.String newDataElementName, java.lang.String oldDataElementName, java.util.Date[] input, java.util.Date[] output)

Parameters

sessionObj: *SessionObject* that is obtained by calling the *createSession* method

newdataElementName: String containing the data element name defined in policy to create the output data

olddataElementName: String containing the data element name defined in policy for the input data

input: Input array with *date* format data

output: Resultant output array with *date* format data

Result

True: The data is successfully reprotected

False: The parameters passed are accurate, but the method failed to perform the required action

> **Note:** For more information, such as, a text explanation and reason for the failure, call *getLastError(session)*.

Exception

ProtectorException: If the *SessionObject* is null or if policy is configured to throw exception, then an exception is thrown

SessionTimeoutException: If the session is invalid or has timed out, then an exception is thrown

2.3.53 reprotect - Date array data

Reprotects the data provided as a date array that uses a preservation data type. The protected data is first unprotected and then protected again with a new data element. It supports the bulk reprotection. There is no maximum data limit. However, you are recommended to pass not more than 1 MB of input data for each reprotection call.

> **Warning:** If you are using the reprotect API, then the old data element and the new data element must have the same data type. For example, if you have used Alpha-Numeric data element to protect the data, then you must use only Alpha-Numeric data element to reprotect the data.

public boolean reprotect(SessionObject sessionObj, java.lang.String newDataElementName, java.lang.String oldDataElementName, java.util.Date[] input, java.util.Date[] output)

Parameters

`sessionObj`: *SessionObject* that is obtained by calling the *createSession* method

`newdataElementName`: String containing the data element name defined in policy to create the output data

`olddataElementName`: String containing the data element name defined in policy for the input data

`input`: Input array with *date* format data

`output`: Resultant output array with *date* format data

Result

`True`: The data is successfully reprotected

`False`: The parameters passed are accurate, but the method failed to perform the required action

> **Note:** For more information, such as, a text explanation and reason for the failure, call *getLastError(session)*.

Exception

`ProtectorException`: If the *SessionObject* is null or if policy is configured to throw exception, then an exception is thrown

`SessionTimeoutException`: If the session is invalid or has timed out, then an exception is thrown

2.3.54 reprotect - Byte array data

Reprotects the data provided as a byte array that uses an encryption data element or a No Encryption data element, or a preservation data type. The protected data is first unprotected and then protected again with a new data element. However, you are recommended to pass not more than 1 MB of input data for each reprotection call.

> **Note:** When the data type preservation methods, such as, Tokenization and No Encryption are used to reprotect data, the output of data protection (protected data) can be stored in the same data type that was used for input data.

> **Warning:** If you are using the reprotect API, then the old data element and the new data element must have the same data type. For example, if you have used Alpha-Numeric data element to protect the data, then you must use only Alpha-Numeric data element to reprotect the data.

public boolean reprotect(SessionObject sessionObj, java.lang.String newDataElementName, java.lang.String oldDataElementName, byte[][] input, byte[][] output, byte[] newExternalIv, byte[] oldExternalIv)

Parameters

`sessionObj`: *SessionObject* that is obtained by calling the *createSession* method

`newdataElementName`: String containing the data element name defined in policy to create the output data

`olddataElementName`: String containing the data element name defined in policy for the input data

`input`: Input array with *byte* format data

`output`: Resultant output array with *byte* format data

`newExternalIV`: This is optional. Buffer containing data that will be used as external IV on the new data element, when *newExternalIv* = null, the value is ignored

`oldExternalIV`: This is optional. Buffer containing data that will be used as external IV on the new data element, when *oldExternalIv* = null, the value is ignored

> **Note:** The Protegrity AP Java protector only supports bytes converted from the *string* data type.
>
> If any other data type is directly converted to *bytes* and passed as input to the API that supports *byte* as input and provides *byte* as output, then data corruption might occur.

Result

`True`: The data is successfully reprotected

`False`: The parameters passed are accurate, but the method failed to perform the required action

> **Note:** For more information, such as, a text explanation and reason for the failure, call *getLastError(session)*.

Exception

`ProtectorException`: If the *SessionObject* is null or if policy is configured to throw exception, then an exception is thrown

`SessionTimeoutException`: If the session is invalid or has timed out, then an exception is thrown

2.3.55 reprotect - String array data with External Tweak

Reprotects the data provided as a string array using the FPE (FF1) that uses a preservation data type with FPE data elements. The protected data is first unprotected and then protected again with a new FPE data element. It supports the bulk reprotection. There is no maximum data limit. However, you are recommended to pass not more than 1 MB of input data for each reprotection call.

> **Warning:** If you are using the reprotect API, then the old data element and the new data element must have the same data type. For example, if you have used Alpha-Numeric data element to protect the data, then you must use only Alpha-Numeric data element to reprotect the data.

> **Warning:** If you are using Format Preserving Encryption (FPE) with the reprotect API, then ensure that the plaintext encoding used for FPE must be the same for both protecting and reprotecting the data. For example, if you have used FPE-Numeric data element with UTF 8 encoding to protect the data, then you must use only FPE-Numeric data element with UTF 8 encoding to reprotect the data.

public boolean reprotect(SessionObject sessionObj, java.lang.String newDataElementName, java.lang.String oldDataElementName, java.lang.String[] input, java.lang.String[] output,byte[] newExternalIv,byte[] oldExternalIv,byte[] newExternalTweak, byte[] oldExternalTweak)

Parameters

`sessionObj`: *SessionObject* that is obtained by calling the *createSession* method

`newdataElementName`: String containing the data element name defined in policy to create the output data

`olddataElementName`: String containing the data element name defined in policy for the input data

`input`: Input array with *String* format data

`output`: Resultant output array with *String* format data

`newExternalIV`: This is optional. Buffer containing data that will be used as external IV on the new data element, when *newExternalIv* = null, the value is ignored

`oldExternalIV`: This is optional. Buffer containing data that will be used as external IV on the new data element, when *oldExternalIv* = null, the value is ignored

`newExternalTweak`: This is optional. Buffer containing data that will be used as Tweak on the new data element, when *newExternalTweak* = null, the value is ignored

`oldExternalTweak`: This is optional. Buffer containing data that will be used as Tweak on the new data element, when *oldExternalTweak* = null, the value is ignored

Result

`True`: The data is successfully reprotected

`False`: The parameters passed are accurate, but the method failed to perform the required action

2.4 Application Protector (AP) Python APIs

A session must be created to run the Application Protector (AP) Python. Before creating the session, the AP Python verifies whether the application invoking the AP Python APIs is trusted. If it is trusted, then a new session is created, and the protect,

unprotect, or reprotect methods can be called, one or many times, depending on the data. After the operation is complete, this session closes implicitly or the session times out if it is idle.

The sessions are needed to handle the audit record generation. A session is valid for a specific time, which is managed by the *timeout* value passed during the *create_session()* method. By default, the session timeout value is set to *15* minutes. For every call to the *create_session()* method, a new session object is created - a pool of session objects is not maintained. Python's garbage collector is used for destroying the session objects once they are out of scope. You can also use the session object as Python's Context manager using the *with* statement.

A session is automatically renewed every time it is used. Thus, for each call to a data protection operation, such as, protect, unprotect, and reprotect, the time for the session to remain alive is renewed.

Each session generates an audit record for every new protection method call combined with the data element used. This means that in case of single data item calls, three audit log events will be generated if you perform one protect operation with data element name *a*, five protect operations with data element name *b*, and 1000 unprotect operations with data element *a*. In case of the bulk data items, every data protection operation, such as, protect, unprotect, and reprotect will generate audit log events. You can use this knowledge to ensure how you want the audit records to be generated, in case of single data item calls, by deciding how long a session is valid and how often new sessions are created using the *create_session()* method.

Note: The AP Python APIs can be invoked by an application only if the application and the user running the application are defined as part of a *Trusted Application* in the ESA.

Note: Only users who are defined as valid *Policy Users* in the ESA can perform security operations using the AP Python APIs.

Note: Only users who are defined with the required protect, unprotect, or reprotect permissions in the policy in the ESA, can perform security operations using the AP Python APIs.

The following figure explains a basic flow of a session.

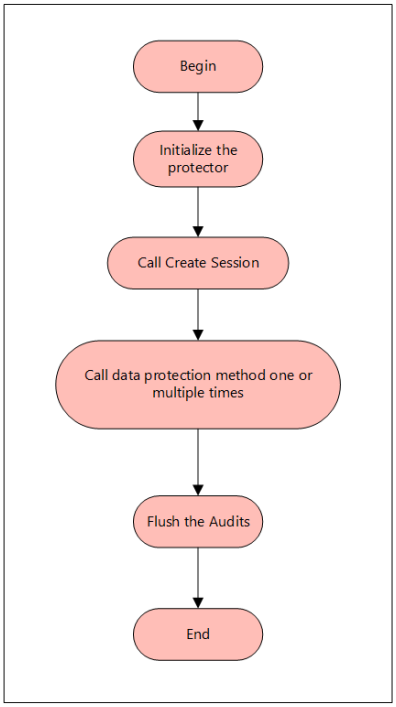

Figure 2-4: Flowchart for the AP Python

Warning:

The Protegrity AP Python protector only supports *bytes* converted from the *string* data type.

If any other data type is directly converted to *bytes* and passed as input to the API that supports *byte* as input and provides *byte* as output, then data corruption might occur.

Important:

You do not have to explicitly close the session. The session is closed implicitly after the API protects, unprotects, or reprotects the data. The *session* object is implemented as a Python Context Manager and can be used with the *with* statement.

2.4.1 Supported Data Types for AP Python

This section lists the data types supported by the AP Python.

The AP Python supports the following data types:

- String
- Integer
- Float
- Bytes
- Date Object

Note: The AP Python does not support the DateTime object.

2.4.2 Supported Modes for AP Python

This section describes the modes supported by the AP Python.

You can use the AP Python APIs in the following modes:

- *Using the AP Python in a Production Environment*: Use the AP Python APIs to protect, unprotect, and reprotect the data using the data elements deployed on the ESA.
- *Using the AP Python in a Development Environment*: Use sample users and data elements with the AP Python APIs to simulate the protect, unprotect, and reprotect operations. You do not require the Log Forwarder, the PEP server, and the ESA to be installed on your machine.

Note: For more information about how to use the AP Python APIs in a development environment for testing purposes, refer to the section *Using AP Python in a Development Environment*.

2.4.3 Using AP Python in a Production Environment

This section provides detailed information of the APIs that are supported by the AP Python in a production environment. It describes the syntax of the AP Python APIs and provides the sample use cases.

2.4.3.1 Initialize the Protector

The *Protector* API returns the Protector object associated with the Protegrity AP APIs. After instantiation, this object is used to create a session. The session object provides APIs to perform the protect, unprotect, or reprotect operations.

Protector(self, comm_id=0)

> **Caution:** Do not pass the *self* parameter while invoking the API.

Parameters

comm_id: The Communication ID that is used by the PEP server. This value must match the value specified by the Communication ID parameter in the *pepserver.cfg* file. This parameter is optional.

Returns

Protector: Object associated with the Protegrity AP Python API.

Exception

InitializationError: This exception is thrown if the protector fails to initialize.

Example

In the following example, the AP Python is initialized.

```
from appython import Protector
protector = Protector()
```

2.4.3.2 create_session

The *create_session* API creates a new session. The sessions that are created using this API, automatically time out after the session timeout value has been reached. The default session timeout value is *15* minutes. However, you can also pass the session timeout value as a parameter to this API.

> **Important:** If the session is invalid or has timed out, then the AP Python APIs that are invoked using this session object, may throw an *InvalidSessionError* exception. Application developers can catch the *InvalidSessionError* exception and create a session by again by invoking the *create_session* API.

def create_session(self, policy_user, timeout=15)

> **Caution:** Do not pass the *self* parameter while invoking the API.

Parameters

policy_user: Username defined in the policy, as a string value.

timeout: Session timeout, specified in minutes. By default, the value of this parameter is set to *15*. This parameter is optional.

Returns

session: Object of the *Session* class. A session object is required for calling the data protection operations, such as, protect, unprotect, and reprotect.

Exception

ProtectorError: This exception is thrown if a null or empty value is passed as the *policy_user* parameter.

Example

In the following example, *User1* is passed as the *policy_user* parameter.

```
from appython import Protector
protector = Protector()
session = protector.create_session("User1")
```

2.4.3.3 get_version

The *get_version* API returns the version of the AP Python in use. Ensure that the version number of the AP Python matches with the PEP server package.

> **Note:**
> You do not need to create a session for invoking the *get_version* API.

def get_version(self)

> **Caution:** Do not pass the *self* parameter while invoking the API.

Parameters

None

Returns

String: Product version of the installed AP Python

Exception

None

Example

In the following example, the current version of the installed AP Python is retrieved.

```
from appython import Protector
protector = Protector()
session = protector.create_session("User1")
print(protector.get_version())
```

Result

```
9.1.0.0.8
```

2.4.3.4 get_version_ex

The *get_version_ex* API returns the extended version of the AP Python in use. The extended version consists of the AP Python version number and the Core version.

> **Note:**
> You do not need to create a session for invoking the *get_version_ex* API.

def get_version_ex(self)

> **Caution:** Do not pass the *self* parameter while invoking the API.

Parameters

None

Returns

String: The product version of the installed AP Python and the Core version.

Exception

None

Example

In the following example, the current version of the AP Python and the Core version is retrieved.

```
from appython import Protector
protector = Protector()
session = protector.create_session("User1")
print(protector.get_version_ex())
```

Result

```
SDK Version: 9.1.0.0.8, Core Version: 1.2.0+81.g1c449.1.2
```

2.4.3.5 get_current_key_id_for_dataelement

The *get_current_key_id_for_dataelement* API returns the key ID for a data element that is passed as an input parameter. The data elements can be of the following encryption method types, such as, 3DES, AES-128, or AES-256.

def get_current_key_id_for_dataelement(self, de)

> **Caution:** Do not pass the *self* parameter while invoking the API.

Parameters

de: String containing the data element name defined in the policy

Returns

int: Returns the current key ID of the specified Encryption data element

Exception

ProtectorError: This exception is thrown if the API is unable to retrieve the key ID.

InvalidSessionError: This exception is thrown if the session is invalid or has timed out.

Example

In the following example, the current key ID for the *AES128* data element is retrieved.

```
from appython import Protector
protector = Protector()
session = protector.create_session("User1")
current_key = session.get_current_key_id_for_dataelement("AES128")
print(current_key)
```

Result

```
115
```

2.4.3.6 extract_key_id_from_data

The *extract_key_id_from_data* API returns the key ID for the data element and the protected data that are passed as input parameters. The data elements can be of type 3DES, AES-128, or AES-256.

> **Note:** If you want to encrypt the data, then you must use bytes in the *encrypt_to* keyword.

def extract_key_id_from_data(self, de, protected_data)

> **Caution:** Do not pass the *self* parameter while invoking the API.

Parameters

de: String containing the data element name defined in the policy

protected_data: Protected data in bytes

Returns

int: Returns the key ID extracted from the protected data and data element that are passed as input parameters

Exception

ProtectorError: This exception is thrown if the API is unable to retrieve the key ID.

InvalidSessionError: This exception is thrown if the session is invalid or has timed out.

Example

In the following example, the *Protegrity1* string is used as the data, which is first encrypted using the *AES256_IV_CRC_KID* data element. The key ID for the *AES256_IV_CRC_KID* data element and the protected data is then retrieved using the *extract_key_id_from_data* API.

```
from appython import Protector
protector = Protector()
session = protector.create_session("User1")
p_out = session.protect("Protegrity1", "AES256_IV_CRC_KID", encrypt_to=bytes)
extracted_key = session.extract_key_id_from_data("AES256_IV_CRC_KID", p_out)
print(extracted_key)
```

Result

```
115
```

2.4.3.7 get_default_de

The *get_default_de* API returns the default data element for the policy provided as an input parameter. A data element becomes a default for a policy when you select it as the default during policy creation.

def get_default_de(self, policyname)

> **Caution:** Do not pass the *self* parameter while invoking the API.

Parameters

policyname: The policy name for which the default data element needs to be returned.

Returns

string: Default data element name

Exception

ProtectorError: This exception is thrown if the API is unable to retrieve the default data element.

InvalidSessionError: This exception is thrown if the session is invalid or has timed out.

Example

In the following example, the default data element for the *Policy_2* policy is retrieved.

```
from appython import Protector
protector = Protector()
session = protector.create_session("User1")
default_de = session.get_default_de("Policy_2")
print(default_de)
```

Result

```
TE_A_S23_L0R0_N
```

2.4.3.8 check_access

The *check_access* API returns the access permission status of the user for a specified data element.

def check_access(self, de, access_type)

> **Caution:** Do not pass the *self* parameter while invoking the API.

Parameters

de: String containing the data element name defined in the policy.

access_type: Type of the access permission of the user for the specified data element. You can specify a value for this parameter from the *CheckAccessType* enumeration.

The following are the different values for the *CheckAccessType* enumeration:

PROTECT	2
REPROTECT	4
UNPROTECT	8

Returns

True: The user has access to the data element.

False: The user does not have access to the data element.

Exception

ProtectorError: This exception is thrown if the API is unable to retrieve the default data element.

InvalidSessionError: This exception is thrown if the session is invalid or has timed out.

Example

In the following example, the *check_access* API is used to check whether the user has reprotect permissions for the *TE_A_N_S23_L2R2_Y* data element.

```
from appython import Protector
from appython import CheckAccessType
protector = Protector()
session = protector.create_session("User1")
print(session.check_access("TE_A_N_S23_L2R2_Y",
      CheckAccessType.REPROTECT))
```

Result

```
True
```

2.4.3.9 flush_audits()

The *flush_audits* API is used for flushing the audit logs at any point within the application. This API is required for a short running process that lasts less than a second, to get the audit logs. It is recommended to invoke the API at the point where the application exits.

def flush_audits(self)

> **Caution:** Do not pass the *self* parameter while invoking the API.

Parameters

None

Returns

None

Exception

ProtectorError: This exception is thrown if the API is unable to flush the audit logs.

Example

In the following example, the *flush_audits* API is used to flush the audit logs.

```
from appython import Protector
protector = Protector()
session = protector.create_session("User1")
output = session.protect("Protegrity1", "TE_A_N_S23_L2R2_Y")
print("Protected Data: %s" %output)
session.flush_audits()
```

2.4.3.10 protect

The *protect* API protects the data using tokenization, data type preserving encryption, No Encryption, or encryption data element. It supports both single and bulk protection without a maximum bulk size limit. However, you are recommended not to pass more than 1 MB of input data for each protection call.

For String and Byte data types, the maximum length for tokenization is 4096 bytes, while no maximum length is defined for encryption.

def protect(self, data, de, **kwargs)

> **Caution:** Do not pass the *self* parameter while invoking the API.

Parameters

data: Data to be protected. You can provide the data of any type that is supported by the AP Python. For example, you can specify data of type string, float, or integer. However, you cannot provide the data of multiple data types at the same time in a bulk call.

de: String containing the data element name defined in policy

*****kwargs***: Specify one or more of the following keyword arguments:

- *external_iv*: Specify the external initialization vector for Tokenization and FPE protection methods. This argument is optional.
- *encrypt_to*: Specify this argument for encrypting the data and set its value to *bytes*. This argument is Mandatory. It must not be used for Tokenization and FPE protection methods.
- *external_tweak*: Specify the external tweak value for FPE protection method. This argument is ptional.

> **Note:** Keyword arguments are case sensitive.

Returns

- *For single data*: Returns the protected data
- *For bulk data*: Returns a tuple of the following data:
  - List or tuple of the protected data
  - Tuple of error codes

Exception

InvalidSessionError: This exception is thrown if the session is invalid or has timed out.

ProtectError: This exception is thrown if the API is unable to protect the data.

> **Note:**
> If the *protect* API is used with bulk data, then it does not throw any exception. Instead, it only returns an error code.

> For more information about the error code, refer to the section *Application Protectors API Return Codes* in the *Protegrity Troubleshooting Guide 9.1.0.0*.

Example

The following table provides examples of the API usage for tokenizing and encrypting the data for each data type.

Data Type	Usage	Refer to
String	Tokenizing string data	*Example: Tokenizing String Data*
	Tokenizing string data with external IV	*Example: Tokenizing String Data with External IV*
	Encrypting string data	*Example: Encrypting String Data*
	Protecting string data using FPE	*Example: Protecting String data using FPE*
	Protecting string data using FPE with external IV and external tweak	*Example: Protecting String Data using FPE with External IV and External Tweak*
	Tokenizing bulk string data	*Example: Tokenizing Bulk String Data*
	Tokenizing bulk string data with external IV	*Example: Tokenizing Bulk String Data with External IV*
	Encrypting bulk string data	*Example: Encrypting Bulk String Data*
	Protecting bulk string data using FPE	*Example: Protecting Bulk String Data Using FPE*
	Protecting bulk string data using FPE with external IV and external tweak	*Example: Protecting Bulk String Using FPE with External IV and External Tweak*
	Tokenizing unicode data	*Example: Tokenizing Unicode Data*
	Encrypting unicode data	*Example: Encrypting Unicode Data*
	Tokenizing bulk unicode data	*Example: Tokenizing Bulk Unicode Data*
	Encrypting bulk unicode data	*Example: Encrypting Bulk Unicode Data*
Integer	Tokenizing integer data	*Example: Tokenizing Integer Data*
	Tokenizing integer data with external IV	*Example: Tokenizing Integer Data with External IV*
	Encrypting integer data	*Example: Encrypting Integer Data*
	Tokenizing bulk integer data	*Example: Tokenizing Bulk Integer Data*
	Tokenizing bulk integer data with external IV	*Example: Tokenizing Bulk Integer Data with External IV*
	Encrypting bulk integer data	*Example: Encrypting Bulk Integer Data*
	Tokenizing long data	*Example: Tokenizing Long Data*
	Tokenizing long data with external IV	*Example: Tokenizing Long Data with External IV*
	Encrypting long data	*Example: Encrypting Long Data*
	Tokenizing bulk long data	*Example: Tokenizing Bulk Long Data*
	Tokenizing bulk long data with external IV	*Example: Tokenizing Bulk Long Data with External IV*
	Encrypting bulk long data	*Example: Encrypting Bulk Long Data*
Float	Tokenizing float data	*Example: Tokenizing Float Data*
	Encrypting float data	*Example: Encrypting Float Data*
	Tokenizing bulk float data	*Example: Tokenizing Bulk Float Data*
	Encrypting bulk float data	*Example: Encrypting Bulk Float Data*
Bytes	Tokenizing bytes data	*Example: Tokenizing Bytes Data*

Data Type	Usage	Refer to
	Tokenizing bytes data with external IV	*Example: Tokenizing Bytes Data with External IV*
	Encrypting bytes data	*Example: Encrypting Bytes Data*
	Tokenizing bulk bytes data	*Example: Tokenizing Bulk Bytes Data*
	Tokenizing bulk bytes data with external IV	*Example: Tokenizing Bulk Bytes Data with External IV*
	Encrypting bulk bytes data	*Example: Encrypting Bulk Bytes Data*
Date	Tokenizing date objects	*Example: Tokenizing Date Objects*
	Tokenizing bulk date objects	*Example: Tokenizing Bulk Date Objects*

2.4.3.10.1 Example - Tokenizing String Data

This section describes how to use the *protect* API for tokenizing the string data.

Example 1: Input string data

In the following example, the *Protegrity1* string is used as the data, which is tokenized using the *TE_A_N_S23_L2R2_Y* Alpha-Numeric data element.

```
from appython import Protector
protector = Protector()
session = protector.create_session("User1")
output = session.protect("Protegrity1", "TE_A_N_S23_L2R2_Y")
print("Protected Data: %s" %output)
```

Result

```
Protected Data: Pr9zdglWRy1
```

Example 2: Input string data using session as Context Manager

In the following example, the *Protegrity1* string is used as the data, which is tokenized using the *TE_A_N_S23_L2R2_Y* Alpha Numeric data element.

```
from appython import Protector
protector = Protector()
with protector.create_session("User1") as session:
    output = session.protect("Protegrity1", "TE_A_N_S23_L2R2_Y")
    print("Protected Data: %s" %output)
```

Result

```
Protected Data: Pr9zdglWRy1
```

Example 3: Input date passed as a string

In the following example, the *29/05/1998* string is used as the data, which is tokenized using the *TE_Date_DMY_S13* Date data element.

Caution: If you have provided the date string as an input in a specific format, then you must use the data element with the same tokenization type as that of the input date format to protect the data. For example, if you have provided the input date string in DD/MM/YYYY format, then you must use only the Date (DD/MM/YYYY) data element to protect the data.

Note: For information regarding the date formats supported by Protegrity tokenization methods, refer to the section *Protegrity Tokenization* in the *Protection Methods Reference Guide 9.1.0.0*.

```
from appython import Protector
protector = Protector()
session = protector.create_session("User1")
output = session.protect("29/05/1998", "TE_Date_DMY_S13")
print("Protected data: "+str(output))
```

Result

```
Protected data: 08/07/2443
```

Example 4: Input date and time passed as a string

In the following example, the *1998/05/29 10:54:47* string is used as the data, which is tokenized using the *TE_Datetime_TN_DN_M* Datetime data element.

Caution: If you have provided the date and time string as an input in a specific format, then you must use the data element with the same tokenization type as that of the input date and time format to protect the data. For example, if you have provided the input date and time string in YYYY/MM/DD HH:MM:SS MMM format, then you must use only the Datetime (YYYY-MM-DD HH:MM:SS MMM) data element to protect the data.

Note: For information regarding the date formats supported by Protegrity tokenization methods, refer to the section *Protegrity Tokenization* in the *Protection Methods Reference Guide 9.1.0.0*.

```
from appython import Protector
protector = Protector()
session = protector.create_session("User1")
output = session.protect("1998/05/29 10:54:47", "TE_Datetime_TN_DN_M")
print("Protected data: "+str(output))
```

Result

```
Protected data: 3311/02/22 10:54:47
```

2.4.3.10.2 Example - Tokenizing String Data with External Initialization Vector (IV)

This section describes how to use the *protect* API for tokenizing string data using external initialization vector (IV).

Note: If you want to pass the external IV as a keyword argument to the *protect* API, then you must first pass the external IV as bytes to the API.

Example

In the following example, the *Protegrity1* string is used as the data, which is tokenized using the *TE_A_N_S23_L2R2_Y* data element, with the help of the external IV *1234* that is passed as bytes.

```
from appython import Protector
protector = Protector()
session = protector.create_session("User1")
output = session.protect("Protegrity1", "TE_A_N_S23_L2R2_Y",
         external_iv=bytes("1234", encoding="utf-8"))
print("Protected Data: %s" %output)
```

Result

```
Protected Data: PrksvEshuy1
```

2.4.3.10.3 Example - Encrypting String Data

This section describes how to use the *protect* API for encrypting the string data.

> **Note:**
> If you want to encrypt the data, then you must use bytes in the *encrypt_to* keyword.

Example

In the following example, the *Protegrity1* string is used as the data, which is encrypted using the *AES256_IV_CRC_KID* data element. Therefore, the *encrypt_to* parameter is passed as a keyword argument and its value is set to *bytes*.

```
from appython import Protector
protector = Protector()
session = protector.create_session("User1")
output = session.protect("Protegrity1", "AES256_IV_CRC_KID",
         encrypt_to=bytes)
print("Encrypted Data: %s" %output)
```

Result

```
Encrypted Data: b'>gmA,i=w^
```

> **Warning:**
> To avoid data corruption, do not convert the encrypted bytes data into the string format. It is recommended to convert the encrypted bytes data to a Hexadecimal, Base 64, or any other appropriate format.

2.4.3.10.4 Example - Protecting String Data Using Format Preserving Encryption (FPE)

This section describes how to use the *protect* API to protect the string data using Format Preserving Encryption (FPE) (FF1).

Example

In the following example, the *protegrity1234ÀÁÂÃÄÅÆÇÈÉ* string is used as the data, which is protected using the FPE data element *FPE_FF1_AES256_ID_AN_LnRn_ASTNE*.

```
from appython import Protector
protector = Protector()
session = protector.create_session("User1")
output = session.protect("protegrity1234ÀÁÂÃÄÅÆÇÈÉ",
         "FPE_FF1_AES256_ID_AN_LnRn_ASTNE")
print("Protected Data: %s" %output)
```

Result

```
Protected Data: NRejBkN7LcBOT4ÀÁÂÃÄÅÆÇÈÉ
```

2.4.3.10.5 Example - Protecting String Data Using FPE with External IV and External Tweak

This section describes how to use the *protect* API for protecting string data using FPE (FF1), with external IV and external tweak.

> **Note:** If you want to pass the external IV and external tweak as keyword arguments to the *protect* API, then you must pass the external IV and external tweak as bytes.

Example

In the following example, the *protegrity1234* string is used as the data, which is protected using the FPE data element *FPE_FF1_AES256_ASCII_APIP_AN_L2R1_ASTNI_ML2*, with the help of external IV *1234* and external tweak *abcdef* that are passed as bytes.

```
from appython import Protector
protector = Protector()
session = protector.create_session("User1")
output = session.protect("protegrity1234",
         "FPE_FF1_AES256_ASCII_APIP_AN_L2R1_ASTNI_ML2", external_iv=bytes("1234",
encoding="utf-8"),
         external_tweak=bytes("abcdef", encoding="utf-8"))
print("Protected Data: %s" %output)
```

Result

```
Protected Data: prS6DaU5Dtd5g4
```

2.4.3.10.6 Example - Tokenizing Bulk String Data

This section describes how to use the *protect* API for tokenizing bulk string data. You can pass bulk string data as a list or a tuple.

> **Caution:** The individual elements of the list or tuple must be of the same data type.

Example 1: Input bulk string data

In the following example, *protegrity1234*, *Protegrity1*, and *Protegrity56* strings are stored in a list and used as bulk data, which is tokenized using the *TE_A_N_S23_L2R2_Y* data element.

```
from appython import Protector
protector = Protector()
session = protector.create_session("User1")
data = ["protegrity1234", "Protegrity1", "Protegrity56"]
p_out = session.protect(data, "TE_A_N_S23_L2R2_Y")
print("Protected Data: ")
print(p_out)
```

Result

```
Protected Data:
(['prMLJsM8fZUp34', 'Pr9zdglWRy1', 'Pra9Ez5LPG56'], (6, 6, 6))
```

6 is the success return code for the protect operation of each element in the list.

Example 2: Input bulk string data

In *Example 1*, the protected output was a tuple of the tokenized data and the error list. The following example shows how you can tweak the code to ensure that you retrieve the protected output and the error list separately, and not as part of a tuple.

```
from appython import Protector
protector = Protector()
session = protector.create_session("User1")
data = ["protegrity1234", "Protegrity1", "Protegrity56"]
p_out, error_list = session.protect(data, "TE_A_N_S23_L2R2_Y")
print("Protected Data: ")
print(p_out)
print("Error List: ")
print(error_list)
```

Result

```
Protected Data:
['prMLJsM8fZUp34', 'Pr9zdglWRy1', 'Pra9Ez5LPG56']
Error List:
(6, 6, 6)
```

6 is the success return code for the protect operation of each element in the list.

Example 3: Input dates passed as bulk strings

In the following example, the *14/02/2019* and *11/03/2018* strings are stored in a list and used as bulk data, which is tokenized using the *TE_Date_DMY_S13* Date data element.

Caution: If you have provided the date string as an input in a specific format, then you must use the data element with the same tokenization type as that of the input date format to protect the data. For example, if you have provided the input date string in DD/MM/YYYY format, then you must use only the Date (DD/MM/YYYY) data element to protect the data.

Note: For information about the date formats supported by Protegrity tokenization methods, refer to the section *Protegrity Tokenization* in the *Protection Methods Reference Guide 9.1.0.0*.

```
from appython import Protector
protector = Protector()
session = protector.create_session("User1")
data = ["14/02/2019", "11/03/2018"]
output = session.protect(data, "TE_Date_DMY_S13")
print("Protected data: "+str(output))
```

Result

```
Protected data: (['08/07/2443', '17/08/1830'], (6, 6))
```

6 is the success return code for the protect operation of each element in the list.

Example 4: Input date and time passed as bulk strings

In the following example, the *2019/02/14 10:54:47* and *2019/11/03 11:01:32* strings is used as the data, which is tokenized using the *TE_Datetime_TN_DN_M* Datetime data element.

Caution: If you have provided the date and time string as an input in a specific format, then you must use the data element with the same tokenization type as that of the input date and time format to protect the data. For example, if you have provided the

input date and time string in YYYY/MM/DD HH:MM:SS MMM format, then you must use only the Datetime (YYYY-MM-DD HH:MM:SS MMM) data element to protect the data.

Note: For information about the date formats supported by Protegrity tokenization methods, refer to the section *Protegrity Tokenization* in the *Protection Methods Reference Guide 9.1.0.0*.

```
from appython import Protector
protector = Protector()
session = protector.create_session("User1")
data = ["2019/02/14 10:54:47", "2019/11/03 11:01:32"]
output = session.protect(data, "TE_Datetime_TN_DN_M")
print("Protected data: "+str(output))
```

Result

```
Protected data: (['3311/02/22 10:54:47', '3311/11/02 11:01:32'], (6, 6))
```

6 is the success return code for the protect operation of each element in the list.

2.4.3.10.7 Example - Tokenizing Bulk String Data with External IV

This section describes how to use the *protect* API for tokenizing bulk string data using external IV. You can pass bulk string data as a list or a tuple.

Caution: The individual elements of the list or tuple must be of the same data type.

Note: If you want to pass the external IV as a keyword argument to the *protect* API, then you must pass external IV as bytes.

Example

In the following example, *protegrity1234*, *Protegrity1*, and *Protegrity56* strings are stored in a list and used as bulk data, which is tokenized using the *TE_A_N_S23_L2R2_Y* data element, with the help of external IV *123* that is passed as bytes.

```
from appython import Protector
protector = Protector()
session = protector.create_session("User1")
data = ["protegrity1234", "Protegrity1", "Protegrity56"]
p_out = session.protect(data, "TE_A_N_S23_L2R2_Y",
        external_iv=bytes("123", encoding="utf-8"))
print("Protected Data: ")
print(p_out)
```

Result

```
Protected Data:
(['prv0WozsSjbS34', 'PrtigABOCy1', 'PrvjDdC2TD56'], (6, 6, 6))
```

6 is the success return code for the protect operation of each element in the list.

2.4.3.10.8 Example - Encrypting Bulk String Data

This section describes how to use the *protect* API for encrypting bulk string data. You can pass bulk string data as a list or a tuple.

Caution: The individual elements of the list or tuple must be of the same data type.

Note:
If you want to encrypt the data, then you must use bytes in the *encrypt_to* keyword.

Example

In the following example, *protegrity1234*, *Protegrity1*, and *Protegrity56* strings are stored in a list and used as bulk data, which is encrypted using the *AES256* data element. Therefore, the *encrypt_to* parameter is passed as a keyword argument, and its value is set to *bytes*.

```
from appython import Protector
protector = Protector()
session = protector.create_session("User1")
data = ["protegrity1234", "Protegrity1", "Protegrity56"]
p_out = session.protect(data, "AES256", encrypt_to=bytes)
print("Encrypted Data: ")
print(p_out)
```

Result

```
Encrypted Data:
([b'\xc9^x\x02)\xcbB\x91}\x7fi\x8a\xce\x8d>H',
b't\x80\xf5\x8d\x9e\x0b+4Lq\x8a\x97\xdb\x8fx\x16',
b'\x87\x08\x938\xf7o~\xab\xa3\xc2L\xa90>\x18_'], (6, 6, 6))
```

6 is the success return code for the protect operation of each element in the list.

Warning:
To avoid data corruption, do not convert the encrypted bytes data into string format. It is recommended to convert the encrypted bytes data to a Hexadecimal, Base 64, or any other appropriate format.

2.4.3.10.9 Example - Protecting Bulk String Data Using FPE

This section describes how to use the *protect* API for protecting bulk string data using FPE (FF1). You can pass bulk string data as a list or a tuple.

Caution: The individual elements of the list or tuple must be of the same data type.

Example

In the following example, *protegrity1234ÀÁ*, *Protegrity1ÆÇÈ*, and *Protegrity56ÀÁÂÃÄÅ* strings are stored in a list and used as bulk data, which is protected using the FPE data element *FPE_FF1_AES256_APIP_AN_LnRn_ASTNE*.

```
from appython import Protector
protector = Protector()
session = protector.create_session("User1")
data = ["protegrity1234ÀÁ", "Protegrity1ÆÇÈ", "Protegrity56ÀÁÂÃÄÅ"]
p_out = session.protect(data, "FPE_FF1_AES256_APIP_AN_LnRn_ASTNE")
```

```
print("Protected Data: ")
print(p_out)
```

Result

```
Protected Data:
([u'MG01UHDQ8VyON3\xc0\xc1', u'8APfLh3W9TY\xc6\xc7\xc8',
u'4XYdSFURF4bV\xc0\xc1\xc2\xc3\xc4\xc5'], (6, 6, 6))
```

6 is the success return code for the protect operation of each element in the list.

2.4.3.10.10 Example - Protecting Bulk String Data Using FPE with External IV and External Tweak

This section describes how to use the *protect* API for protecting the bulk string data using FPE (FF1), with external IV and external tweak. You can pass bulk string data as a list or a tuple.

Caution: The individual elements of the list or tuple must be of the same data type.

Note: If you want to pass the external IV and external tweak as keyword arguments to the *protect* API, then you must pass the external IV and external tweak as bytes.

Example

In the following example, *protegrity1234ÀÁ*, *Protegrity1ÆÇÈ*, and *Protegrity56ÀÁÂÃÄÅ* strings are stored in a list and used as bulk data. This bulk data is protected using the FPE data element *FPE_FF1_AES256_APIP_AN_LnRn_ASTNE*, with the help of external IV *1234* and external tweak *xyz* that are passed as bytes.

```
from appython import Protector
protector = Protector()
session = protector.create_session("User1")
data = ["protegrity1234ÀÁ", "Protegrity1ÆÇÈ", "Protegrity56ÀÁÂÃÄÅ"]
p_out = session.protect(data, "FPE_FF1_AES256_APIP_AN_LnRn_ASTNE",
        external_iv=bytes("1234", encoding="utf-8"), external_tweak=bytes("xyz",
encoding="utf-8"))
print("Protected Data: ")
print(p_out)
```

Result

```
Protected Data:
([u'WwR5aK2BMoUlcz\xc0\xc1', u'nW6lqjd7NGR\xc6\xc7\xc8',
u'o6eBUZDNuyWU\xc0\xc1\xc2\xc3\xc4\xc5'], (6, 6, 6))
```

6 is the success return code for the protect operation of each element in the list.

2.4.3.10.11 Example - Tokenizing Integer Data

This section describes how to use the *protect* API for tokenizing integer data.

Example

In the following example, *21* is used as the integer data, which is tokenized using the *TE_INT_4* data element.

```
from appython import Protector
protector = Protector()
session = protector.create_session("User1")
output = session.protect(21, "TE_INT_4")
print("Protected Data: %s" %output)
```

Result

```
Protected Data: -1926573911
```

2.4.3.10.12 Example - Tokenizing Integer Data with External IV

This section describes how to use the *protect* API for tokenizing integer data using the external IV.

> **Note:** If you want to pass the external IV as a keyword argument to the *protect* API, then you must pass the external IV as bytes to the API.

Example

In the following example, *21* is used as the integer data, which is tokenized using the *TE_INT_4* data element, with the help of external IV *1234* that is passed as bytes.

```
from appython import Protector
protector = Protector()
session = protector.create_session("User1")
output = session.protect(21, "TE_INT_4", external_iv=bytes("1234", encoding="utf-8"))
print("Protected Data: %s" %output)
```

Result

```
Protected Data: -2122057622
```

2.4.3.10.13 Example - Encrypting Integer Data

This section describes how to use the *protect* API for encrypting integer data.

> **Note:**
> If you want to encrypt the data, then you must use bytes in the *encrypt_to* keyword.

Example

In the following example, *21* is used as the integer data, which is encrypted using the *AES256* data element. Therefore, the *encrypt_to* parameter is passed as a keyword argument, and its value is set to *bytes*.

```
from appython import Protector
protector = Protector()
session = protector.create_session("User1")
output = session.protect(21, "AES256", encrypt_to=bytes)
print("Encrypted Data: %s" %output)
```

Result

```
Encrypted Data: b'@upkN'
```

> **Warning:**
> To avoid data corruption, do not convert the encrypted bytes data into string format. It is recommended to convert the encrypted bytes data to a Hexadecimal, Base 64, or any other appropriate format.

2.4.3.10.14 Example - Tokenizing Bulk Integer Data

This section describes how to use the *protect* API for tokenizing bulk integer data. You can pass bulk integer data as a list or a tuple.

Caution: The individual elements of the list or tuple must be of the same data type.

Example

In the following example, *21*, *42*, and *55* integers are stored in a list and used as bulk data, which is tokenized using the *TE_INT_4* data element.

```
from appython import Protector
protector = Protector()
session = protector.create_session("User1")
data = [21, 42, 55]
p_out = session.protect(data, "TE_INT_4")
print("Protected Data: ")
print(p_out)
```

Result

```
Protected Data:
([-1926573911, -1970496120, -814489753], (6, 6, 6))
```

6 is the success return code for the protect operation of each element in the list.

2.4.3.10.15 Example - Tokenizing Bulk Integer Data with External IV

This section describes how to use the *protect* API for tokenizing bulk integer data using external IV. You can pass bulk integer data as a list or a tuple.

Caution: The individual elements of the list or tuple must be of the same data type.

Note: If you want to pass the external IV as a keyword argument to the *protect* API, then you must pass the external IV as bytes to the API.

Example

In the following example, *21*, *42*, and *55* integers are stored in a list and used as bulk data, which is tokenized using the *TE_INT_4* data element, with the help of external IV *1234* that is passed as bytes.

```
from appython import Protector
protector = Protector()
session = protector.create_session("User1")
data = [21, 42, 55]
p_out = session.protect(data, "TE_INT_4", external_iv=bytes("1234", encoding="utf-8"))
print("Protected Data: ")
print(p_out)
```

Result

```
Protected Data:
([-2122057622, 1795905968, 228587043], (6, 6, 6))
```

6 is the success return code for the protect operation of each element in the list.

2.4.3.10.16 Example - Encrypting Bulk Integer Data

This section describes how to use the *protect* API for encrypting bulk integer data. You can pass bulk integer data as a list or a tuple.

> **Note:**
> If you want to encrypt the data, then you must use bytes in the *encrypt_to* keyword.

Example

In the following example, *21*, *42*, and *55* integers are stored in a list and used as bulk data, which is encrypted using the *AES256* data element. Therefore, the *encrypt_to* parameter is passed as a keyword argument and its value is set to *bytes*.

```
from appython import Protector
protector = Protector()
session = protector.create_session("User1")
data = [21, 42, 55]
p_out = session.protect(data, "AES256", encrypt_to=bytes)
print("Encrypted Data: ")
print(p_out)
```

Result

```
Encrypted Data:
([b'@\x19\xccu\x04\xc7\xd8\xc1p\xad\xa7\x1fk\xe4N\xd0',
b'"@\xec\x97(\x96\xab\x18\xd0\x99\xd4~\x1e\xf4\xba\xd1',
b'y\xec\x9b+f\xa8\xb1I\xc2=[\x11\xfd\x06\xa1C'], (6, 6, 6))
```

6 is the success return code for the protect operation of each element in the list.

> **Warning:**
> To avoid data corruption, do not convert the encrypted bytes data into string format. It is recommended to convert the encrypted bytes data to a Hexadecimal, Base 64, or any other appropriate format.

2.4.3.10.17 Example - Tokenizing Long Data

This section describes how to use the *protect* API for tokenizing long data.

Example

In the following example, *1376235139103947* is used as the long data, which is tokenized using the *TE_INT_8* data element.

```
from appython import Protector
protector = Protector()
session = protector.create_session("User1")
output = session.protect(1376235139103947, "TE_INT_8")
print("Protected Data: %s" %output)
```

Result

```
Protected Data: -1770169866845757900
```

2.4.3.10.18 Example - Tokenizing Long Data with External IV

This section describes how to use the *protect* API for tokenizing long data using external IV.

> **Note:** If you want to pass the external IV as a keyword argument to the *protect* API, then you must use bytes in the *encrypt_to* keyword.

Example

In the following example, *1376235139103947* is used as the long data, which is tokenized using the *TE_INT_8* data element, with the help of external IV *1234* that is passed as bytes.

```
from appython import Protector
protector = Protector()
session = protector.create_session("User1")
output = session.protect(1376235139103947, "TE_INT_8",
         external_iv=bytes("1234", encoding="utf-8"))
print("Protected Data: %s" %output)
```

Result

```
Protected Data: 5846214101577367207
```

2.4.3.10.19 Example - Encrypting Long Data

This section describes how to use the *protect* API for encrypting long data.

> **Note:**
> If you want to encrypt the data, then you must use bytes in the *encrypt_to* keyword.

Example

In the following example, *1376235139103947* is used as the long data, which is encrypted using the *AES256* data element. Therefore, the *encrypt_to* parameter is passed as a keyword argument and its value is set to *bytes*.

```
from appython import Protector
protector = Protector()
session = protector.create_session("User1")
output = session.protect(1376235139103947, "AES256", encrypt_to=bytes)
print("Encrypted Data: %s" %output)
```

Result

```
Encrypted Data: b'Dsp0Xl<\'
```

> **Warning:**
> To avoid data corruption, do not convert the encrypted bytes data into string format. It is recommended to convert the encrypted bytes data to a Hexadecimal, Base 64, or any other appropriate format.

2.4.3.10.20 Example - Tokenizing Bulk Long Data

This section describes how to use the *protect* API for tokenizing bulk long data. You can pass bulk long data as a list or a tuple.

> **Caution:** The individual elements of the list or tuple must be of the same data type.

Example

In the following example, *1376235139103947*, *2396235839173981*, and *9371234126176985* long data are stored in a list and used as bulk data, which is tokenized using the *TE_INT_8* data element.

```
from appython import Protector
protector = Protector()
session = protector.create_session("User1")
data = [1376235139103947, 2396235839173981, 9371234126176985]
p_out = session.protect(data, "TE_INT_8")
print("Protected Data: ")
print(p_out)
```

Result

```
Protected Data:
([-1770169866845757900L, -8142006510957348982L, -206876567049699669L], (6, 6, 6))
```

6 is the success return code for the protect operation of each element in the list.

2.4.3.10.21 Example - Tokenizing Bulk Long Data with External IV

This section describes how to use the *protect* API for tokenizing bulk long data using external IV. You can pass bulk long data as a list or a tuple.

Caution: The individual elements of the list or tuple must be of the same data type.

Note: If you want to pass the external IV as a keyword argument to the *protect* API, then you must pass the external IV and external tweak as bytes.

Example

In the following example, *1376235139103947*, *2396235839173981*, and *9371234126176985* long data are stored in a list and used as bulk data, which is tokenized using the *TE_INT_8* data element, with the help of external IV *1234* that is passed as bytes.

```
from appython import Protector
protector = Protector()
session = protector.create_session("User1")
data = [1376235139103947, 2396235839173981, 9371234126176985]
p_out = session.protect(data, "TE_INT_8", external_iv=bytes("1234", encoding="utf-8"))
print("Protected Data: ")
print(p_out)
```

Result

```
Protected Data:
([5846214101577367207L, 5661139619224336475L, 7806173497368534531L], (6, 6, 6))
```

6 is the success return code for the protect operation of each element in the list.

2.4.3.10.22 Example - Encrypting Bulk Long Data

This section describes how to use the *protect* API for encrypting bulk long data. You can pass bulk long data as a list or a tuple.

Caution: The individual elements of the list or tuple must be of the same data type.

Note:

> If you want to encrypt the data, then you must use bytes in the *encrypt_to* keyword.

Example

In the following example, *1376235139103947*, *2396235839173981*, and *9371234126176985* long data are stored in a list and used as bulk data, which is encrypted using the *AES256* data element. Therefore, the *encrypt_to* parameter is passed as a keyword argument and its value is set to *bytes*.

```
from appython import Protector
protector = Protector()
session = protector.create_session("User1")
data = [1376235139103947, 2396235839173981, 9371234126176985]
p_out = session.protect(data, "AES256", encrypt_to=bytes)
print("Encrypted Data: ")
print(p_out)
```

Result

```
Encrypted Data:
([b'\xd5Ds\xb3\xfb\x95\xf2wp0Xl<\\\x1a\x07', b'\xaf\x05aq\xb6\xcd,L`JC4\x87\x87\t\x0b',
b']j@*S\x96\xf5\xf5S<\x08M\xa6\x18\xbf\xda'], (6, 6, 6))
```

6 is the success return code for the protect operation of each element in the list.

> **Warning:**
> To avoid data corruption, do not convert the encrypted bytes data into string format. It is recommended to convert the encrypted bytes data to a Hexadecimal, Base 64, or any other appropriate format.

2.4.3.10.23 Example - Protecting Float Data

This section describes how to use the *protect* API for protecting float data using a No Encryption data element. You can use this API for access control and auditing.

Example

In the following example, *22.5* is used as the float data, which is protected using the *NoEncryption_1* data element.

```
from appython import Protector
protector = Protector()
session = protector.create_session("User1")
output = session.protect(22.5, "NoEncryption_1")
print("Protected Data: %s" %output)
```

Result

```
Protected Data: 22.5
```

As we are using a No Encryption data element to protect the data, the protected output data is the same as the input data.

2.4.3.10.24 Example - Encrypting Float Data

This section describes how to use the *protect* API for encrypting float data.

> **Note:**
> If you want to encrypt the data, then you must use bytes in the *encrypt_to* keyword.

Example

In the following example, *22.5* is used as the float data, which is encrypted using the *AES256* data element. Therefore, the *encrypt_to* parameter is passed as a keyword argument and its value is set to *bytes*.

```
from appython import Protector
protector = Protector()
session = protector.create_session("User1")
output = session.protect(22.5, "AES256", encrypt_to=bytes)
print("Encrypted Data: %s" %output)
```

Result

```
Encrypted Data: b'g.O▒Vk▒>'
```

Warning:
To avoid data corruption, do not convert the encrypted bytes data into string format. It is recommended to convert the encrypted bytes data to a Hexadecimal, Base 64, or any other appropriate format.

2.4.3.10.25 Example - Protecting Bulk Float Data

This section describes how to use the *protect* API for protecting bulk float data using a No Encryption data element. You can pass bulk float data as a list or a tuple. You can use this API for access control and auditing.

Caution: The individual elements of the list or tuple must be of the same data type.

Example

In the following example, *22.5*, *48.93*, and *94.14* float data are stored in a list and used as bulk data, which is protected using the *NoEncryption_1* data element.

```
from appython import Protector
protector = Protector()
session = protector.create_session("User1")
data = [22.5, 48.93, 94.31]
p_out = session.protect(data, "NoEncryption_1")
print("Protected Data: ")
print(p_out)
```

Result

```
Protected Data:
([22.5, 48.93, 94.31], (6, 6, 6))
```

6 is the success return code for the protect operation of each element in the list.

As we are using a No Encryption data element to protect the data, the protected output data is the same as the input data.

2.4.3.10.26 Example - Encrypting Bulk Float Data

This section describes how to use the *protect* API for encrypting bulk float data. You can pass bulk float data as a list or a tuple.

Caution: The individual elements of the list or tuple must be of the same data type.

Note:

If you want to encrypt the data, then you must use bytes in the *encrypt_to* keyword.

Example

In the following example, *22.5*, *48.93*, and *94.14* float data are stored in a list and used as bulk data, which is encrypted using the *AES256* data element. Therefore, the *encrypt_to* parameter is passed as a keyword argument and its value is set to *bytes*.

```
from appython import Protector
protector = Protector()
session = protector.create_session("User1")
data = [22.5, 48.93, 94.31]
p_out = session.protect(data, "AES256", encrypt_to=bytes)
print("Encrypted Data: ")
print(p_out)
```

Result

```
Encrypted Data:
([b'g.O\xd8\x8b\x12\x89\x15Vk\x88\xbe\xf4;\x18>',
b'.\xb0Q\xb9\xc9\xca\xba\xc2\xcb8\xfe\xd8\xf4q\x00\xb8',
b'\xb6x\xf4\x94l9\xe6uaN\x83\x8d\n\x98\n;'], (6, 6, 6))
```

6 is the success return code for the protect operation of each element in the list.

Warning:

To avoid data corruption, do not convert the encrypted bytes data into string format. It is recommended to convert the encrypted bytes data to a Hexadecimal, Base 64, or any other appropriate format.

2.4.3.10.27 Example - Tokenizing Bytes Data

This section describes how to use the *protect* API for tokenizing bytes data.

Example

In the following example, *"Protegrity1"* string is first converted to bytes using the Python *bytes()* method. The bytes data is then tokenized using the *TE_A_N_S23_L2R2_Y* data element.

```
from appython import Protector
protector = Protector()
session = protector.create_session("User1")
data=bytes("Protegrity1", encoding="utf-8")
p_out = session.protect(data, "TE_A_N_S23_L2R2_Y")
print("Protected Data: %s" %p_out)
```

Result

```
Protected Data: b'Pr9zdglWRy1'
```

2.4.3.10.28 Example - Tokenizing Bytes Data with External IV

This section describes how to use the *protect* API for tokenizing bytes data using external IV.

Example

In the following example, *"Protegrity1"* string is first converted to bytes using the Python *bytes()* method. The bytes data is then tokenized using the *TE_A_N_S23_L2R2_Y* data element, with the help of external IV *1234* that is passed as bytes.

```
from appython import Protector
protector = Protector()
session = protector.create_session("User1")
data=bytes("Protegrity1", encoding="utf-8")
output = session.protect(data, "TE_A_N_S23_L2R2_Y",
         external_iv=bytes("1234", encoding="utf-8"))
print("Protected Data: %s" %output)
```

Result

```
Protected Data: b'PrksvEshuy1'
```

2.4.3.10.29 Example - Encrypting Bytes Data

This section describes how to use the *protect* API for encrypting bytes data.

Example

In the following example, *"Protegrity1"* string is first converted to bytes using the Python *bytes()* method. The bytes data is then encrypted using the *AES256* data element. Therefore, the *encrypt_to* parameter is passed as a keyword argument and its value is set to *bytes*.

```
from appython import Protector
protector = Protector()
session = protector.create_session("User1")
data=bytes("Protegrity1", encoding="utf-8")
p_out = session.protect(data, "AES256", encrypt_to = bytes)
print("Encrypted Data: %s" %p_out)
```

Result

```
Encrypted Data: b't▒4Lq▒'
```

Warning:
To avoid data corruption, do not convert the encrypted bytes data into string format. It is recommended to convert the encrypted bytes data to a Hexadecimal, Base 64, or any other appropriate format.

2.4.3.10.30 Example - Tokenizing Bulk Bytes Data

This section describes how to use the *protect* API for tokenizing bulk bytes data. You can pass bulk bytes data as a list or a tuple.

Caution: The individual elements of the list or tuple must be of the same data type.

Example

In the following example, *protegrity1234*, *Protegrity1*, and *Protegrity56* strings are first converted to bytes using the Python *bytes()* method. The converted bytes are then stored in a list and used as bulk data, which is tokenized using the *TE_A_N_S23_L2R2_Y* data element.

```
from appython import Protector
protector = Protector()
session = protector.create_session("User1")
data = [bytes("protegrity1234", encoding="UTF-8"), bytes("Protegrity1",
encoding="UTF-8"), bytes("Protegrity56", encoding="UTF-8")]
```

```
p_out = session.protect(data, "TE_A_N_S23_L2R2_Y")
print("Protected Data: ")
print(p_out)
```

Result

```
Protected Data:
([b'prMLJsM8fZUp34', b'Pr9zdglWRy1', b'Pra9Ez5LPG56'], (6, 6, 6))
```

6 is the success return code for the protect operation of each element in the list.

2.4.3.10.31 Example - Tokenizing Bulk Bytes Data with External IV

This section describes how to use the *protect* API for tokenizing bulk bytes data using external IV. You can pass bulk bytes data as a list or a tuple.

> **Caution:** The individual elements of the list or tuple must be of the same data type.

Example

In the following example, *protegrity1234*, *Protegrity1*, and *Protegrity56* strings are first converted to bytes using the Python *bytes()* method. The converted bytes are then stored in a list and used as bulk data, which is tokenized using the *TE_A_N_S23_L2R2_Y* data element, with the help of external IV *1234* that is passed as bytes.

```
from appython import Protector
protector = Protector()
session = protector.create_session("User1")
data = [bytes("protegrity1234", encoding="UTF-8"), bytes("Protegrity1",
encoding="UTF-8"), bytes("Protegrity56", encoding="UTF-8")]
p_out = session.protect(data, "TE_A_N_S23_L2R2_Y",
       external_iv=bytes("1234", encoding="utf-8"))
print("Protected Data: ")
print(p_out)
```

Result

```
Protected Data:
([b'prbm147L5pc434', b'PrksvEshuy1', b'Prmx0hG8Nj56'], (6, 6, 6))
```

6 is the success return code for the protect operation of each element in the list.

2.4.3.10.32 Example - Encrypting Bulk Bytes Data

This section describes how to use the *protect* API for encrypting bulk bytes data. You can pass bulk bytes data as a list or a tuple.

> **Caution:** The individual elements of the list or tuple must be of the same data type.

Example

In the following example, *protegrity1234*, *Protegrity1*, and *Protegrity56* strings are first converted to bytes using the Python *bytes()* method. The converted bytes are then stored in a list and used as bulk data, which is encrypted using the *AES256* data element. Therefore, the *encrypt_to* parameter is passed as a keyword argument and its value is set to *bytes*.

```
from appython import Protector
protector = Protector()
session = protector.create_session("User1")
data = [bytes("protegrity1234", encoding="UTF-8"), bytes("Protegrity1",
encoding="UTF-8"), bytes("Protegrity56", encoding="UTF-8")]
p_out = session.protect(data, "AES256", encrypt_to = bytes)
```

```
print("Encrypted Data: ")
print(p_out)
```

Result

```
Encrypted Data:
([b'\xc9^x\x02)\xcbB\x91}\x7fi\x8a\xce\x8d>H',
b't\x80\xf5\x8d\x9e\x0b+4Lq\x8a\x97\xdb\x8fx\x16',
b'\x87\x08\x938\xf7o~\xab\xa3\xc2L\xa90>\x18_'], (6, 6, 6))
```

6 is the success return code for the protect operation of each element in the list.

Warning:

To avoid data corruption, do not convert the encrypted bytes data into string format. It is recommended to convert the encrypted bytes data to a Hexadecimal, Base 64, or any other appropriate format.

2.4.3.10.33 Example - Tokenizing Date Objects

This section describes how to use the *protect* API for tokenizing the date objects.

Caution: If you have provided the date object as an input in a specific format, then you must use the data element with the same tokenization type as that of the input date format to protect the data. For example, if you have provided the input date object in DD/MM/YYYY format, then you must use only the Date (DD/MM/YYYY) data element to protect the data.

Note: For information about the date formats supported by Protegrity tokenization methods, refer to the section *Protegrity Tokenization* in the *Protection Methods Reference Guide 9.1.0.0*.

Example 1: Input date object in DD/MM/YYYY format

In the following example, the *29/05/1998* date string is used as the data, which is first converted to a date object using the Python *date* method of the *datetime* module.

The date object is then tokenized using the *TE_Date_DMY_S13* data element.

```
from appython import Protector
from datetime import datetime
protector = Protector()
session = protector.create_session("User1")
data = datetime.strptime("29/05/1998", "%d/%m/%Y").date()
print("Input date as a Date object : "+str(data))
p_out = session.protect(data, "TE_Date_DMY_S13")
print("Protected date: "+str(p_out))
```

Result

```
Input date as a Date object : 1998-05-29
Protected date: 1896-10-21
```

Example 2: Input date object in MM/DD/YYYY format

In the following example, the *05/29/1998* date string is used as the data, which is first converted to a date object using the Python *date* method of the *datetime* module.

The date object is then tokenized using the *TE_Date_MDY_S13* data element.

```
from appython import Protector
from datetime import datetime
```

```
protector = Protector()
session = protector.create_session("User1")
data = datetime.strptime("05/29/1998", "%m/%d/%Y").date()
print("\nInput date as a Date object : "+str(data))
p_out = session.protect(data, "TE_Date_MDY_S13")
print("Protected date: "+str(p_out))
```

Result

```
Input date as a Date object : 1998-05-29
Protected date: 2037-06-12
```

Example 3: Input date object in YYYY/DD/MM format

In the following example, the *1998/05/29* date string is used as the data, which is first converted to a date object using the Python *date* method of the *datetime* module.

The date object is then tokenized using the *TE_Date_ YMD_S13* data element.

```
from appython import Protector
from datetime import datetime
protector = Protector()
session = protector.create_session("User1")
data = datetime.strptime("1998/05/29", "%Y/%m/%d").date()
print("\nInput date as a Date object : "+str(data))
p_out = session.protect(data, "TE_Date_YMD_S13")
print("Protected date: "+str(p_out))
```

Result

```
Input date as a Date object : 1998-05-29
Protected date: 2615-12-23
```

2.4.3.10.34 Example - Tokenizing Bulk Date Objects

This section describes how to use the *protect* API for tokenizing bulk date objects. You can pass bulk date objects as a list or a tuple.

Caution: The individual elements of the list or tuple must be of the same data type.

Caution: If you have provided the date object as an input in a specific format, then you must use the data element with the same tokenization type as that of the input date format to protect the data. For example, if you have provided the input date object in DD/MM/YYYY format, then you must use only the Date (DD/MM/YYYY) data element to protect the data.

Note: For information about the date formats supported by Protegrity tokenization methods, refer to the section *Protegrity Tokenization* in the *Protection Methods Reference Guide 9.1.0.0*.

Example: Input as a Date Object

In the following example, the *12/02/2019* and *11/01/2018* date strings are used as the data, which are first converted to a date objects using the Python *date* method of the *datetime* module. The two date objects are then used to create a list, which is used as the input data.

The input list is then tokenized using the *TE_Date_DMY_S13* data element.

```
from appython import Protector
from datetime import datetime
protector = Protector()
session = protector.create_session("User1")
data1 = datetime.strptime("12/02/2019", "%d/%m/%Y").date()
```

```
data2 = datetime.strptime("11/01/2018", "%d/%m/%Y").date()
data = [data1, data2]
print("Input data: ", str(data))
p_out = session.protect(data, "TE_Date_DMY_S13")
print("Protected data: "+str(p_out))
```

Result

```
Input data:  [datetime.date(2019, 2, 12), datetime.date(2018, 1, 11)]
Protected data: ([datetime.date(1896, 10, 21), datetime.date(696, 3, 1)], (6, 6))
```

6 is the success return code for the protect operation of each element in the list.

2.4.3.10.35 Example - Tokenizing Unicode Data

This section explains how to use the *protect* API for tokenizing unicode data.

Example

In the following example, the *u'protegrity1234ÀÁÂÃÄÅÆÇÈÉ'* unicode data is used as the input data, which is tokenized using the *TE_A_N_S23_L2R2_Y* data element.

```
from appython import Protector
protector = Protector()
session = protector.create_session("User1")
output = session.protect(u'protegrity1234ÀÁÂÃÄÅÆÇÈÉ', "TE_A_N_S23_L2R2_Y")
print("Protected Data: %s" %output)
```

Result

```
Protected Data:prZeslalwuQQy3ÀÁÂÃÄÅÆÇÈÉ
```

2.4.3.10.36 Example - Encrypting Unicode Data

This section describes how to use the *protect* API for encrypting unicode data.

> **Note:**
> If you want to encrypt the data, then you must use bytes in the *encrypt_to* keyword.

Example

In the following example, the *u'protegrity1234ÀÁÂÃÄÅÆÇÈÉ'* unicode data is used as the input data, which is encrypted using the *AES256_IV_CRC_KID* data element. Therefore, the *encrypt_to* parameter is passed as a keyword argument, and its value is set to *bytes*.

```
from appython import Protector
protector = Protector()
session = protector.create_session("User1")
output = session.protect(u'protegrity1234ÀÁÂÃÄÅÆÇÈÉ', "AES256_IV_CRC_KID",
         encrypt_to=bytes)
print("Encrypted Data: %s" %output)
```

Result

```
Encrypted Data: b' +>[4AzVOKc1W-&ng%-
```

> **Warning:**

To avoid data corruption, do not convert the encrypted bytes data into string format. It is recommended to convert the encrypted bytes data to a Hexadecimal, Base 64, or any other appropriate format.

2.4.3.10.37 Example - Tokenizing Bulk Unicode Data

This section describes how to use the *protect* API for tokenizing bulk unicode data. You can pass bulk unicode data as a list or a tuple.

Caution: The individual elements of the list or tuple must be of the same data type.

Example

In the following example, *u'protegrity1234ÀÁÂÃÄÅÆÇÈÉ'*, *u'Protegrity1ÆÇÈÉÀÁÂÃÄÅ'*, and *u'Protegrity56ÇÅÆÈÉÂÃ'* unicode data are stored in a list and used as bulk data, which is tokenized using the *TE_A_N_S23_L2R2_Y* data element.

```
from appython import Protector
protector = Protector()
session = protector.create_session("User1")
data = [u'protegrity1234ÀÁÂÃÄÅÆÇÈÉ', u'Protegrity1ÆÇÈÉÀÁÂÃÄÅ', u'Protegrity56ÇÅÆÈÉÂÃ']
p_out = session.protect(data, "TE_A_N_S23_L2R2_Y")
print("Protected Data: ")
print(p_out)
```

Result

```
Protected Data:
([u'prZeslalwuQQy3\xc0\xc1\xc2\xc3\xc4\xc5\xc6\xc7\xc8\xc9',
u'PrVt6rfyW81\xc6\xc7\xc8\xc9\xc0\xc1\xc2\xc3\xc4\xc5',
u'PrFgczleNkNG\xc7\xc5\xc6\xc8\xc9\xc2\xc3'], (6, 6, 6))
```

6 is the success return code for the protect operation of each element in the list.

2.4.3.10.38 Example - Encrypting Bulk Unicode Data

This section describes how to use the *protect* API for encrypting bulk unicode data. You can pass bulk string data as a list or a tuple.

Caution: The individual elements of the list or tuple must be of the same data type.

Note:
If you want to encrypt the data, then you must use bytes in the *encrypt_to* keyword.

Example

In the following example, *u'protegrity1234ÀÁÂÃÄÅÆÇÈÉ'*, *u'Protegrity1ÆÇÈÉÀÁÂÃÄÅ'*, and *u'Protegrity56ÇÅÆÈÉÂÃ'* unicode data are stored in a list and used as bulk data, which is encrypted using the *AES256* data element. Therefore, the *encrypt_to* parameter is passed as a keyword argument and its value is set to *bytes*.

```
from appython import Protector
protector = Protector()
session = protector.create_session("User1")
data = [u'protegrity1234ÀÁÂÃÄÅÆÇÈÉ', u'Protegrity1ÆÇÈÉÀÁÂÃÄÅ', u'Protegrity56ÇÅÆÈÉÂÃ']
p_out = session.protect(data, "AES256", encrypt_to=bytes)
```

```
print("Encrypted Data: ")
print(p_out)
```

Result

```
Encrypted Data:
([b'F2\xd2\xddR\xda\x9e7#\xfc\xe6\xe2Ore\x18>=\x87\xfc\xea\x9c\xb8\x94\x9e$M?
\x9a\xec\xefO5\xc3\x8fjun\xe3\r4\x0f\xedD76\xe4\xfa',
b'\x9f\xc0}G\x12\x1bu\x02\xfdMO\x8e\x01\xb6\x0f\xf5\xbbi\xbe\xc9\x11J\x1c\xa4\x12\x1e\xf
0\xbeA\x19\xa4\xc3', b'G\xa3(\xee\xb7\x81m\xfc\x96-I\xa2\x9eGt\xcc\x0b-
\x97\xc73\x00O\xdc\xfb\t.\xfa=\x99:\xe7'], (6, 6, 6))
```

6 is the success return code for the protect operation of each element in the list.

Warning:
To avoid data corruption, do not convert the encrypted bytes data into string format. It is recommended to convert the encrypted bytes data to a Hexadecimal, Base 64, or any other appropriate format.

2.4.3.11 unprotect

The *unprotect* API unprotects the protected data and returns it in its original form.

def unprotect(self, data, de, **kwargs)

Parameters

data: Data to be unprotected

de: String containing the data element name defined in policy

*****kwargs***: Specify one or more of the following keyword arguments:

- *external_iv*: Specify the external initialization vector for Tokenization and FPE protection methods. This argument is optional.
- *decrypt_to*: Specify this argument for decrypting the data and set its value to the data type of the original data. For example, if you are unprotecting a string data, then you must specify the output data type as *str*. This argument is Mandatory. This argument must not be used for Tokenization and FPE protection methods. The possible values for the *decrypt_to* argument are:
  - *str*
  - *int*
  - *long*
  - *float*
  - *bytes*
- *external_tweak*: Specify the external tweak value for FPE protection method. This argument is optional.

Note: Keyword arguments are case sensitive.

Returns

- *For single data*: Returns the unprotected data
- *For bulk data*: Returns a tuple of the following data:
  - List or tuple of the unprotected data
  - Tuple of error codes

Exception

InvalidSessionError : This exception is thrown if the session is invalid or has timed out.

`UnprotectError` : This exception is thrown if the API is unable to unprotect the data.

> **Note:**
> If the *unprotect* API is used with bulk data, then it does not throw any exception. Instead, it only returns an error code.
>
> For more information about the error codes, refer to the section *Application Protectors API Return Codes* in the *Protegrity Troubleshooting Guide 9.1.0.0*.

Example

This section provides examples of the API usage for detokenizing and decrypting the data for each data type.

Data Type	Usage	Refer to
String	Detokenizing string data	*Example: Detokenizing String Data*
	Detokenizing string data with external IV	*Example: Detokenizing String Data with External IV*
	Decrypting string data	*Example: Decrypting String Data*
	Unprotecting string data using FPE	*Example: Unprotecting String Data Using FPE*
	Unprotecting string data using FPE with external IV and external tweak	*Example: Unprotecting String Data Using FPE with External IV and External Tweak*
	Detokenizing bulk string data	*Example: Detokenizing Bulk String Data*
	Detokenizing bulk string data with external IV	*Example: Detokenizing Bulk String Data with External IV*
	Decrypting bulk string data	*Example: Decrypting Bulk String Data*
	Unprotecting bulk string data using FPE	*Example: Unprotecting Bulk String Data Using FPE*
	Unprotecting bulk string data using FPE with external IV and external tweak	*Example: Unprotecting Bulk String Data Using FPE with External IV and External Tweak*
Integer	Detokenizing integer data	*Example: Detokenizing Integer Data*
	Detokenizing integer data with external IV	*Example: Detokenizing Integer Data with External IV*
	Decrypting integer data	*Example: Decrypting Integer Data*
	Detokenizing bulk integer data	*Example: Detokenizing Bulk Integer Data*
	Detokenizing bulk integer data with external IV	*Example: Detokenizing Bulk Integer Data with External IV*
	Decrypting bulk integer data	*Example: Decrypting Bulk Integer Data*
Long	Detokenizing long data	*Example: Detokenizing Long Data*
	Detokenizing long data with external IV	*Example: Detokenizing Long Data with External IV*
	Decrypting long data	*Example: Decrypting Long Data*
	Detokenizing bulk long data	*Example: Detokenizing Bulk Long Data*
	Detokenizing bulk long data with external IV	*Example: Detokenizing Bulk Long Data with External IV*
	Decrypting bulk long data	*Example: Decrypting Bulk Long Data*
Float	Detokenizing float data	*Example: Detokenizing Float Data*
	Decrypting float data	*Example: Decrypting Float Data*
	Detokenizing bulk float data	*Example: Detokenizing Bulk Float Data*
	Decrypting bulk float data	*Example: Decrypting Bulk Float Data*

Data Type	Usage	Refer to
Bytes	Detokenizing bytes data	*Example: Detokenizing Bytes Data*
	Detokenizing bytes data with external IV	*Example: Detokenizing Bytes Data with External IV*
	Decrypting bytes data	*Example: Decrypting Bytes Data*
	Detokenizing bulk bytes data	*Example: Detokenizing Bulk Bytes Data*
	Detokenizing bulk bytes data with external IV	*Example: Detokenizing Bulk Bytes Data with External IV*
	Decrypting bulk bytes data	*Example: Decrypting Bulk Bytes Data*
Date	Detokenizing date object	*Example: Detokenizing Date Object*
	Detokenizng bulk date objects	*Example: Detokenizing Bulk Date Objects*
Unicode*1	Detokenizing unicode object	*Example: Detokenizing Unicode Data*
	Decrypting unicode data	*Example: Decrypting Unicode Data*
	Detokenizing bulk unicode data	*Example: Detokenizing Bulk Unicode Data*
	Decrypting bulk unicode data	*Example: Decrypting Bulk Unicode Data*

Note:
*1 - This data type is only applicable for Python 2.7.

2.4.3.11.1 Example - Detokenizing String Data

This section describes how to use the *unprotect* API for retrieving the original string data from the token data.

Example 1: Input string data

In the following example, the *Protegrity1* string that was tokenized using the *TE_A_N_S23_L2R2_Y* data element, is now detokenized using the same data element.

```
from appython import Protector
protector = Protector()
session = protector.create_session("User1")
output = session.protect("Protegrity1", "TE_A_N_S23_L2R2_Y")
print("Protected Data: %s" %output)
org = session.unprotect(output, "TE_A_N_S23_L2R2_Y")
print("Unprotected Data: %s" %org)
```

Result

```
Protected Data: Pr9zdglWRy1
Unprotected Data: Protegrity1
```

Example 2: Input date passed as a string

In the following example, the *29/05/1998* string that was tokenized using the *TE_Date_DMY_S13* Date data element, is now detokenized using the same data element.

Caution: If you have provided the date string as an input in a specific format, then you must use the data element with the same tokenization type as that of the input date format to protect the data. For example, if you have provided the input date string in DD/MM/YYYY format, then you must use only the Date (DD/MM/YYYY) data element to protect the data.

Note: For information about the date formats supported by Protegrity tokenization methods, refer to the section *Protegrity Tokenization* in the *Protection Methods Reference Guide 9.1.0.0*.

```
from appython import Protector
protector = Protector()
session = protector.create_session("User1")
output = session.protect("29/05/1998", "TE_Date_DMY_S13")
print("Protected data: "+str(output))
org = session.unprotect(output, "TE_Date_DMY_S13")
print("Unprotected data: "+str(org))
```

Result

```
Protected data: 08/07/2443
Unprotected data: 29/05/1998
```

Example 3: Input date and time passed as a string

In the following example, the *1998/05/29 10:54:47* string that was tokenized using the *TE_Datetime_TN_DN_M* Datetime data element is now detokenized using the same data element.

Caution: If you have provided the date and time string as an input in a specific format, then you must use the data element with the same tokenization type as that of the input date and time format to protect the data. For example, if you have provided the input date and time string in YYYY/MM/DD HH:MM:SS MMM format, then you must use only the Datetime (YYYY-MM-DD HH:MM:SS MMM) data element to protect the data.

Note: For information about the date formats supported by Protegrity tokenization methods, refer to the section *Protegrity Tokenization* in the *Protection Methods Reference Guide 9.1.0.0*.

```
from appython import Protector
protector = Protector()
session = protector.create_session("User1")
output = session.protect("1998/05/29 10:54:47", "TE_Datetime_TN_DN_M")
print("Protected data: "+str(output))
org = session.unprotect(output, "TE_Datetime_TN_DN_M")
print("Unprotected data: "+str(org))
```

Result

```
Protected data: 3311/02/22 10:54:47
Unprotected data: 1998/05/29 10:54:47
```

2.4.3.11.2 Example - Detokenizing String Data with External IV

This section describes how to use the *unprotect* API for retrieving the original string data from token data, using external IV.

Note: If you want to pass the external IV as a keyword argument to the *unprotect* API, then you must pass the external IV as bytes to the API.

Example

In the following example, the *Protegrity1* string that was tokenized using the *TE_A_N_S23_L2R2_Y* data element and the external IV *1234* is now detokenized using the same data element and external IV.

```
from appython import Protector
protector = Protector()
session = protector.create_session("User1")
output = session.protect("Protegrity1", "TE_A_N_S23_L2R2_Y",
         external_iv=bytes("1234", encoding="utf-8"))
print("Protected Data: %s" %output)
org = session.unprotect(output, "TE_A_N_S23_L2R2_Y",
      external_iv=bytes("1234", encoding="utf-8"))
print("Unprotected Data: %s" %org)
```

Result

```
Protected Data: PrksvEshuy1
Unprotected Data: Protegrity1
```

2.4.3.11.3 Example - Decrypting String Data

This section describes how to use the *unprotect* API for decrypting string data.

> **Note:**
> If you want to encrypt the data, then you must use bytes in the *encrypt_to* keyword.

Example

In the following example, the *Protegrity1* string that was encrypted using the *AES256_IV_CRC_KID* data element is now decrypted using the same data element. Therefore, the *decrypt_to* parameter is passed as a keyword argument and its value is set to *str*.

```
from appython import Protector
protector = Protector()
session = protector.create_session("User1")
output = session.protect("Protegrity1", "AES256_IV_CRC_KID",
         encrypt_to=bytes)
print("Encrypted Data: %s" %output)
org = session.unprotect(output, "AES256_IV_CRC_KID", decrypt_to=str)
print("Decrypted Data: %s" %org)
```

Result

```
Encrypted Data: b'>gmA i=w
Decrypted Data: Protegrity1
```

2.4.3.11.4 Example - Unprotecting String Data Using FPE

This section describes how to use the *unprotect* API for unprotecting string data using FPE (FF1).

Example

In the following example, the *protegrity1234ÀÁÂÃÄÅÆÇÈÉ* string that was protected using the *FPE_FF1_AES256_ID_AN_LnRn_ASTNE* data element, is now unprotected using the same data element.

```
from appython import Protector
protector = Protector()
session = protector.create_session("User1")
output = session.protect("protegrity1234ÀÁÂÃÄÅÆÇÈÉ",
         "FPE_FF1_AES256_ID_AN_LnRn_ASTNE")
print("Protected Data: %s" %output)
```

```
org = session.unprotect(output, "FPE_FF1_AES256_ID_AN_LnRn_ASTNE")
print("Unprotected Data: %s" %org)
```

Result

```
Protected Data: NRejBkN7LcBOT4ÀÁÂÃÄÅÆÇÈÉ
Unprotected Data: protegrity1234ÀÁÂÃÄÅÆÇÈÉ
```

2.4.3.11.5 Example - Unprotecting String Data Using FPE with External IV and External Tweak

This section describes how to use the *unprotect* API for unprotecting string data using FPE (FF1), with external IV and tweak.

> **Note:** If you want to pass the external IV and external tweak as keyword arguments to the *unprotect* API, then you must pass the external IV and external tweak as bytes.

Example

In the following example, the *protegrity1234* string that was protected using the *FPE_FF1_AES256_ID_AN_LnRn_ASTNE* data element, is now unprotected using the same data element, external IV, and external tweak.

```
from appython import Protector
protector = Protector()
session = protector.create_session("User1")
output = session.protect("protegrity1234",
        "FPE_FF1_AES256_ASCII_APIP_AN_L2R1_ASTNI_ML2", external_iv=bytes("1234",
encoding="utf-8"),
        external_tweak=bytes("abcdef", encoding="utf-8"))
print("Protected Data: %s" %output)
org = session.unprotect(output,
      "FPE_FF1_AES256_ASCII_APIP_AN_L2R1_ASTNI_ML2", external_iv=bytes("1234",
encoding="utf-8"),
      external_tweak=bytes("abcdef", encoding="utf-8"))
print("Unprotected Data: %s" %org)
```

Result

```
Protected Data: prS6DaU5Dtd5g4
Unprotected Data: protegrity1234
```

2.4.3.11.6 Example - Detokenizing Bulk String Data

This section describes how to use the *unprotect* API for retrieving the original bulk string data from the token data.

Example 1: Input bulk string data

In the following example, *protegrity1234*, *Protegrity1*, and *Protegrity56* strings are stored in a list and used as bulk data, which is tokenized using the *TE_A_N_S23_L2R2_Y* data element. The bulk string data is then detokenized using the same data element.

```
from appython import Protector
protector = Protector()
session = protector.create_session("User1")
data = ["protegrity1234", "Protegrity1", "Protegrity56"]
p_out = session.protect(data, "TE_A_N_S23_L2R2_Y")
print("Protected Data: ")
print(p_out)
out = session.unprotect(p_out[0], "TE_A_N_S23_L2R2_Y")
print("Unprotected Data: ")
print(out)
```

Result

```
Protected Data:
(['prMLJsM8fZUp34', 'Pr9zdglWRy1', 'Pra9Ez5LPG56'], (6, 6, 6))
Unprotected Data:
(['protegrity1234', 'Protegrity1', 'Protegrity56'], (8, 8, 8))
```

6 is the success return code for the protect operation of each element in the list.

8 is the success return code for the unprotect operation of each element in the list.

Example 2: Input bulk string data

In *Example 1*, the unprotected output was a tuple of the detokenized data and the error list. The following example shows how you can tweak the code to ensure that you retrieve the unprotected output and the error list separately, and not as part of a tuple.

```
from appython import Protector
protector = Protector()
session = protector.create_session("User1")
data = protegrity1234
data = [data]*5
p_out, error_list = session.protect(data, "TE_A_N_S23_L2R2_Y")
print("Protected Data: ")
print(p_out)
print("Error List: ")
print(error_list)
org, error_list = session.unprotect(p_out, "TE_A_N_S23_L2R2_Y")
print("Unprotected Data: ")
print(org)
print("Error List: ")
print(error_list)
```

Result

```
Protected Data:
['prMLJsM8fZUp34', 'prMLJsM8fZUp34', 'prMLJsM8fZUp34', 'prMLJsM8fZUp34',
'prMLJsM8fZUp34']
Error List:
(6, 6, 6, 6, 6)
Unprotected Data:
['protegrity1234', 'protegrity1234', 'protegrity1234', 'protegrity1234',
'protegrity1234']
Error List:
(8, 8, 8, 8, 8)
```

6 is the success return code for the protect operation of each element in the list.

8 is the success return code for the unprotect operation of each element in the list.

Example 3: Input dates passed as bulk strings

In the following example, the *14/02/2019* and *11/03/2018* strings are stored in a list and used as bulk data, which is tokenized using the *TE_Date_DMY_S13* Date data element. The bulk string data is then detokenized using the same data element.

Caution: If you have provided the date string as an input in a specific format, then you must use the data element with the same tokenization type as that of the input date format to protect the data. For example, if you have provided the input date string in DD/MM/YYYY format, then you must use only the Date (DD/MM/YYYY) data element to protect the data.

Note: For information about the date formats supported by Protegrity tokenization methods, refer to the section *Protegrity Tokenization* in the *Protection Methods Reference Guide 9.2.0.0*.

```
from appython import Protector
protector = Protector()
session = protector.create_session("User1")
data = ["14/02/2019", "11/03/2018"]
output = session.protect(data, "TE_Date_DMY_S13")
print("Protected data: "+str(output))
org = session.unprotect(output[0], "TE_Date_DMY_S13")
print("Unprotected data: "+str(org))
```

Result

```
Protected data: (['08/07/2443', '17/08/1830'], (6, 6))
Unprotected data: (['14/02/2019', '11/03/2018'], (8, 8))
```

6 is the success return code for the protect operation of each element in the list.

8 is the success return code for the unprotect operation of each element in the list.

Example 4: Input date and time passed as bulk strings

In the following example, the *2019/02/14 10:54:47* and *2019/11/03 11:01:32* strings is used as the data, which is tokenized using the *TE_Datetime_TN_DN_M* Datetime data element. The bulk string data is then detokenized using the same data element.

Caution: If you have provided the date and time string as an input in a specific format, then you must use the data element with the same tokenization type as that of the input date and time format to protect the data. For example, if you have provided the input date and time string in YYYY/MM/DD HH:MM:SS MMM format, then you must use only the Datetime (YYYY-MM-DD HH:MM:SS MMM) data element to protect the data.

Note: For information about the date formats supported by Protegrity tokenization methods, refer to the section *Protegrity Tokenization* in the *Protection Methods Reference Guide 9.2.0.0*.

```
from appython import Protector
protector = Protector()
session = protector.create_session("User1")
data = ["2019/02/14 10:54:47", "2019/11/03 11:01:32"]
output = session.protect(data, "TE_Datetime_TN_DN_M")
print("Protected data: "+str(output))
org = session.unprotect(output[0], "TE_Datetime_TN_DN_M")
print("Unprotected data: "+str(org))
```

Result

```
Protected data: (['3311/02/22 10:54:47', '3311/11/02 11:01:32'], (6, 6))
Unprotected data: (['2019/02/14 10:54:47', '2019/11/03 11:01:32'], (8, 8))
```

6 is the success return code for the protect operation of each element in the list.

8 is the success return code for the unprotect operation of each element in the list.

2.4.3.11.7 Example - Detokenizing Bulk String Data with External IV

This section describes how to use the *unprotect* API for retrieving the original bulk string data from token data using the external IV.

> **Note:** If you want to pass the external IV as a keyword argument to the *unprotect* API, then you must pass the external IV as bytes to the API.

Example

In the following example, *protegrity1234*, *Protegrity1*, and *Protegrity56* strings are stored in a list and used as bulk data, which is tokenized using the *TE_A_N_S23_L2R2_Y* data element, with the help of external IV *123* that is passed as bytes. The bulk string data is then detokenized using the same data element and external IV.

```
from appython import Protector
protector = Protector()
session = protector.create_session("User1")
data = ["protegrity1234", "Protegrity1", "Protegrity56"]
p_out = session.protect(data, "TE_A_N_S23_L2R2_Y",
        external_iv=bytes("123", encoding="UTF-8"))
print("Protected Data: ")
print(p_out)
out = session.unprotect(p_out[0], "TE_A_N_S23_L2R2_Y",
      external_iv=bytes("123", encoding="UTF-8"))
print("Unprotected Data: ")
print(out)
```

Result

```
Protected Data:
(['prv0WozsSjbS34', 'PrtigABOCy1', 'PrvjDdC2TD56'], (6, 6, 6))
Unprotected Data:
(['protegrity1234', 'Protegrity1', 'Protegrity56'], (8, 8, 8))
```

6 is the success return code for the protect operation of each element in the list.

8 is the success return code for the unprotect operation of each element in the list.

2.4.3.11.8 Example - Decrypting Bulk String Data

This section describes how to use the *unprotect* API for decrypting bulk string data.

> **Note:**
> If you want to encrypt the data, then you must use bytes in the *encrypt_to* keyword.

Example

In the following example, *protegrity1234*, *Protegrity1*, and *Protegrity56* strings are stored in a list and used as bulk data, which is encrypted using the *AES256* data element. The bulk string data is then decrypted using the same data element. Therefore, the *decrypt_to* parameter is passed as a keyword argument and its value is set to *str*.

```
from appython import Protector
protector = Protector()
session = protector.create_session("User1")
data = ["protegrity1234", "Protegrity1", "Protegrity56"]
```

```
p_out = session.protect(data, "AES256", encrypt_to=bytes)
print("Encrypted Data: ")
print(p_out)
out = session.unprotect(p_out[0], "AES256", decrypt_to=str)
print("Decrypted Data: ")
print(out)
```

Result

```
Encrypted Data:
([b'\xc9^x\x02)\xcbB\x91}\x7fi\x8a\xce\x8d>H',
b't\x80\xf5\x8d\x9e\x0b+4Lq\x8a\x97\xdb\x8fx\x16',
b'\x87\x08\x938\xf7o~\xab\xa3\xc2L\xa90>\x18_'], (6, 6, 6))
Decrypted Data:
(['protegrity1234', 'Protegrity1', 'Protegrity56'], (8, 8, 8))
```

6 is the success return code for the protect operation of each element in the list.

8 is the success return code for the unprotect operation of each element in the list.

2.4.3.11.9 Example - Unprotecting Bulk String Data Using FPE

This section describes how to use the *unprotect* API for retrieving the original bulk string data from token data using FPE (FF1).

Example

In the following example, *protegrity1234ÀÁ*, *Protegrity1ÆÇÈ*, and *Protegrity56ÀÁÂÃÄÅ* strings are stored in a list and used as bulk data, which is protected using the FPE data element *FPE_FF1_AES256_APIP_AN_LnRn_ASTNE*. The bulk string data is then unprotected using the same data element.

```
from appython import Protector
protector = Protector()
session = protector.create_session("User1")
data = ["protegrity1234ÀÁ", "Protegrity1ÆÇÈ", "Protegrity56ÀÁÂÃÄÅ"]
p_out = sessionr.protect(data, "FPE_FF1_AES256_APIP_AN_LnRn_ASTNE")
print("Protected Data: ")
print(p_out)
out = session.unprotect(p_out[0], "FPE_FF1_AES256_APIP_AN_LnRn_ASTNE")
print("Unprotected Data: ")
print(out)
```

Result

```
Protected Data:
([u'MG01UHDQ8VyON3\xc0\xc1', u'8APfLh3W9TY\xc6\xc7\xc8',
u'4XYdSFURF4bV\xc0\xc1\xc2\xc3\xc4\xc5'], (6, 6, 6))
Unprotected Data:
([u'protegrity1234\xc0\xc1', u'Protegrity1\xc6\xc7\xc8',
u'Protegrity56\xc0\xc1\xc2\xc3\xc4\xc5'], (8, 8, 8))
```

6 is the success return code for the protect operation of each element in the list.

8 is the success return code for the unprotect operation of each element in the list.

2.4.3.11.10 Example - Unprotecting Bulk String Data Using FPE with External IV and External Tweak

This section describes how to use the *unprotect* API for retrieving the original bulk string data from token data using FPE (FF1), using external IV and external tweak.

> **Note:** If you want to pass the external IV and external tweak as keyword arguments to the *protect* API, then you must pass the external IV and external tweak as bytes to the API.

Example

In the following example, *protegrity1234ÀÁ*, *Protegrity1ÆÇÈ*, and *Protegrity56ÀÁÂÃÄÅ* strings are stored in a list and used as bulk data. This bulk data is protected using the FPE data element *FPE_FF1_AES256_APIP_AN_LnRn_ASTNE*, with the help of external IV *1234* and external tweak *xyz* that are both passed as bytes. The protected bulk string data is then unprotected using the same data element, external IV, and external tweak.

```
from appython import Protector
protector = Protector()
session = protector.create_session("User1")
data = ["protegrity1234ÀÁ", "Protegrity1ÆÇÈ", "Protegrity56ÀÁÂÃÄÅ"]
p_out = session.protect(data, "FPE_FF1_AES256_APIP_AN_LnRn_ASTNE",
        external_iv=bytes("1234", encoding="utf-8"), external_tweak=bytes("xyz",
encoding="utf-8"))
print("Protected Data: ")
print(p_out)
out = session.unprotect(p_out[0], "FPE_FF1_AES256_APIP_AN_LnRn_ASTNE",
      external_iv=bytes("1234", encoding="utf-8"),  external_tweak=bytes("xyz",
encoding="utf-8"))
print("Unprotected Data: ")
print(out)
```

Result

```
Protected Data:
([u'WwR5aK2BMoUlcz\xc0\xc1', u'nW6lqjd7NGR\xc6\xc7\xc8',
u'o6eBUZDNuyWU\xc0\xc1\xc2\xc3\xc4\xc5'], (6, 6, 6))
Unprotected Data:
([u'protegrity1234\xc0\xc1', u'Protegrity1\xc6\xc7\xc8',
u'Protegrity56\xc0\xc1\xc2\xc3\xc4\xc5'], (8, 8, 8))
```

6 is the success return code for the protect operation of each element in the list.

8 is the success return code for the unprotect operation of each element in the list.

2.4.3.11.11 Example - Detokenizing Integer Data

This section describes how to use the *unprotect* API for retrieving the original integer data from token data.

Example

In the following example, the integer data *21* that was tokenized using the *TE_INT_4* data element, is now detokenized using the same data element.

```
from appython import Protector
protector = Protector()
session = protector.create_session("User1")
output = session.protect(21, "TE_INT_4")
print("Protected Data: %s" %output)
org = session.unprotect(output, "TE_INT_4")
print("Unprotected Data: %s" %org)
```

Result

```
Protected Data: -2122057622
Unprotected Data: 21
```

2.4.3.11.12 Example - Detokenizing Integer Data with External IV

This section describes how to use the *unprotect* API for retrieving the original integer data from token data, using external IV.

> **Note:** If you want to pass the external IV as a keyword argument to the *unprotect* API, then you must pass the external IV as bytes to the API.

Example

In the following example, the integer data *21* that was tokenized using the *TE_INT_4* data element and the external IV *1234* is now detokenized using the same data element and external IV.

```
from appython import Protector
protector = Protector()
session = protector.create_session("User1")
output = session.protect(21, "TE_INT_4",
         external_iv=bytes("1234", encoding="utf-8"))
print("Protected Data: %s" %output)
org = session.unprotect(output, "TE_INT_4",
      external_iv=bytes("1234", encoding="utf-8"))
print("Unprotected Data: %s" %org)
```

Result

```
Protected Data: -2122057622
Unprotected Data: 21
```

2.4.3.11.13 Example - Decrypting Integer Data

This section describes how to use the *unprotect* API for decrypting integer data.

> **Note:**
> If you want to encrypt the data, then you must use bytes in the *encrypt_to* keyword.

Example

In the following example, the integer data *21* that was encrypted using the *AES256* data element is now decrypted using the same data element. Therefore, the *decrypt_to* parameter is passed as a keyword argument and its value is set to *int*.

```
from appython import Protector
protector = Protector()
session = protector.create_session("User1")
output = session.protect(21, "AES256", encrypt_to=bytes)
print("Encrypted Data: %s" %output)
org = session.unprotect(output, "AES256", decrypt_to=int)
print("Decrypted Data: %s" %org)
```

Result

```
Encrypted Data: b'@upkN'
Decrypted Data: 21
```

2.4.3.11.14 Example - Detokenizing Bulk Integer Data

This section describes how to use the *unprotect* API for retrieving the original bulk integer data from token data.

> **Note:**

> The AP Python APIs support integer values only between -2147483648 and 2147483648, both inclusive.

Example

In the following example, *21*, *42*, and *55* integers are stored in a list and used as bulk data, which is tokenized using the *TE_INT_4* data element. The bulk integer data is then detokenized using the same data element.

```
from appython import Protector
protector = Protector()
session = protector.create_session("User1")
data = [21, 42, 55]
p_out = session.protect(data, "TE_INT_4")
print("Protected Data: ")
print(p_out)
out = session.unprotect(p_out[0], "TE_INT_4")
print("Unprotected Data: ")
print(out)
```

Result

```
Protected Data:
([-1926573911, -1970496120, -814489753], (6, 6, 6))
Unprotected Data:
([21, 42, 55], (8, 8, 8))
```

6 is the success return code for the protect operation of each element in the list.

8 is the success return code for the unprotect operation of each element in the list.

2.4.3.11.15 Example - Detokenizing Bulk Integer Data with External IV

This section describes how to use the *unprotect* API for retrieving the original bulk integer data from token data using external IV.

> **Note:** If you want to pass the external IV as a keyword argument to the *unprotect* API, then you must pass the external IV as bytes to the API.

Example

In the following example, *21*, *42*, and *55* integers are stored in a list and used as bulk data, which is tokenized using the *TE_INT_4* data element, with the help of external IV *1234* that is passed as bytes. The bulk integer data is then detokenized using the same data element and external IV.

```
from appython import Protector
protector = Protector()
session = protector.create_session("User1")
data = [21, 42, 55]
p_out = session.protect(data, "TE_INT_4", external_iv=bytes("1234", encoding="utf-8"))
print("Protected Data: ")
print(p_out)
out = session.unprotect(p_out[0], "TE_INT_4", external_iv=bytes("1234",
encoding="utf-8"))
print("Unprotected Data: ")
print(out)
```

Result

```
Protected Data:
([-2122057622, 1795905968, 228587043], (6, 6, 6))
Unprotected Data:
([21, 42, 55], (8, 8, 8))
```

6 is the success return code for the protect operation of each element in the list.

8 is the success return code for the unprotect operation of each element in the list.

2.4.3.11.16 Example - Decrypting Bulk Integer Data

This section describes how to use the *unprotect* API for decrypting bulk integer data.

> **Note:**
> If you want to encrypt the data, then you must use bytes in the *encrypt_to* keyword.

Example

In the following example, *21*, *42*, and *55* integers are stored in a list and used as bulk data, which is encrypted using the *AES256* data element. The bulk integer data is then decrypted using the same data element. Therefore, the *decrypt_to* parameter is passed as a keyword argument and its value is set to *int*.

```
from appython import Protector
protector = Protector()
session = protector.create_session("User1")
data = [21, 42, 55]
p_out = session.protect(data, "AES256", encrypt_to=bytes)
print("Encrypted Data: ")
print(p_out)
out = session.unprotect(p_out[0], "AES256", decrypt_to=int)
print("Decrypted Data: ")
print(out)
```

Result

```
Encrypted Data:
([b'@\x19\xccu\x04\xc7\xd8\xc1p\xad\xa7\x1fk\xe4N\xd0',
b'"@\xec\x97(\x96\xab\x18\xd0\x99\xd4~\x1e\xf4\xba\xd1',
b'y\xec\x9b+f\xa8\xb1I\xc2=[\x11\xfd\x06\xa1C'], (6, 6, 6))
Decrypted Data:
([21, 42, 55], (8, 8, 8))
```

6 is the success return code for the protect operation of each element in the list.

8 is the success return code for the unprotect operation of each element in the list.

2.4.3.11.17 Example - Detokenizing Long Data

This section describes how to use the *unprotect* API for retrieving the original long data from the token data.

Example

In the following example, the long data *1376235139103947* that was tokenized using the *TE_INT_8* data element, is now detokenized using the same data element.

```
from appython import Protector
protector = Protector()
session = protector.create_session("User1")
output = session.protect(1376235139103947, "TE_INT_8")
print("Protected Data: %s" %output)
org = session.unprotect(output, "TE_INT_8")
print("Unprotected Data: %s" %org)
```

Result

```
Protected Data: -1770169866845757900
Unprotected Data: 1376235139103947
```

2.4.3.11.18 Example - Detokenizing Long Data with External IV

This section describes how to use the *unprotect* API for retrieving the original long data from the token data using external IV.

> **Note:** If you want to pass the external IV as a keyword argument to the *unprotect* API, then you must pass the external IV as bytes to the API.

Example

In the following example, the long data *1376235139103947* that was tokenized using the *TE_INT_8* data element and the external IV *1234* is now detokenized using the same data element and external IV.

```
from appython import Protector
protector = Protector()
session = protector.create_session("User1")
output = session.protect(1376235139103947, "TE_INT_8",
         external_iv=bytes("1234", encoding="utf-8"))
print("Protected Data: %s" %output)
org = session.unprotect(output, "TE_INT_8",
      external_iv=bytes("1234", encoding="utf-8"))
print("Unprotected Data: %s" %org)
```

Result

```
Protected Data: 5846214101577367207
Unprotected Data: 1376235139103947
```

2.4.3.11.19 Example - Decrypting Long Data

This section describes how to use the *unprotect* API for decrypting long data.

> **Note:**
> If you want to encrypt the data, then you must use bytes in the *encrypt_to* keyword.

Example

In the following example, the long data *1376235139103947* that was encrypted using the *AES256* data element is now decrypted using the same data element. Therefore, the *decrypt_to* parameter is passed as a keyword argument and its value is set to *long*.

```
from appython import Protector
protector = Protector()
session = protector.create_session("User1")
output = session.protect(1376235139103947, "AES256", encrypt_to=bytes)
print("Encrypted Data: %s" %output)
org = session.unprotect(output, "AES256", decrypt_to=int)
print("Decrypted Data: %s" %org)
```

Result

```
Encrypted Data: b'Ds▒▒p0Xl<\'
Decrypted Data: 1376235139103947
```

2.4.3.11.20 Example - Detokenizing Bulk Long Data

This section describes how to use the *unprotect* API for retrieving the original bulk long data from the token data.

Example

In the following example, *1376235139103947*, *2396235839173981*, and *9371234126176985* long data are stored in a list and used as bulk data, which is tokenized using the *TE_INT_8* data element. The bulk long data is then detokenized using the same data element.

```
from appython import Protector
protector = Protector()
session = protector.create_session("User1")
data = [1376235139103947, 2396235839173981, 9371234126176985]
p_out = session.protect(data, "TE_INT_8")
print("Protected Data: ")
print(p_out)
out = session.unprotect(p_out[0], "TE_INT_8")
print("Unprotected Data: ")
print(out)
```

Result

```
Protected Data:
([-1770169866845757900L, -8142006510957348982L, -20687656704969669L], (6, 6, 6))
Unprotected Data:
([1376235139103947L, 2396235839173981L, 9371234126176985L], (8, 8, 8))
```

6 is the success return code for the protect operation of each element in the list.

8 is the success return code for the unprotect operation of each element in the list.

2.4.3.11.21 Example - Detokenizing Bulk Long Data with External IV

This section describes how to use the *unprotect* API for retrieving the original bulk long data from the token data using external IV.

> **Note:** If you want to pass the external IV as a keyword argument to the *unprotect* API, then you must pass the external IV as bytes to the API.

Example

In the following example, *1376235139103947*, *2396235839173981*, and *9371234126176985* long data are stored in a list and used as bulk data, which is tokenized using the *TE_INT_8* data element, with the help of external IV *1234* that is passed as bytes. The bulk long data is then detokenized using the same data element and external IV.

```
from appython import Protector
protector = Protector()
session = protector.create_session("User1")
data = [1376235139103947, 2396235839173981, 9371234126176985]
p_out = session.protect(data, "TE_INT_8", external_iv=bytes("1234", encoding="utf-8"))
print("Protected Data: ")
print(p_out)
out = session.unprotect(p_out[0], "TE_INT_8", external_iv=bytes("1234",
encoding="utf-8"))
print("Unprotected Data: ")
print(out)
```

Result

```
Protected Data:
([5846214101577367207L, 5661139619224336475L, 7806173497368534531L], (6, 6, 6))
```

```
Unprotected Data:
([1376235139103947L, 2396235839173981L, 9371234126176985L], (8, 8, 8))
```

6 is the success return code for the protect operation of each element in the list.

8 is the success return code for the unprotect operation of each element in the list.

2.4.3.11.22 Example - Decrypting Bulk Long Data

This section describes how to use the *unprotect* API for decrypting bulk long data.

> **Note:**
> If you want to encrypt the data, then you must use bytes in the *encrypt_to* keyword.

Example

In the following example, *1376235139103947*, *2396235839173981*, and *9371234126176985* long data are stored in a list and used as bulk data, which is encrypted using the *AES256* data element. The bulk long data is then decrypted using the same data element. Therefore, the *decrypt_to* parameter is passed as a keyword argument and its value is set to *long*.

```
from appython import Protector
protector = Protector()
session = protector.create_session("User1")
data = [1376235139103947, 2396235839173981, 9371234126176985]
p_out = session.protect(data, "AES256", encrypt_to=bytes)
print("Encrypted Data: ")
print(p_out)
out = session.unprotect(p_out[0], "AES256", decrypt_to=int)
print("Decrypted Data: ")
print(out)
```

Result

```
Encrypted Data:
([b'\xd5Ds\xb3\xfb\x95\xf2wp0Xl<\\\x1a\x07', b'\xaf\x05aq\xb6\xcd,L`JC4\x87\x87\t\x0b',
b']j@*S\x96\xf5\xf5S<\x08M\xa6\x18\xbf\xda'], (6, 6, 6))
Decrypted Data:
([1376235139103947L, 2396235839173981L, 9371234126176985L], (8, 8, 8))
```

6 is the success return code for the protect operation of each element in the list.

8 is the success return code for the unprotect operation of each element in the list.

2.4.3.11.23 Example - Unprotecting Float Data

This section describes how to use the *unprotect* API for unprotecting float data using a No Encryption data element. You can use this API for access control and auditing.

Example

In the following example, the long data *22.5* that was protected using the *NoEncryption_1* data element, is now unprotected using the same data element.

```
from appython import Protector
protector = Protector()
session = protector.create_session("User1")
output = session.protect(22.5, "NoEncryption_1")
print("Protected Data: %s" %output)
```

```
org = session.unprotect(output, "NoEncryption_1")
print("Unprotected Data: %s" %org)
```

Result

```
Protected Data: 22.5
Unprotected Data: 22.5
```

The input data, the protected output data, and the unprotected data are the same, as we are using a No Encryption data element to protect and unprotect the data.

2.4.3.11.24 Example - Decrypting Float Data

This section describes how to use the *unprotect* API for decrypting float data.

> **Note:**
> If you want to encrypt the data, then you must use bytes in the *encrypt_to* keyword.

Example

In the following example, the float data *22.5* that was encrypted using the *AES256* data element is now decrypted using the same data element. Therefore, the *decrypt_to* parameter is passed as a keyword argument and its value is set to *float*.

```
from appython import Protector
protector = Protector()
session = protector.create_session("User1")
output = session.protect(22.5, "AES256", encrypt_to=bytes)
print("Encrypted Data: %s" %output)
org = session.unprotect(output, "AES256", decrypt_to=float)
print("Decrypted Data: %s" %org)
```

Result

```
Encrypted Data: b'g.O▒Vk▒>'
Decrypted Data: 22.5
```

2.4.3.11.25 Example - Unprotecting Bulk Float Data

This section describes how to use the *unprotect* API for unprotecting bulk float data using a No Encryption data element. You can use this API for access control and auditing.

Example

In the following example, *22.5*, *48.93*, and *94.14* float data are stored in a list and used as bulk data, which is protected using the *NoEncryption_1* data element. The bulk float data is then unprotected using the same data element.

```
from appython import Protector
protector = Protector()
session = protector.create_session("User1")
data = [22.5, 48.93, 94.31]
p_out = session.protect(data, "NoEncryption_1")
print("Protected Data: ")
print(p_out)
out = session.unprotect(p_out[0], "NoEncryption_1")
print("Unprotected Data: ")
print(out)
```

Result

```
Protected Data:
([22.5, 48.93, 94.31], (6, 6, 6))
```

```
Unprotected Data:
([22.5, 48.93, 94.31], (8, 8, 8))
```

6 is the success return code for the protect operation of each element in the list.

8 is the success return code for the unprotect operation of each element in the list.

The input data, the protected output data, and the unprotected data are the same, as we are using a No Encryption data element to protect and unprotect the data.

2.4.3.11.26 Example - Decrypting Bulk Float Data

This section describes how to use the *unprotect* API for decrypting bulk float data.

> **Note:**
> If you want to encrypt the data, then you must use bytes in the *encrypt_to* keyword.

Example

In the following example, *22.5*, *48.93*, and *94.14* float data are stored in a list and used as bulk data, which is encrypted using the *AES256* data element. The bulk float data is then decrypted using the same data element. Therefore, the *decrypt_to* parameter is passed as a keyword argument and its value is set to *float*.

```
from appython import Protector
protector = Protector()
session = protector.create_session("User1")
data = [22.5, 48.93, 94.31]
p_out = session.protect(data, "AES256", encrypt_to=bytes)
print("Encrypted Data: ")
print(p_out)
out = session.unprotect(p_out[0], "AES256", decrypt_to=float)
print("Decrypted Data: ")
print(out)
```

Result

```
Encrypted Data:
([b'g.O\xd8\x8b\x12\x89\x15Vk\x88\xbe\xf4;\x18>',
b'.\xb0Q\xb9\xc9\xca\xba\xc2\xcb8\xfe\xd8\xf4q\x00\xb8',
b'\xb6x\xf4\x94l9\xe6uaN\x83\x8d\n\x98\n;'], (6, 6, 6))
Decrypted Data:
([22.5, 48.93, 94.31], (8, 8, 8))
```

6 is the success return code for the protect operation of each element in the list.

8 is the success return code for the unprotect operation of each element in the list.

2.4.3.11.27 Example - Detokenizing Bytes Data

This section describes how to use the *unprotect* API for retrieving the original bytes data from the token data.

Example

In the following example, the bytes data *b'Protegrity1'* that was tokenized using the *TE_A_N_S23_L2R2_Y* data element, is now detokenized using the same data element.

```
from appython import Protector
protector = Protector()
```

```
session = protector.create_session("User1")
data=bytes("Protegrity1", encoding="utf-8")
p_out = session.protect(data, "TE_A_N_S23_L2R2_Y")
print("Protected Data: %s" %p_out)
org = session.unprotect(p_out, "TE_A_N_S23_L2R2_Y")
print("Unprotected Data: %s" %org)
```

Result

```
Protected Data: b'Pr9zdglWRy1'
Unprotected Data: b'Protegrity1'
```

2.4.3.11.28 Example - Detokenizing Bytes Data with External IV

This section describes how to use the *unprotect* API for retrieving the original bytes data from the token data using external IV.

Example

In the following example, the bytes data *b'Protegrity1'* that was tokenized using the *TE_A_N_S23_L2R2_Y* data element and the external IV *1234* is now detokenized using the same data element and external IV.

```
from appython import Protector
protector = Protector()
session = protector.create_session("User1")
data=bytes("Protegrity1", encoding="utf-8")
p_out = session.protect(data, "TE_A_N_S23_L2R2_Y",
        external_iv=bytes("1234", encoding="utf-8"))
print("Protected Data: %s" %p_out)
org = session.unprotect(p_out, "TE_A_N_S23_L2R2_Y",
      external_iv=bytes("1234", encoding="utf-8"))
print("Unprotected Data: %s" %org)
```

Result

```
Protected Data: b'PrksvEshuy1'
Unprotected Data: b'Protegrity1'
```

2.4.3.11.29 Example - Decrypting Bytes Data

This section describes how to use the *unprotect* API for decrypting bytes data.

Example

In the following example, the bytes data *b'Protegrity1'* that was encrypted using the *AES256* data element, is now decrypted using the same data element. Therefore, the *decrypt_to* parameter is passed as a keyword argument and its value is set to *bytes*.

```
from appython import Protector
protector = Protector()
session = protector.create_session("User1")
data=bytes("Protegrity1", encoding="utf-8")
p_out = session.protect(data, "AES256", encrypt_to=bytes)
print("Encrypted Data: %s" %p_out)
org = session.unprotect(p_out, "AES256", decrypt_to=bytes)
print("Decrypted Data: %s" %org)
```

Result

```
Encrypted Data: b't▒4Lq▒'
Decrypted Data: b'Protegrity1'
```

2.4.3.11.30 Example - Detokenizing Bulk Bytes Data

This section describes how to use the *unprotect* API for retrieving the original bulk bytes data from the token data.

Example

In the following example, *protegrity1234*, *Protegrity1*, and *Protegrity56* strings are first converted to bytes using the Python *bytes()* method. The converted bytes are then stored in a list and used as bulk data, which is tokenized using the *TE_A_N_S23_L2R2_Y* data element. The bulk bytes data is then detokenized using the same data element.

```
from appython import Protector
protector = Protector()
session = protector.create_session("User1")
data = [bytes("protegrity1234"), bytes("Protegrity1"), bytes("Protegrity56")]
p_out = session.protect(data, "TE_A_N_S23_L2R2_Y")
print("Protected Data: ")
print(p_out)
org = session.unprotect(p_out[0], "TE_A_N_S23_L2R2_Y")
print("Unprotected Data: ")
print(org)
```

Result

```
Protected Data:
([b'prMLJsM8fZUp34', b'Pr9zdglWRy1', b'Pra9Ez5LPG56'], (6, 6, 6))
Unprotected Data:
([b'protegrity1234', b'Protegrity1', b'Protegrity56'], (8, 8, 8))
```

6 is the success return code for the protect operation of each element in the list.

8 is the success return code for the unprotect operation of each element in the list.

2.4.3.11.31 Example - Detokenizing Bulk Bytes Data with External IV

This section describes how to use the *unprotect* API for retrieving the original bulk bytes data from the token data using external IV.

Example

In the following example, *protegrity1234*, *Protegrity1*, and *Protegrity56* strings are first converted to bytes using the Python *bytes()* method. The converted bytes are then stored in a list and used as bulk data, which is tokenized using the *TE_A_N_S23_L2R2_Y* data element, with the help of external IV *1234* that is passed as bytes. The bulk bytes data is then detokenized using the same data element and external IV.

```
from appython import Protector
protector = Protector()
session = protector.create_session("User1")
data = [bytes("protegrity1234"), bytes("Protegrity1"), bytes("Protegrity56")]
p_out = session.protect(data, "TE_A_N_S23_L2R2_Y",
        external_iv=bytes("1234"))
print("Protected Data: ")
print(p_out)
org = session.unprotect(p_out[0], "TE_A_N_S23_L2R2_Y",
      external_iv=bytes("1234"))
print("Unprotected Data: ")
print(org)
```

Result

```
Protected Data:
([b'prbm147L5pc434', b'PrksvEshuy1', b'Prmx0hG8Nj56'], (6, 6, 6))
Unprotected Data:
([b'protegrity1234', b'Protegrity1', b'Protegrity56'], (8, 8, 8))
```

6 is the success return code for the protect operation of each element in the list.

8 is the success return code for the unprotect operation of each element in the list.

2.4.3.11.32 Example - Decrypting Bulk Bytes Data

This section describes how to use the *unprotect* API for decrypting bulk bytes data.

Example

In the following example, *protegrity1234*, *Protegrity1*, and *Protegrity56* strings are first converted to bytes using the Python *bytes()* method. The converted bytes are then stored in a list and used as bulk data, which is encrypted using the *AES256* data element. The bulk bytes data is then decrypted using the same data element. Therefore, the *decrypt_to* parameter is passed as a keyword argument and its value is set to *bytes*.

```
from appython import Protector
protector = Protector()
session = protector.create_session("User1")
data = [bytes("protegrity1234", encoding ="UTF-8"), bytes("Protegrity1", encoding
="UTF-8"), bytes("Protegrity56", encoding ="UTF-8")]
p_out = session.protect(data, "AES256", encrypt_to=bytes)
print("Encrypted Data: ")
print(p_out)
org = session.unprotect(p_out[0], "AES256", decrypt_to=bytes)
print("Decrypted Data: ")
print(org)
```

Result

```
Encrypted Data:
([b'\xc9^x\x02)\xcbB\x91}\x7fi\x8a\xce\x8d>H',
b't\x80\xf5\x8d\x9e\x0b+4Lq\x8a\x97\xdb\x8fx\x16',
b'\x87\x08\x938\xf7o~\xab\xa3\xc2L\xa90>\x18_'], (6, 6, 6))
Decrypted Data:
([b'protegrity1234', b'Protegrity1', b'Protegrity56'], (8, 8, 8))
```

6 is the success return code for the protect operation of each element in the list.

8 is the success return code for the unprotect operation of each element in the list.

2.4.3.11.33 Example - Detokenizing Date Objects

This section describes how to use the *unprotect* API for retrieving the original data objects from token data.

Caution: If you have provided the date object as an input in a specific format, then you must use the data element with the same tokenization type as that of the input date format to protect the data. For example, if you have provided the input date object in DD/MM/YYYY format, then you must use only the Date (DD/MM/YYYY) data element to protect the data.

Note: For information about the date formats supported by Protegrity tokenization methods, refer to the section *Protegrity Tokenization* in the *Protection Methods Reference Guide 9.1.0.0*.

Example 1: Input date object in DD/MM/YYYY format

In the following example, the *12/02/2019* date string is used as the data, which is first converted to a date object using the Python *date* method of the *datetime* module.

The date object is then tokenized using the *TE_Date_DMY_S13* data element, and then detokenized using the same data element.

```
from appython import Protector
from datetime import datetime
protector = Protector()
session = protector.create_session("User1")
data = datetime.strptime("12/02/2019", "%d/%m/%Y").date()
```

```
print("Input date as a Date object : "+str(data))
p_out = session.protect(data, "TE_Date_DMY_S13")
print("Protected date: "+str(p_out))
unprotected_output = session.unprotect(p_out, "TE_Date_DMY_S13")
print("Unprotected date: "+str(unprotected_output))
```

Result

```
Input date as a Date object : 2019-02-12
Protected date: 1896-10-21
Unprotected date: 2019-02-12
```

Example 2: Input date object in MM.DD.YYYY format

In the following example, the *02/12/2019* date string is used as the data, which is first converted to a date object using the Python *date* method of the *datetime* module.

The date object is then tokenized using the *TE_Date_MDY_S13* data element, and then detokenized using the same data element.

```
from appython import Protector
from datetime import datetime
protector = Protector()
session = protector.create_session("User1")
data = datetime.strptime("02/12/2019", "%m/%d/%Y").date()
print("\nInput date as a Date object : "+str(data))
p_out = session.protect(data, "TE_Date_MDY_S13")
print("Protected date: "+str(p_out))
unprotected_output = session.unprotect(p_out, "TE_Date_MDY_S13")
print("Unprotected date: "+str(unprotected_output))
```

Result

```
Input date as a Date object : 2019-02-12
Protected date: 2037-06-12
Unprotected date: 2019-02-12
```

Example 3: Input date object in YYYY-MM-DD format

In the following example, the *2019/02/12* date string is used as the data, which is first converted to a date object using the Python *date* method of the *datetime* module.

The date object is then tokenized using the *TE_Date_ YMD_S13* data element, and then detokenized using the same data element.

```
from appython import Protector
from datetime import datetime
protector = Protector()
session = protector.create_session("User1")
data = datetime.strptime("2019/02/12", "%Y/%m/%d").date()
print("\nInput date as a Date object : "+str(data))
p_out = session.protect(data, "TE_Date_YMD_S13")
print("Protected date: "+str(p_out))
unprotected_output = session.unprotect(p_out, "TE_Date_YMD_S13")
print("Unprotected date: "+str(unprotected_output))
```

Result

```
Input date as a Date object : 2019-02-12
Protected date: 2615-12-23
Unprotected date: 2019-02-12
```

2.4.3.11.34 Example - Detokenizing Bulk Date Objects

This section describes how to use the *unprotect* API for retrieving the original bulk date objects from the token data.

Caution: If you have provided the date object as an input in a specific format, then you must use the data element with the same tokenization type as that of the input date format to protect the data. For example, if you have provided the input date object in DD/MM/YYYY format, then you must use only the Date (DD/MM/YYYY) data element to protect the data.

Note: For information about the date formats supported by Protegrity tokenization methods, refer to the section *Protegrity Tokenization* in the *Protection Methods Reference Guide 9.1.0.0*.

Example: Input as a Date Object

In the following example, the *12/02/2019* and *11/01/2018* date strings are used as the data, which are first converted to a date objects using the Python *date* method of the *datetime* module. The two date objects are then used to create a list, which is used as the input data.

The input list is then tokenized using the *TE_Date_DMY_S13* data element, and then detokenized using the same data element.

```
from appython import Protector
from datetime import datetime
protector = Protector()
session = protector.create_session("User1")
data1 = datetime.strptime("12/02/2019", "%d/%m/%Y").date()
data2 = datetime.strptime("11/01/2018", "%d/%m/%Y").date()
data = [data1, data2]
print("Input data: "+str(data))
p_out = session.protect(data, "TE_Date_DMY_S13")
print("Protected data: "+str(p_out))
unprotected_output = session.unprotect(p_out[0], "TE_Date_DMY_S13")
print("Unprotected date: "+str(unprotected_output))
```

Result

```
Input data:  [datetime.date(2019, 2, 12), datetime.date(2018, 1, 11)]
Protected data: ([datetime.date(1896, 10, 21), datetime.date(696, 3, 1)], (6, 6))
Unprotected date: ([datetime.date(2019, 2, 12), datetime.date(2018, 1, 11)], (8, 8))
```

6 is the success return code for the protect operation of each element in the list.

8 is the success return code for the unprotect operation of each element in the list.

2.4.3.11.35 Example - Detokenizing Unicode Data

This section describes how to use the *unprotect* API for retrieving the original unicode data from the token data.

Example

In the following example, the *u'protegrity1234ÀÁÂÃÄÅÆÇÈÉ'* unicode data that was tokenized using the *TE_A_N_S23_L2R2_Y* data element, is now detokenized using the same data element.

```
from appython import Protector
protector = Protector()
session = protector.create_session("User1")
output = session.protect(u'protegrity1234ÀÁÂÃÄÅÆÇÈÉ', "TE_A_N_S23_L2R2_Y")
print("Protected Data: %s" %output)
org = session.unprotect(output, "TE_A_N_S23_L2R2_Y")
print("Unprotected Data: %s" %org)
```

Result

```
Protected Data: prZeslalwuQQy3ÀÁÂÃÄÅÆÇÈÉ
Unprotected Data: protegrity1234ÀÁÂÃÄÅÆÇÈÉ
```

2.4.3.11.36 Example - Decrypting Unicode Data

This section describes how to use the *unprotect* API for decrypting unicode data.

> **Note:**
> If you want to encrypt the data, then you must use bytes in the *encrypt_to* keyword.

Example

In the following example, the *u'protegrity1234ÀÁÂÃÄÅÆÇÈÉ'* unicode data that was encrypted using the *AES256_IV_CRC_KID* data element is now decrypted using the same data element. Therefore, the *decrypt_to* parameter is passed as a keyword argument and its value is set to *unicode*.

```
from appython import Protector
protector = Protector()
session = protector.create_session("User1")
output = session.protect(u'protegrity1234ÀÁÂÃÄÅÆÇÈÉ', "AES256_IV_CRC_KID",
         encrypt_to=bytes)
print("Encrypted Data: %s" %output)
org = session.unprotect(output, "AES256_IV_CRC_KID", decrypt_to=unicode)
print("Decrypted Data: %s" %org)
```

Result

```
Encrypted Data: b' 8"+[/O*wMaX{B[u]E(R1!wt?V6QW]G'
Decrypted Data: protegrity1234ÀÁÂÃÄÅÆÇÈÉ
```

2.4.3.11.37 Example - Detokenizing Bulk Unicode Data

This section describes how to use the *unprotect* API for retrieving the original bulk unicode data from the token data.

Example

In the following example, *u'protegrity1234ÀÁÂÃÄÅÆÇÈÉ'*, *u'Protegrity1ÆÇÈÉÀÁÂÃÄÅ'*, and *u'Protegrity56ÇÅÆÈÉÂÃ'* unicode data are stored in a list and used as bulk data, which is tokenized using the *TE_A_N_S23_L2R2_Y* data element. The bulk unicode data is then detokenized using the same data element.

```
from appython import Protector
protector = Protector()
session = protector.create_session("User1")
data = [u'protegrity1234ÀÁÂÃÄÅÆÇÈÉ', u'Protegrity1ÆÇÈÉÀÁÂÃÄÅ', u'Protegrity56ÇÅÆÈÉÂÃ']
p_out = session.protect(data, "TE_A_N_S23_L2R2_Y")
print("Protected Data: ")
print(p_out)
out = session.unprotect(p_out[0], "TE_A_N_S23_L2R2_Y")
print("Unprotected Data: ")
print(out)
```

Result

```
Protected Data:
([u'prZeslalwuQQy3\xc0\xc1\xc2\xc3\xc4\xc5\xc6\xc7\xc8\xc9',
u'PrVt6rfyW81\xc6\xc7\xc8\xc9\xc0\xc1\xc2\xc3\xc4\xc5',
u'PrFgczleNkNG\xc7\xc5\xc6\xc8\xc9\xc2\xc3'], (6, 6, 6))
Unprotected Data:
([u'protegrity1234\xc0\xc1\xc2\xc3\xc4\xc5\xc6\xc7\xc8\xc9',
```

```
u'Protegrity1\xc6\xc7\xc8\xc9\xc0\xc1\xc2\xc3\xc4\xc5',
u'Protegrity56\xc7\xc5\xc6\xc8\xc9\xc2\xc3'], (8, 8, 8))
```

6 is the success return code for the protect operation of each element in the list.

8 is the success return code for the unprotect operation of each element in the list.

2.4.3.11.38 Example - Decrypting Bulk Unicode Data

This section describes how to use the *unprotect* API for decrypting bulk unicode data.

> **Note:**
> If you want to encrypt the data, then you must use bytes in the *encrypt_to* keyword.

Example

In the following example, *u'protegrity1234ÀÁÂÃÄÅÆÇÈÉ'*, *u'Protegrity1ÆÇÈÉÀÁÂÃÄÅ'*, and *u'Protegrity56ÇÅÆÈÉÂÃ'* unicode data are stored in a list and used as bulk data, which is encrypted using the *AES256* data element. The bulk unicode data is then decrypted using the same data element. Therefore, the *decrypt_to* parameter is passed as a keyword argument and its value is set to *unicode*.

```
from appython import Protector
protector = Protector()
session = protector.create_session("User1")
data = [u'protegrity1234ÀÁÂÃÄÅÆÇÈÉ', u'Protegrity1ÆÇÈÉÀÁÂÃÄÅ', u'Protegrity56ÇÅÆÈÉÂÃ']
p_out = session.protect(data, "AES256", encrypt_to=bytes)
print("Encrypted Data: ")
print(p_out)
out = session.unprotect(p_out[0], "AES256", decrypt_to=str)
print("Decrypted Data: ")
print(out)
```

Result

```
Encrypted Data:
([b'F2\xd2\xddR\xda\x9e7#\xfc\xe6\xe2Ore\x18>=\x87\xfc\xea\x9c\xb8\x94\x9e$M?
\x9a\xec\xefO5\xc3\x8fjun\xe3\r4\x0f\xedD76\xe4\xfa',
b'\x9f\xc0}G\x12\x1bu\x02\xfdMO\x8e\x01\xb6\x0f\xf5\xbbi\xbe\xc9\x11J\x1c\xa4\x12\x1e\xf
0\xbeA\x19\xa4\xc3', b'G\xa3(\xee\xb7\x81m\xfc\x96-I\xa2\x9eGt\xcc\x0b-
\x97\xc73\x00O\xdc\xfb\t.\xfa=\x99:\xe7'], (6, 6, 6))
Decrypted Data:
([u'protegrity1234\xc0\xc1\xc2\xc3\xc4\xc5\xc6\xc7\xc8\xc9',
u'Protegrity1\xc6\xc7\xc8\xc9\xc0\xc1\xc2\xc3\xc4\xc5',
u'Protegrity56\xc7\xc5\xc6\xc8\xc9\xc2\xc3'], (8, 8, 8))
```

6 is the success return code for the protect operation of each element in the list.

8 is the success return code for the unprotect operation of each element in the list.

2.4.3.12 reprotect

The *reprotect* API reprotects data using tokenization, data type preserving encryption, No Encryption, or encryption data element. The protected data is first unprotected and then protected again with a new data element. It supports bulk protection without a maximum data limit. However, you are recommended not to pass more than 1 MB of input data for each protection call.

For String and Byte data types, the maximum length for tokenization is 4096 bytes, while no maximum length is defined for encryption.

Warning: If you are retokenizing the data using the *reprotect* API, then the old data element and the new data element must have the same tokenization type. For example, if you have used Alpha-Numeric data element to protect the data, then you must use only Alpha-Numeric data element to reprotect the data.

def reprotect(self, data, old_de, new_de, **kwargs)

Caution: Do not pass the *self* parameter while invoking the API.

Parameters

data: Protected data to be reprotected. The data is first unprotected with the old data element and then protected with the new data element.

old_de: String containing the data element name defined in the policy for the input data. This data element is used to unprotect the protected data as part of the reprotect operation.

new_de: String containing the data element name defined in the policy to create the output data. This data element is used to protect the data as part of the reprotect operation.

*****kwargs***: Specify one or more of the following keyword arguments:

- *old_external_iv*: Specify the old external IV in bytes for Tokenization and FPE protection methods. This old external IV is used to unprotect the protected data as part of the reprotect operation. This argument is optional.
- *new_external_iv*: Specify the new external IV in bytes for Tokenization and FPE protection methods. This new external IV is used to protect the data as part of the reprotect operation. This argument is optional.
- *old_external_tweak*: Specify the old external tweak value in bytes for the FPE protection method. This old external tweak is used to unprotect the protected data as part of the reprotect operation. This argument is optional.
- *new_external_tweak*: Specify the new external tweak value in bytes for the FPE protection method. This new external tweak is used to protect the data as part of the reprotect operation. This argument is optional.
- *encrypt_to*: Specify this argument for re-encrypting the bytes data and set its value to *bytes*. This argument is Mandatory. This argument must not be used for Tokenization and FPE protection methods.

Note: Keyword arguments are case sensitive.

Returns

- *For single data*: Returns the reprotected data
- *For bulk data*: Returns a tuple of the following data:
  - List or tuple of the reprotected data
  - Tuple of error codes

Exception

InvalidSessionError : This exception is thrown if the session is invalid or has timed out.

ReprotectError : This exception is thrown if the API is unable to reprotect the data.

Note:

If the *reprotect* API is used with bulk data, then it does not throw any exception. Instead, it only returns an error code.

For more information regarding the error codes, refer to the section *Application Protectors API Return Codes* in the *Protegrity Troubleshooting Guide 9.2.0.0*.

Example

This section provides examples of the API usage for retokenizing and re-encrypting the data for each data type.

Data Type	Usage	Refer to
String	Retokenizing string data	*Example: Retokenizing String Data*
	Retokenizing string data with external IV	*Example: Tokenizing String Data with External IV*
	Reprotecting string data using FPE	*Example: Reprotecting String Data Using FPE*
	Reprotecting string data using FPE with external IV and external Tweak	*Example: Reprotecting String Data Using FPE with External IV and External Tweak*
	Retokenizing bulk string data	*Example: Retokenizing Bulk String Data*
	Retokenizing bulk string data with external IV	*Example: Retokenizing Bulk String Data with External IV*
	Reprotecting bulk string data using FPE	*Example: Reprotecting Bulk String Data Using FPE*
	Reprotecting bulk string data using FPE with external IV and external tweak	*Example: Reprotecting Bulk String Data Using FPE with External IV and External Tweak*
Integer	Retokenizing integer data	*Example: Retokenizing Integer Data*
	Retokenizing integer data with external IV	*Example: Retokenizing Integer Data with External IV*
	Retokenizing bulk integer data	*Example: Retokenizing Bulk Integer Data*
	Retokenizing bulk integer data with external IV	*Example: Retokenizing Bulk Integer Data with External IV*
Long	Retokenizing long data	*Example: Retokenizing Long Data*
	Retokenizing long data with external IV	*Example: Retokenizing Long Data with External IV*
	Retokenizing bulk long data	*Example: Retokenizing Bulk Long Data*
	Retokenizing bulk long data with external IV	*Example: Retokenizing Bulk Long Data with External IV*
Float	Retokenizing float data	*Example: Retokenizing Float Data*
	Retokenizing bulk float data	*Example: Retokenizing Bulk Long Data*
Bytes	Retokenizing bytes data	*Example: Retokenizing Bytes Data*
	Retokenizing bytes data with external IV	*Example: Retokenizing Bytes Data with External IV*
	Re-encrypting bytes data	*Example: Re-Encrypting Bytes Data*
	Retokenizing bulk bytes data	*Example: Retokenizing Bulk Bytes Data*
	Retokenizing bulk bytes data with external IV	*Example: Retokenizing Bulk Bytes Data with External IV*
	Re-encrypting bulk bytes data	*Example: Re-Encrypting Bulk Bytes Data*
Date	Retokenizing date object	*Example: Retokenizing Date Object*
	Retokenizng bulk date objects	*Example: Retokenizing Bulk Date Objects*
Unicode[*1]	Retokenizing unicode data	*Example: Retokenizing Unicode Data*
	Retokenizng bulk unicode data	*Example: Retokenizing Unicode Data*

Note:

[*1] - This data type is only applicable for Python 2.7.

2.4.3.12.1 Example - Retokenizing String Data

This section describes how to use the *reprotect* API for retokenizing string data.

Warning: If you are retokenizing the data using the *reprotect* API, then the old data element and the new data element must have the same tokenization type. For example, if you have used the Alpha-Numeric data element to protect the data, then you must use only the Alpha-Numeric data element to reprotect the data.

Example 1: Input string data

In the following example, the *Protegrity1* string is used as the input data, which is first tokenized using the *TE_A_N_S23_L2R2_Y* data element.

The tokenized input data, the old data element *TE_A_N_S23_L2R2_Y*, and a new data element *TE_A_N_S23_L0R0_Y* are then passed as inputs to the *reprotect* API. The *reprotect* API first detokenizes the protected input data using the old data element and then retokenizes it using the new data element, as part of a single reprotect operation.

```
from appython import Protector
protector = Protector()
session = protector.create_session("User1")
output = session.protect("Protegrity1", "TE_A_N_S23_L2R2_Y")
print("Protected Data: %s" %output)
r_out = session.reprotect(output, "TE_A_N_S23_L2R2_Y",
        "TE_A_N_S23_L0R0_Y")
print("Reprotected Data: %s" %r_out)
```

Result

```
Protected Data: Pr9zdglWRy1
Reprotected Data: 7gD6aY1Aja9
```

Example 2: Input date passed as a string

In the following example, the *14/02/2019* string is used as the input data, which is first tokenized using the *TE_Date_DMY_S13* Date data element.

Caution: If you have provided the date string as an input in a specific format, then you must use the data element with the same tokenization type as that of the input date format to protect the data. For example, if you have provided the input date string in DD/MM/YYYY format, then you must use only the Date (DD/MM/YYYY) data element to protect the data.

Note: For information about the date formats supported by Protegrity tokenization methods, refer to the section *Protegrity Tokenization* in the *Protection Methods Reference Guide 9.1.0.0*.

The tokenized input data, the old data element *TE_Date_DMY_S13*, and a new data element *TE_Date_DMY_S16* are then passed as inputs to the *reprotect* API. The *reprotect* API first detokenizes the protected input data using the old data element and then retokenizes it using the new data element, as part of a single reprotect operation.

```
from appython import Protector
protector = Protector()
session = protector.create_session("User1")
output = session.protect("14/02/2019", "TE_Date_DMY_S13")
print("Protected data: "+str(output))
r_out = session.reprotect(output, "TE_Date_DMY_S13", "TE_Date_DMY_S16")
print("Reprotected data: "+str(r_out))
```

Result

```
Protected data: 08/07/2443
Reprotected data: 19/10/1231
```

Example 3: Input date and time passed as a string

In the following example, the *2019/02/14 10:54:47* string is used as the input data, which is first tokenized using the *TE_Datetime_TN_DN_M* Datetime data element.

> **Caution:** If you have provided the date and time string as an input in a specific format, then you must use the data element with the same tokenization type as that of the input date and time format to protect the data. For example, if you have provided the input date and time string in YYYY/MM/DD HH:MM:SS format, then you must use only the Datetime (YYYY-MM-DD HH:MM:SS MMM) data element to protect the data.

> **Note:** For information about the date formats supported by Protegrity tokenization methods, refer to the section *Protegrity Tokenization* in the *Protection Methods Reference Guide 9.1.0.0*.

The tokenized input data, the old data element *TE_Datetime_TN_DN_M*, and a new data element *TE_Datetime_TN_DN_Y* are then passed as inputs to the *reprotect* API. The *reprotect* API first detokenizes the protected input data using the old data element and then retokenizes it using the new data element, as part of a single reprotect operation.

```
from appython import Protector
protector = Protector()
session = protector.create_session("User1")
output = session.protect("2019/02/14 10:54:47", "TE_Datetime_TN_DN_M")
print("Protected data: "+str(output))
r_out = session.reprotect(output, "TE_Datetime_TN_DN_M", "TE_Datetime_TN_DN_Y")
print("Reprotected data: "+str(r_out))
```

Result

```
Protected data: 3311/02/22 10:54:47
Reprotected data: 2019/09/25 10:54:47
```

2.4.3.12.2 Example - Retokenizing String Data with External IV

This section describes how to use the *reprotect* API for retokenizing string data using external IV.

> **Warning:** If you are retokenizing the data using the *reprotect* API, then the old data element and the new data element must have the same tokenization type. For example, if you have used the Alpha-Numeric data element to protect the data, then you must use only the Alpha-Numeric data element to reprotect the data.

> **Note:** If you want to pass the external IV as a keyword argument to the *reprotect* API, then you must pass the external IV as bytes to the API.

Example

In the following example, the *Protegrity1* string is used as the input data, which is first tokenized using the *TE_A_N_S23_L2R2_Y* data element, with the help of external IV *1234* that is passed as bytes.

The tokenized input data, the *TE_A_N_S23_L2R2_Y* data element, the old external IV *1234* in bytes, and a new external IV *123456* in bytes are then passed as inputs to the *reprotect* API. As part of a single reprotect operation, the *reprotect* API first detokenizes the protected input data using the given data element and old external IV, and then retokenizes it using the same data element, but with the new external IV.

```
from appython import Protector
protector = Protector()
session = protector.create_session("User1")
p_out = session.protect("Protegrity1", "TE_A_N_S23_L2R2_Y",
        external_iv=bytes("1234", encoding="utf-8"))
```

```
print("Protected Data: %s" %p_out)
r_out = session.reprotect(p_out, "TE_A_N_S23_L2R2_Y",
        "TE_A_N_S23_L2R2_Y", old_external_iv=bytes("1234", encoding="utf-8"),
        new_external_iv=bytes("123456", encoding="utf-8"))
print("Reprotected Data: %s" %r_out)
```

Result

```
Protected Data: PrksvEshuy1
Reprotected Data: PrKxfmdTGy1
```

2.4.3.12.3 Example - Reprotecting String Data Using FPE

This section describes how to use the *reprotect* API for reprotecting string data using FPE (FF1).

Warning: If you are using FPE with the *reprotect* API, then ensure that the plaintext alphabet type and the plaintext encoding used for FPE must be the same for both protecting and reprotecting the data. For example, if you have used FPE-Numeric data element with UTF-8 encoding to protect the data, then you must use only FPE-Numeric data element with UTF-8 encoding to reprotect the data.

Example

In the following example, the *protegrity1234ÀÁÂÃÄÅÆÇÈÉ* string is used as the input data, which is first protected using the FPE data element *FPE_FF1_AES256_ID_AN_LnRn_ASTNE*.

The protected input data, the old data element *FPE_FF1_AES256_ID_AN_LnRn_ASTNE*, and a new data element *FPE_FF1_AES256_ID_AN_LnRn_ASTNI* are then passed as inputs to the *reprotect* API. The *reprotect* API first unprotects the protected input data using the old data element and then reprotects it using the new data element, as part of a single reprotect operation.

```
from appython import Protector
protector = Protector()
session = protector.create_session("User1")
p_out = session.protect("protegrity1234ÀÁÂÃÄÅÆÇÈÉ",
        "FPE_FF1_AES256_ID_AN_LnRn_ASTNE")
print("Protected Data: %s" %p_out)
r_out = session.reprotect(p_out, "FPE_FF1_AES256_ID_AN_LnRn_ASTNE",
        "FPE_FF1_AES256_ID_AN_LnRn_ASTNI")
print("Reprotected Data: %s" %r_out)
```

Result

```
Protected Data: NRejBkN7LcBOT4ÀÁÂÃÄÅÆÇÈÉ
Reprotected Data: AdbY0XkXIW7MvHÀÁÂÃÄÅÆÇÈÉ
```

2.4.3.12.4 Example - Reprotecting String Data Using FPE with External IV and External Tweak

This section describes how to use the *reprotect* API for reprotecting string data using FPE (FF1), with external IV and external tweak.

Warning: If you are using FPE with the *reprotect* API, then ensure that the plaintext alphabet type and the plaintext encoding used for FPE must be the same for both protecting and reprotecting the data. For example, if you have used FPE-Numeric data element with UTF-8 encoding to protect the data, then you must use only FPE-Numeric data element with UTF-8 encoding to reprotect the data.

Note: If you want to pass the external IV and external tweak as keyword arguments to the *reprotect* API, then you must pass the external IV and external tweak as bytes.

Example

In the following example, the *protegrity1234* string is used as the data, which is first protected using the FPE data element *FPE_FF1_AES256_ASCII_APIP_AN_L2R1_ASTNI_ML2*, with the help of external IV *1234* and external tweak *abcdef* that are both passed as bytes.

The protected input data, the *FPE_FF1_AES256_ASCII_APIP_AN_L2R1_ASTNI_ML2* data element, the old external IV *1234* and external tweak *abcdef* in bytes, and a new external IV *123456* and external tweak *xyz* in bytes are then passed as inputs to the *reprotect* API. As part of a single reprotect operation, the *reprotect* API first unprotects the protected input data using the given data element, and old external IV and external tweak, and then reprotects it using the same data element, but with the new external IV and external tweak.

```
from appython import Protector
protector = Protector()
session = protector.create_session("User1")
p_out = session.protect("protegrity1234",
        "FPE_FF1_AES256_ASCII_APIP_AN_L2R1_ASTNI_ML2", external_iv=bytes("1234",
encoding="utf-8"),
        external_tweak=bytes("abcdef", encoding="utf-8"))
print("Protected Data: %s" %p_out)
r_out = session.reprotect(p_out,
        "FPE_FF1_AES256_ASCII_APIP_AN_L2R1_ASTNI_ML2",
        "FPE_FF1_AES256_ASCII_APIP_AN_L2R1_ASTNI_ML2",
        old_external_iv=bytes("1234", encoding="utf-8"), new_external_iv=bytes("12345",
encoding="utf-8"),
        old_external_tweak=bytes("abcdef", encoding="utf-8"),
new_external_tweak=bytes("xyz", encoding="utf-8"))
print("Reprotected Data: %s" %r_out)
```

Result

```
Protected Data: prS6DaU5Dtd5g4
Reprotected Data: pr7hzGvIWOZQf4
```

2.4.3.12.5 Example - Retokenizing Bulk String Data

This section describes how to use the *reprotect* API for retokenizing bulk string data. You can pass bulk string data as a list or a tuple.

Caution: The individual elements of the list or tuple must be of the same data type.

Warning: If you are retokenizing the data using the *reprotect* API, then the old data element and the new data element must have the same tokenization type. For example, if you have used the Alpha-Numeric data element to protect the data, then you must use only the Alpha-Numeric data element to reprotect the data.

Example 1: Input bulk string data

In the following example, *protegrity1234*, *Protegrity1*, and *Protegrity56* strings are stored in a list and used as bulk data, which is tokenized using the *TE_A_N_S13_L1R3_N* data element.

The tokenized input data, the old data element *TE_A_N_S13_L1R3_N*, and a new data element *TE_A_N_S23_L2R2_Y* are then passed as inputs to the *reprotect* API. The *reprotect* API first detokenizes the protected input data using the old data element and then retokenizes it using the new data element, as part of a single reprotect operation.

```
from appython import Protector
protector = Protector()
session = protector.create_session("User1")
data = ["protegrity1234", "Protegrity1", "Protegrity56"]
p_out = session.protect(data, "TE_A_N_S13_L1R3_N")
print("Protected Data: ")
print(p_out)
r_out = session.reprotect(p_out[0], "TE_A_N_S13_L1R3_N",
```

```
        "TE_A_N_S23_L2R2_Y")
print("Reprotected Data: ")
print(r_out)
```

Result

```
Protected Data:
(['pLAvXYIAbp5234', 'P8PCmC8gty1', 'PHNjXrw7Iy56'], (6, 6, 6))
Reprotected Data:
(['prMLJsM8fZUp34', 'Pr9zdglWRy1', 'Pra9Ez5LPG56'], (6, 6, 6))
```

6 is the success return code for the protect operation of each element in the list.

Example 2: Input dates passed as bulk strings

In the following example, the *14/02/2019* and *11/03/2018* strings are stored in a list and used as bulk data, which is tokenized using the *TE_Date_DMY_S13* Date data element.

> **Caution:** If you have provided the date string as an input in a specific format, then you must use the data element with the same tokenization type as that of the input date format to protect the data. For example, if you have provided the input date string in DD/MM/YYYY format, then you must use only the Date (DD/MM/YYYY) data element to protect the data.

> **Note:** For information about the date formats supported by Protegrity tokenization methods, refer to the section *Protegrity Tokenization* in the *Protection Methods Reference Guide 9.1.0.0*.

The tokenized input data, the old data element *TE_Date_DMY_S13*, and a new data element *TE_Date_DMY_S16* are then passed as inputs to the *reprotect* API. The *reprotect* API first detokenizes the protected input data using the old data element and then retokenizes it using the new data element, as part of a single reprotect operation.

```
from appython import Protector
protector = Protector()
session = protector.create_session("User1")
data = ["14/02/2019", "11/03/2018"]
output = session.protect(data, "TE_Date_DMY_S13")
print("Protected data: "+str(output))
r_out = session.reprotect(output[0], "TE_Date_DMY_S13", "TE_Date_DMY_S16")
print("Reprotected data: "+str(r_out))
```

6 is the success return code for the protect operation of each element in the list.

Result

```
Protected data: (['08/07/2443', '17/08/1830'], (6, 6))
Reprotected data: (['19/10/1231', '25/09/2588'], (6, 6))
```

Example 3: Input date and time passed as bulk strings

In the following example, the *2019/02/14 10:54:47* and *2019/11/03 11:01:32* strings is used as the data, which is tokenized using the *TE_Datetime_TN_DN_M* Datetime data element.

> **Caution:** If you have provided the date and time string as an input in a specific format, then you must use the data element with the same tokenization type as that of the input date and time format to protect the data. For example, if you have provided the input date and time string in YYYY-MM-DD HH:MM:SS MMM format, then you must use only the Datetime (YYYY-MM-DD HH:MM:SS MMM) data element to protect the data.

> **Note:** For information about the date formats supported by Protegrity tokenization methods, refer to the section *Protegrity Tokenization* in the *Protection Methods Reference Guide 9.1.0.0*.

The tokenized input data, the old data element *TE_Datetime_TN_DN_M*, and a new data element *TE_Datetime_TN_DN_Y* are then passed as inputs to the *reprotect* API. The *reprotect* API first detokenizes the protected input data using the old data element and then retokenizes it using the new data element, as part of a single reprotect operation.

```
from appython import Protector
protector = Protector()
session = protector.create_session("User1")
data = ["2019/02/14 10:54:47", "2019/11/03 11:01:32"]
output = session.protect(data, "TE_Datetime_TN_DN_M")
print("Protected data: "+str(output))
r_out = session.reprotect(output[0], "TE_Datetime_TN_DN_M", "TE_Datetime_TN_DN_Y")
print("Reprotected data: "+str(r_out))
```

Result

```
Protected data: (['3311/02/22 10:54:47', '3311/11/02 11:01:32'], (6, 6))
Reprotected data: (['2019/09/25 10:54:47', '2019/05/16 11:01:32'], (6, 6))
```

6 is the success return code for the protect operation of each element in the list.

2.4.3.12.6 Example - Retokenizing Bulk String Data with External IV

This section describes how to use the *reprotect* API for retokenizing bulk string data using external IV. You can pass bulk string data as a list or a tuple.

Caution: The individual elements of the list or tuple must be of the same data type.

Warning: If you are retokenizing the data using the *reprotect* API, then the old data element and the new data element must have the same tokenization type. For example, if you have used the Alpha-Numeric data element to protect the data, then you must use only the Alpha-Numeric data element to reprotect the data.

Note: If you want to pass the external IV as a keyword argument to the *reprotect* API, then you must pass the external IV as bytes to the API.

Example

In the following example, *protegrity1234*, *Protegrity1*, and *Protegrity56* strings are stored in a list and used as bulk data, which is tokenized using the *TE_A_N_S23_L2R2_Y* data element, with the help of external IV *123* that is passed as bytes.

The tokenized input data, the *TE_A_N_S23_L2R2_Y* data element, the old external IV *1234* in bytes, and a new external IV *123456* in bytes are then passed as inputs to the *reprotect* API. As part of a single reprotect operation, the *reprotect* API first detokenizes the protected input data using the given data element and old external IV, and then retokenizes it using the same data element, but with the new external IV.

```
from appython import Protector
protector = Protector()
session = protector.create_session("User1")
data = ["protegrity1234", "Protegrity1", "Protegrity56"]
p_out = session.protect(data, "TE_A_N_S23_L2R2_Y",
        external_iv=bytes("1234", encoding="utf-8"))
print("Protected Data: ")
print(p_out)
r_out = session.reprotect(p_out[0], "TE_A_N_S23_L2R2_Y","TE_A_N_S23_L2R2_Y",
old_external_iv=bytes("1234", encoding="utf-8"),
new_external_iv=bytes("123456", encoding="utf-8"))
```

```
print("Reprotected Data: ")
print(r_out)
```

Result

```
Protected Data:
(['prbm147L5pc434', 'PrksvEshuy1', 'Prmx0hG8Nj56'], (6, 6, 6))
Reprotected Data:
(['prFApvQWkhC934', 'PrKxfmdTGy1', 'PrKciFj8Ng56'], (6, 6, 6))
```

6 is the success return code for the protect operation of each element in the list.

2.4.3.12.7 Example - Reprotecting Bulk String Data Using FPE

This section describes how to use the *reprotect* API for reprotecting bulk string data using FPE (FF1). You can pass bulk string data as a list or a tuple.

Caution: The individual elements of the list or tuple must be of the same data type.

Warning: If you are using FPE with the *reprotect* API, then ensure that the plaintext alphabet type and the plaintext encoding used for FPE must be the same for both protecting and reprotecting the data. For example, if you have used FPE-Numeric data element with UTF-8 encoding to protect the data, then you must use only FPE-Numeric data element with UTF-8 encoding to reprotect the data.

Example

In the following example, *protegrity1234ÀÁ*, *Protegrity1ÆÇÈ*, and *Protegrity56ÀÁÂÃÄÅ* strings are stored in a list and used as bulk data, which is protected using the FPE data element *FPE_FF1_AES256_ID_AN_LnRn_ASTNE*.

The tokenized input data, the old data element *FPE_FF1_AES256_ID_AN_LnRn_ASTNE*, and a new data element *FPE_FF1_AES256_ID_AN_LnRn_ASTNI* are then passed as inputs to the *reprotect* API. The *reprotect* API first unprotects the protected input data using the old data element and then reprotects it using the new data element, as part of a single reprotect operation.

```
from appython import Protector
protector = Protector()
session = protector.create_session("User1")
data = ["protegrity1234ÀÁ", "Protegrity1ÆÇÈ", "Protegrity56ÀÁÂÃÄÅ"]
p_out = session.protect(data, "FPE_FF1_AES256_ID_AN_LnRn_ASTNE")
print("Protected Data: ")
print(p_out)
r_out = session.reprotect(p_out[0], "FPE_FF1_AES256_ID_AN_LnRn_ASTNE",
        "FPE_FF1_AES256_ID_AN_LnRn_ASTNI")
print("Reprotected Data: ")
print(r_out)
```

Result

```
Protected Data:
([u'NRejBkN7LcBOT4\xc0\xc1', u'8BT1NNNqnPZ\xc6\xc7\xc8',
u'ecZslauY6iAl\xc0\xc1\xc2\xc3\xc4\xc5'], (6, 6, 6))
Reprotected Data:
([u'AdbY0XkXIW7MvH\xc0\xc1', u'1sw4XpkXXn2\xc6\xc7\xc8',
u'0dEqKSUy7OEX\xc0\xc1\xc2\xc3\xc4\xc5'], (6, 6, 6))
```

6 is the success return code for the protect operation of each element in the list.

2.4.3.12.8 Example - Reprotecting Bulk String Data Using FPE with External IV and External Tweak

This section explains how to use the *reprotect* API for reprotecting bulk string data using FPE (FF1), with external IV and external tweak. You can pass bulk string data as a list or a tuple.

Caution: The individual elements of the list or tuple must be of the same data type.

Warning: If you are using FPE with the *reprotect* API, then ensure that the plaintext alphabet type and the plaintext encoding used for FPE must be the same for both protecting and reprotecting the data. For example, if you have used FPE-Numeric data element with UTF-8 encoding to protect the data, then you must use only FPE-Numeric data element with UTF-8 encoding to reprotect the data.

Note: If you want to pass the external IV and external tweak as keyword arguments to the *reprotect* API, then you must pass the external IV and external tweak as bytes.

Example

In the following example, *protegrity1234ÀÁ*, *Protegrity1ÆÇÈ*, and *Protegrity56ÀÁÂÃÄÅ* strings are stored in a list and used as bulk data, which is first protected using the FPE data element *FPE_FF1_AES256_ASCII_APIP_AN_L2R1_ASTNI_ML2* data element, with the help of an external IV *1234* and external tweak *abc* that are both passed as bytes.

The protected input data, the *FPE_FF1_AES256_ASCII_APIP_AN_L2R1_ASTNI_ML2* data element, the old external IV *1234* and external tweak *abc* in bytes, and a new external IV *123456* and external tweak *xyz* in bytes are then passed as inputs to the *reprotect* API. As part of a single reprotect operation, the *reprotect* API first unprotects the protected input data using the given data element, and old external IV and external tweak, and then reprotects it using the same data element, but with the new external IV and external tweak.

```
from appython import Protector
protector = Protector()
session = protector.create_session("User1")
data = ["protegrity1234ÀÁ", "Protegrity1ÆÇÈ", "Protegrity56ÀÁÂÃÄÅ"]
p_out = session.protect(data,
        "FPE_FF1_AES256_ASCII_APIP_AN_L2R1_ASTNI_ML2", external_iv=bytes("1234",
encoding="utf-8"),
        external_tweak=bytes("abc", encoding="utf-8"))
print("Protected Data: ")
print(p_out)
r_out = session.reprotect(p_out[0],
        "FPE_FF1_AES256_ASCII_APIP_AN_L2R1_ASTNI_ML2",
        "FPE_FF1_AES256_ASCII_APIP_AN_L2R1_ASTNI_ML2",
        old_external_iv=bytes("1234", encoding="utf-8"),
new_external_iv=bytes("123456", encoding="utf-8"),
        old_external_tweak=bytes("abc", encoding="utf-8"),
new_external_tweak=bytes("xyz", encoding="utf-8"))
print("Reprotected Data: ")
print(r_out)
```

Result

```
Protected Data:
([u'prngoI74u6NZrY\xc0\xc1', u'PrFBtLOLDBJ\xc6\xc7\xc8',
u'PrlIizsBZ8Bc\xc0\xc1\xc2\xc3\xc4\xc5'], (6, 6, 6))
Reprotected Data:
([u'prvKwWyJiHTjtV\xc0\xc1', u'PrOjAc1YuIp\xc6\xc7\xc8',
u'PrgiU5fdHGXE\xc0\xc1\xc2\xc3\xc4\xc5'], (6, 6, 6))
```

6 is the success return code for the protect operation of each element in the list.

2.4.3.12.9 Example - Retokenizing Integer Data

This section explains how to use the *reprotect* API for retokenizing integer data.

Warning: If you are retokenizing the data using the *reprotect* API, then the old data element and the new data element must have the same tokenization type. For example, if you have used Integer data element to protect the data, then you must use only Integer data element to reprotect the data.

Example

In the following example, *21* is used as the input integer data, which is first tokenized using the *TE_INT_4* data element.

The tokenized input data, the old data element *TE_INT_4*, and a new data element *TE_INT_4_1* are then passed as inputs to the *reprotect* API. The *reprotect* API first detokenizes the protected input data using the old data element and then retokenizes it using the new data element, as part of a single reprotect operation.

```
from appython import Protector
protector = Protector()
session = protector.create_session("User1")
output = session.protect(21, "TE_INT_4")
print("Protected Data: %s" %output)
r_out = session.reprotect(output, "TE_INT_4", "TE_INT_4_1")
print("Reprotected Data: %s" %r_out)
```

Result

```
Protected Data: -1926573911
Reprotected Data: 1673602066
```

2.4.3.12.10 Example - Retokenizing Integer Data with External IV

This section describes how to use the *reprotect* API for retokenizing integer data using external IV.

Warning: If you are retokenizing the data using the *reprotect* API, then the old data element and the new data element must have the same tokenization type. For example, if you have used the Integer data element to protect the data, then you must use only the Integer data element to reprotect the data.

Note: If you want to pass the external IV as a keyword argument to the *reprotect* API, then you must pass the external IV as bytes to the API.

Note:

The AP Python APIs support integer values only between -2147483648 and 2147483648, both inclusive.

Example

In the following example, *21* is used as the input integer data, which is first tokenized using the *TE_INT_4* data element, with the help of external IV *1234* that is passed as bytes.

The tokenized input data, the *TE_INT_4* data element, the old external IV *1234* in bytes, and a new external IV *123456* in bytes are then passed as inputs to the *reprotect* API. As part of a single reprotect operation, the *reprotect* API first detokenizes the protected input data using the given data element and old external IV, and then retokenizes it using the same data element, but with the new external IV.

```
from appython import Protector
protector = Protector()
```

```
session = protector.create_session("User1")
p_out = session.protect(21, "TE_INT_4",
        external_iv=bytes("1234", encoding="utf-8"))
print("Protected Data: %s" %p_out)
r_out = session.reprotect(p_out, "TE_INT_4", "TE_INT_4",
        old_external_iv=bytes("1234", encoding="utf-8"),
new_external_iv=bytes("123456", encoding="utf-8"))
print("Reprotected Data: %s" %r_out)
```

Result

```
Protected Data: -2122057622
Reprotected Data: 342830163
```

2.4.3.12.11 Example - Retokenizing Bulk Integer Data

This section describes how to use the *reprotect* API for retokenizing bulk integer data. You can pass bulk integer data as a list or a tuple.

Caution: The individual elements of the list or tuple must be of the same data type.

Warning: If you are retokenizing the data using the *reprotect* API, then the old data element and the new data element must have the same tokenization type. For example, if you have used the Integer data element to protect the data, then you must use only the Integer data element to reprotect the data.

Example

In the following example, *21*, *42*, and *55* integers are stored in a list and used as bulk data, which is tokenized using the *TE_INT_4* data element.

The tokenized input data, the old data element *TE_INT_4*, and a new data element *TE_INT_4_1* are then passed as inputs to the *reprotect* API. The *reprotect* API first detokenizes the protected input data using the old data element and then retokenizes it using the new data element, as part of a single reprotect operation.

```
from appython import Protector
protector = Protector()
session = protector.create_session("User1")
data = [21, 42, 55]
p_out = session.protect(data, "TE_INT_4")
print("Protected Data: ")
print(p_out)
r_out = session.reprotect(p_out[0], "TE_INT_4", "TE_INT_4_1")
print("Reprotected Data: ")
print(r_out)
```

Result

```
Protected Data:
([-1926573911, -1970496120, -814489753], (6, 6, 6))
Reprotected Data:
([1673602066, -2106523868, 1683756976], (6, 6, 6))
```

6 is the success return code for the protect operation of each element in the list.

2.4.3.12.12 Example - Retokenizing Bulk Integer Data with External IV

This section describes how to use the *reprotect* API for retokenizing bulk integer data using external IV. You can pass bulk integer data as a list or a tuple.

Caution: The individual elements of the list or tuple must be of the same data type.

Warning: If you are retokenizing the data using the *reprotect* API, then the old data element and the new data element must have the same tokenization type. For example, if you have used the Integer data element to protect the data, then you must use only the Integer data element to reprotect the data.

Note: If you want to pass the external IV as a keyword argument to the *reprotect* API, then you must pass the external IV as bytes to the API.

Example

In the following example, *21*, *42*, and *55* integers are stored in a list and used as bulk data, which is tokenized using the *TE_INT_4* data element, with the help of external IV *1234* that is passed as bytes.

The tokenized input data, the *TE_INT_4* data element, the old external IV *1234* in bytes, and a new external IV *123456* in bytes are then passed as inputs to the *reprotect* API. As part of a single reprotect operation, the *reprotect* API first detokenizes the protected input data using the given data element and old external IV, and then retokenizes it using the same data element, but with the new external IV.

```
from appython import Protector
protector = Protector()
session = protector.create_session("User1")
data = [21, 42, 55]
p_out = session.protect(data, "TE_INT_4", external_iv=bytes("1234", encoding="utf-8"))
print("Protected Data: ")
print(p_out)
r_out = session.reprotect(p_out[0], "TE_INT_4", "TE_INT_4",
        old_external_iv=bytes("1234", encoding="utf-8"),
new_external_iv=bytes("123456", encoding="utf-8"))
print("Reprotected Data: ")
print(r_out)
```

Result

```
Protected Data:
([-2122057622, 1795905968, 228587043], (6, 6, 6))
Reprotected Data:
([342830163, 1360764745, -1892139659], (6, 6, 6))
```

6 is the success return code for the protect operation of each element in the list.

2.4.3.12.13 Example - Retokenizing Long Data

This section describes how to use the *reprotect* API for retokenizing long data.

Warning: If you are retokenizing the data using the *reprotect* API, then the old data element and the new data element must have the same tokenization type. For example, if you have used the Integer data element to protect the data, then you must use only the Integer data element to reprotect the data.

Example

In the following example, *1376235139103947* is used as the input long data, which is first tokenized using the *TE_INT_8* data element.

The tokenized input data, the old data element *TE_INT_8*, and a new data element *TE_INT_8_1* are then passed as inputs to the *reprotect* API. The *reprotect* API first detokenizes the protected input data using the old data element and then retokenizes it using the new data element, as part of a single reprotect operation.

```
from appython import Protector
protector = Protector()
session = protector.create_session("User1")
output = session.protect(1376235139103947, "TE_INT_8")
print("Protected Data: %s" %output)
r_out = session.reprotect(output, "TE_INT_8", "TE_INT_8_1")
print("Reprotected Data: %s" %r_out)
```

Result

```
Protected Data: -1770169866845757900
Reprotected Data: 1496033169477057599
```

2.4.3.12.14 Example - Retokenizing Long Data with External IV

This section describes how to use the *reprotect* API for retokenizing long data using external IV.

Warning: If you are retokenizing the data using the *reprotect* API, then the old data element and the new data element must have the same tokenization type. For example, if you have used the Integer data element to protect the data, then you must use only the Integer data element to reprotect the data.

Note: If you want to pass the external IV as a keyword argument to the *reprotect* API, then you must pass the external IV as bytes to the API.

Example

In the following example, *1376235139103947* is used as the input long data, which is first tokenized using the *TE_INT_8* data element, with the help of external IV *1234* that is passed as bytes.

The tokenized input data, the *TE_INT_8* data element, the old external IV *1234* in bytes, and a new external IV *123456* in bytes are then passed as inputs to the *reprotect* API. As part of a single reprotect operation, the *reprotect* API first detokenizes the protected input data using the given data element and old external IV, and then retokenizes it using the same data element, but with the new external IV.

```
from appython import Protector
protector = Protector()
session = protector.create_session("User1")
p_out = session.protect(1376235139103947, "TE_INT_8",
        external_iv=bytes("1234", encoding="utf-8"))
print("Protected Data: %s" %p_out)
r_out = session.reprotect(p_out, "TE_INT_8", "TE_INT_8",
        old_external_iv=bytes("1234", encoding="utf-8"),
new_external_iv=bytes("123456", encoding="utf-8"))
print("Reprotected Data: %s" %r_out)
```

Result

```
Protected Data: 5846214101577367207
Reprotected Data: 2547273918835895593
```

2.4.3.12.15 Example - Retokenizing Bulk Long Data

This section describes how to use the *reprotect* API for retokenizing bulk long data. You can pass bulk long data as a list or a tuple.

Caution: The individual elements of the list or tuple must be of the same data type.

Warning: If you are retokenizing the data using the *reprotect* API, then the old data element and the new data element must have the same tokenization type. For example, if you have used the Integer data element to protect the data, then you must use only the Integer data element to reprotect the data.

Example

In the following example, *1376235139103947*, *2396235839173981*, and *9371234126176985* long data are stored in a list and used as bulk data, which is tokenized using the *TE_INT_8* data element.

The tokenized input data, the old data element *TE_INT_8*, and a new data element *TE_INT_8_1* are then passed as inputs to the *reprotect* API. The *reprotect* API first detokenizes the protected input data using the old data element and then retokenizes it using the new data element, as part of a single reprotect operation.

```
from appython import Protector
protector = Protector()
session = protector.create_session("User1")
data = [1376235139103947, 2396235839173981, 9371234126176985]
p_out = session.protect(data, "TE_INT_8")
print("Protected Data: ")
print(p_out)
r_out = session.reprotect(p_out[0], "TE_INT_8", "TE_INT_8_1")
print("Reprotected Data: ")
print(r_out)
```

Result

```
Protected Data:
([-1770169866845757900L, -8142006510957348982L, -20687656704969669L], (6, 6, 6))
Reprotected Data:
([1496033169477057599L, -751706970736718821L, 648488512692712284L], (6, 6, 6))
```

6 is the success return code for the protect operation of each element in the list.

2.4.3.12.16 Example - Retokenizing Bulk Long Data with External IV

This section describes how to use the *reprotect* API for retokenizing bulk long data using external IV. You can pass bulk long data as a list or a tuple.

Caution: The individual elements of the list or tuple must be of the same data type.

Warning: If you are retokenizing the data using the *reprotect* API, then the old data element and the new data element must have the same tokenization type. For example, if you have used the Integer data element to protect the data, then you must use only the Integer data element to reprotect the data.

Note: If you want to pass the external IV as a keyword argument to the *reprotect* API, then you must pass the external IV as bytes to the API.

Example

In the following example, *1376235139103947*, *2396235839173981*, and *9371234126176985* long data are stored in a list and used as bulk data, which is tokenized using the *TE_INT_8* data element, with the help of external IV *1234* that is passed as bytes.

The tokenized input data, the *TE_INT_8* data element, the old external IV *1234* in bytes, and a new external IV *123456* in bytes are then passed as inputs to the *reprotect* API. As part of a single reprotect operation, the *reprotect* API first detokenizes the protected input data using the given data element and old external IV, and then retokenizes it using the same data element, but with the new external IV.

```
from appython import Protector
protector = Protector()
session = protector.create_session("User1")
data = [1376235139103947, 2396235839173981, 9371234126176985]
p_out = session.protect(data, "TE_INT_8", external_iv=bytes("1234", encoding="utf-8"))
print("Protected Data: ")
print(p_out)
r_out = session.reprotect(p_out[0], "TE_INT_8", "TE_INT_8",
        old_external_iv=bytes("1234", encoding="utf-8"),
new_external_iv=bytes("123456", encoding="utf-8"))
print("Reprotected Data: ")
print(r_out)
```

Result

```
Protected Data:
([5846214101577367207L, 5661139619224336475L, 7806173497368534531L], (6, 6, 6))
Reprotected Data:
([2547273918835895593L, 3484073575451507396L, 1789344813959912458L], (6, 6, 6))
```

6 is the success return code for the protect operation of each element in the list.

2.4.3.12.17 Example - Reprotecting Float Data

This section describes how to use the *reprotect* API for reprotecting float data using a No-Encryption data element. You can use this API for access control and auditing.

Warning: If you are reprotecting the data using the *reprotect* API, then the old data element and the new data element must have the same tokenization type. For example, if you have used the No Encryption data element to protect the data, then you must use only the No Encryption data element to reprotect the data.

Example

In the following example, *22.5* is used as the input float data, which is first protected using the *NoEncryption_1* data element.

The protected input data, the old data element *NoEncryption_1*, and a new data element *NoEncryption_2* are then passed as inputs to the *reprotect* API. The *reprotect* API first unprotects the protected input data using the old data element and then reprotects it using the new data element, as part of a single reprotect operation.

```
from appython import Protector
protector = Protector()
session = protector.create_session("User1")
output = session.protect(22.5, "NoEncryption_1")
print("Protected Data: %s" %output)
r_out = session.reprotect(output, "NoEncryption_1", "NoEncryption_2")
print("Reprotected Data: %s" %r_out)
```

Result

```
Protected Data: 22.5
Reprotected Data: 22.5
```

As we are using a No-Encryption data element to protect and reprotect the data, the reprotected output data is the same as the protected data.

2.4.3.12.18 Example - Reprotecting Bulk Float Data

This section describes how to use the *reprotect* API for reprotecting bulk float data using a No-Encryption data element. You can pass bulk float data as a list or a tuple. You can use this API for access control and auditing.

Caution: The individual elements of the list or tuple must be of the same data type.

Warning: If you are reprotecting the data using the *reprotect* API, then the old data element and the new data element must have the same tokenization type. For example, if you have used the No Encryption data element to protect the data, then you must use only the No Encryption data element to reprotect the data.

Example

In the following example, *22.5*, *48.93*, and *94.14* float data are stored in a list and used as bulk data, which is tokenized using the *NoEncryption_1* data element.

The tokenized input data, the old data element *NoEncryption_1*, and a new data element *NoEncryption_2* are then passed as inputs to the *reprotect* API. The *reprotect* API first unprotects the protected input data using the old data element and then reprotects it using the new data element, as part of a single reprotect operation.

```
from appython import Protector
protector = Protector()
session = protector.create_session("User1")
data = [22.5, 48.93, 94.31]
p_out = session.protect(data, "NoEncryption_1")
print("Protected Data: ")
print(p_out)
r_out = session.reprotect(p_out[0], "NoEncryption_1", "NoEncryption_2")
print("Reprotected Data: ")
print(r_out)
```

Result

```
Protected Data:
([22.5, 48.93, 94.31], (6, 6, 6))
Reprotected Data:
([22.5, 48.93, 94.31], (6, 6, 6))
```

6 is the success return code for the protect operation of each element in the list.

As we are using a No Encryption data element to protect and reprotect the data, the reprotected output data is the same as the protected data.

2.4.3.12.19 Example - Retokenizing Bytes Data

This section describes how to use the *reprotect* API for retokenizing bytes data.

Warning: If you are retokenizing the data using the *reprotect* API, then the old data element and the new data element must have the same tokenization type. For example, if you have used the Alpha-Numeric data element to protect the data, then you must use only the Alpha-Numeric data element to reprotect the data.

Example

In the following example, *Protegrity1* string is first converted to bytes using the Python *bytes()* method. The bytes data is then tokenized using the *TE_A_N_S23_L2R2_Y* data element.

The tokenized input data, the old data element *TE_A_N_S23_L2R2_Y*, and a new data element *TE_A_N_S13_L1R3_N* are then passed as inputs to the *reprotect* API. The *reprotect* API first detokenizes the protected input data using the old data element and then retokenizes it using the new data element, as part of a single reprotect operation.

```
from appython import Protector
protector = Protector()
session = protector.create_session("User1")
data=bytes("Protegrity1", encoding="utf-8")
p_out = session.protect(data, "TE_A_N_S23_L2R2_Y")
print("Protected Data: %s" %p_out)
r_out = session.reprotect(p_out, "TE_A_N_S23_L2R2_Y",
        "TE_A_N_S13_L1R3_N")
print("Reprotected Data: %s" %r_out)
```

Result

```
Protected Data: b'Pr9zdglWRy1'
Reprotected Data: b'P8PCmC8gty1'
```

2.4.3.12.20 Example - Retokenizing Bytes Data with External IV

This section describes how to use the *reprotect* API for retokenizing bytes data using external IV.

Warning: If you are retokenizing the data using the *reprotect* API, then the old data element and the new data element must have the same tokenization type. For example, if you have used the Alpha-Numeric data element to protect the data, then you must use only the Alpha-Numeric data element to reprotect the data.

Example

In the following example, *Protegrity1* string is first converted to bytes using the Python *bytes()* method. The bytes data is then tokenized using the *TE_A_N_S23_L2R2_Y* data element, with the help of external IV *1234* that is passed as bytes.

The tokenized input data, the *TE_A_N_S23_L2R2_Y* data element, the old external IV *1234* in bytes, and a new external IV *123456* in bytes are then passed as inputs to the *reprotect* API. As part of a single reprotect operation, the *reprotect* API first detokenizes the protected input data using the given data element and old external IV, and then retokenizes it using the same data element, but with the new external IV.

```
from appython import Protector
protector = Protector()
session = protector.create_session("User1")
data=bytes("Protegrity1", encoding="utf-8")
p_out = session.protect(data, "TE_A_N_S23_L2R2_Y",
        external_iv=bytes("1234", encoding="utf-8"))
print("Protected Data: %s" %p_out)
r_out = session.reprotect(p_out, "TE_A_N_S23_L2R2_Y",
        "TE_A_N_S23_L2R2_Y", old_external_iv=bytes("1234", encoding="utf-8"),
        new_external_iv=bytes("123456", encoding="utf-8"))
print("Reprotected Data: %s" %r_out)
```

Result

```
Protected Data: b'PrksvEshuy1'
Reprotected Data: b'PrKxfmdTGy1'
```

2.4.3.12.21 Example - Re-Encrypting Bytes Data

This section describes how to use the *reprotect* API for re-encrypting bytes data.

Warning: If you are using the *reprotect* API, then the old data element and the new data element must be of the same protection method. For example, if you have used the AES256 data element to protect the data, then you must use only the AES256 data element to reprotect the data.

Example

In the following example, *Protegrity1* string is first converted to bytes using the Python *bytes()* method. The bytes data is then encrypted using the *AES256* data element. Therefore, the *encrypt_to* parameter is passed as a keyword argument, and its value is set to *bytes*.

The encrypted input data, the old data element *AES256*, and a new data element *AES256_IV_CRC_KID* are then passed as inputs to the *reprotect* API. The *reprotect* API first decrypts the protected input data using the old data element and then re-encrypts it using the new data element, as part of a single reprotect operation. Therefore, the *encrypt_to* parameter is passed as a keyword argument, and its value is set to *bytes*.

```
from appython import Protector
protector = Protector()
session = protector.create_session("User1")
data=bytes("Protegrity1", encoding="utf-8")
p_out = session.protect(data, "AES256", encrypt_to = bytes)
print("Encrypted Data: %s" %p_out)
r_out = session.reprotect(p_out, "AES256", "AES256_IV_CRC_KID", encrypt_to = bytes)
print("Re-encrypted Data: %s" %r_out)
```

Result

```
Encrypted Data: b't▒4Lq▒'
Re-encrypted Data: b' ,f7d▒:sD&w]Vdy-'
```

2.4.3.12.22 Example - Retokenizing Bulk Bytes Data

This section describes how to use the *reprotect* API for retokenizing bulk bytes data. You can pass bulk bytes data as a list or a tuple.

Caution: The individual elements of the list or tuple must be of the same data type.

Warning: If you are retokenizing the data using the *reprotect* API, then the old data element and the new data element must have the same tokenization type. For example, if you have used the Alpha-Numeric data element to protect the data, then you must use only the Alpha-Numeric data element to reprotect the data.

Example

In the following example, *protegrity1234*, *Protegrity1*, and *Protegrity56* strings are first converted to bytes using the Python *bytes()* method. The converted bytes are then stored in a list and used as bulk data, which is tokenized using the *TE_A_N_S23_L2R2_Y* data element.

The tokenized input data, the old data element *TE_A_N_S23_L2R2_Y*, and a new data element *TE_A_N_S13_L1R3_N* are then passed as inputs to the *reprotect* API. The *reprotect* API first detokenizes the protected input data using the old data element and then retokenizes it using the new data element, as part of a single reprotect operation.

```
from appython import Protector
protector = Protector()
session = protector.create_session("User1")
data = [bytes("protegrity1234"), bytes("Protegrity1"), bytes("Protegrity56")]
```

```
p_out = session.protect(data, "TE_A_N_S23_L2R2_Y")
print("Protected Data: ")
print(p_out)
r_out = session.reprotect(p_out[0], "TE_A_N_S23_L2R2_Y",
        "TE_A_N_S13_L1R3_N")
print("Reprotected Data: ")
print(r_out)
```

Result

```
Protected Data:
([b'prMLJsM8fZUp34', b'Pr9zdglWRy1', b'Pra9Ez5LPG56'], (6, 6, 6))
Reprotected Data:
([b'pLAvXYIAbp5234', b'P8PCmC8gty1', b'PHNjXrw7Iy56'], (6, 6, 6))
```

6 is the success return code for the protect operation of each element in the list.

2.4.3.12.23 Example - Retokenizing Bulk Bytes Data with External IV

This section describes how to use the *reprotect* API for retokenizing bulk bytes data using external IV. You can pass bulk bytes data as a list or a tuple.

Caution: The individual elements of the list or tuple must be of the same data type.

Warning: If you are retokenizing the data using the *reprotect* API, then the old data element and the new data element must have the same tokenization type. For example, if you have used the Alpha-Numeric data element to protect the data, then you must use only the Alpha-Numeric data element to reprotect the data.

Example

In the following example, *protegrity1234*, *Protegrity1*, and *Protegrity56* strings are first converted to bytes using the Python *bytes()* method. The converted bytes are then stored in a list and used as bulk data, which is tokenized using the *TE_A_N_S23_L2R2_Y* data element, with the help of external IV *1234* that is passed as bytes.

The tokenized input data, the *TE_A_N_S23_L2R2_Y* data element, the old external IV *1234* in bytes, and a new external IV *123456* in bytes are then passed as inputs to the *reprotect* API. As part of a single reprotect operation, the *reprotect* API first detokenizes the protected input data using the given data element and old external IV, and then retokenizes it using the same data element, but with the new external IV.

```
from appython import Protector
protector = Protector()
session = protector.create_session("User1")
data = [bytes("protegrity1234", encoding="utf-8"), bytes("Protegrity1",
encoding="utf-8"), bytes("Protegrity56", encoding="utf-8")]
p_out = session.protect(data, "TE_A_N_S23_L2R2_Y",
        external_iv=bytes("1234", encoding="utf-8"))
print("Protected Data: ")
print(p_out)
r_out = session.reprotect(p_out[0], "TE_A_N_S23_L2R2_Y",
        "TE_A_N_S23_L2R2_Y", old_external_iv=bytes("1234", encoding="utf-8"),
        new_external_iv=bytes("123456", encoding="utf-8"))
print("Reprotected Data: ")
print(r_out)
```

Result

```
Protected Data:
([b'prbm147L5pc434', b'PrksvEshuy1', b'Prmx0hG8Nj56'], (6, 6, 6))
Reprotected Data:
([b'prFApvQWkhC934', b'PrKxfmdTGy1', b'PrKciFj8Ng56'], (6, 6, 6))
```

6 is the success return code for the protect operation of each element in the list.

2.4.3.12.24 Example - Re-Encrypting Bulk Bytes Data

This section describes how to use the *reprotect* API for re-encrypting bulk bytes data. You can pass bulk bytes data as a list or a tuple.

Caution: The individual elements of the list or tuple must be of the same data type.

Warning: If you are using the *reprotect* API, then the old data element and the new data element must be of the same protection method. For example, if you have used the AES256 data element to protect the data, then you must use only the AES256 data element to reprotect the data.

Example

In the following example, *protegrity1234*, *Protegrity1*, and *Protegrity56* strings are first converted to bytes using the Python *bytes()* method. The converted bytes are then stored in a list and used as bulk data, which is encrypted using the *AES256* data element. Therefore, the *encrypt_to* parameter is passed as a keyword argument, and its value is set to *bytes*.

The encrypted input data, the old data element *AES256*, and a new data element *AES256_IV_CRC_KID* are then passed as inputs to the *reprotect* API. The *reprotect* API first decrypts the protected input data using the old data element and then re-encrypts it using the new data element, as part of a single reprotect operation. Therefore, the *encrypt_to* parameter is passed as a keyword argument, and its value is set to *bytes*.

```
from appython import Protector
protector = Protector()
session = protector.create_session("User1")
data = [bytes("protegrity1234", encoding ="UTF-8"), bytes("Protegrity1", encoding
="UTF-8"), bytes("Protegrity56", encoding ="UTF-8")]
p_out = session.protect(data, "AES256", encrypt_to = bytes)
print("Encrypted Data: ")
print(p_out)
r_out = session.reprotect(p_out[0], "AES256", "AES256_IV_CRC_KID", encrypt_to = bytes)
print("Re-encrypted Data: ")
print(r_out)
```

Result

```
Encrypted Data:
([b'\xc9^x\x02)\xcbB\x91}\x7fi\x8a\xce\x8d>H',
b't\x80\xf5\x8d\x9e\x0b+4Lq\x8a\x97\xdb\x8fx\x16',
b'\x87\x08\x938\xf7o~\xab\xa3\xc2L\xa90>\x18_'], (6, 6, 6))
Re-encrypted Data:
([b' \x08\xdfV2)A/
\xc2\x96X\x86M\xbf&$P\xa1\xb9\x83o\xb4\x90\x9b\x8d\xf8\xf5\x976\x95\xcd\xf4\xea\xc7\xad\
xedl\xbck\xd1\xf3@\xf7.\xfd\xe0\x13H\xe6\xb1', b'
\x08\x11\x7f\xdf\x05\xf0I\xaa\xd1\xe2v`\xe9\x9dH\xa1\xa3\x025oW~\xc7\xf0KT\xd4\x1c\x05V\
xaei\xee', b' \x08)\x84N&\xd4e(lq\xfa\x8d\x05\xa9\xe5\x8do(\xf2T\xe3l\xa9|
V\xc2&X\x1d\x02yF[\xbfb(x\xe3\x1a/|\x91K\xc2\xc8\xf2"\x89\xc3'], (6, 6, 6))
```

Warning:

To avoid data corruption, do not convert the encrypted bytes data into string format. It is recommended that you to convert the encrypted bytes data to a Hexadecimal, Base 64, or any other appropriate format.

2.4.3.12.25 Example - Retokenizing Date Objects

This section describes how to use the *reprotect* API for retokenizing date objects.

Warning: If you are retokenizing the data using the *reprotect* API, then the old data element and the new data element must have the same tokenization type. For example, if you have used the Date (DD/MM/YYYY) data element to protect the data, then you must use only the Date (DD/MM/YYYY) data element to reprotect the data.

Note: For information about the date formats supported by Protegrity tokenization methods, refer to the section *Protegrity Tokenization* in the *Protection Methods Reference Guide 9.1.0.0*.

Example: Input as a data object

In the following example, the *12/02/2019* date string is used as the data, which is first converted to a date object using the Python *date* method of the *datetime* module. The date object is then tokenized using the *TE_Date_DMY_S13* data element.

The tokenized input data, the old data element *TE_Date_DMY_S13*, and a new data element *TE_Date_DMY_S16* are then passed as inputs to the *reprotect* API. The *reprotect* API first detokenizes the protected input data using the old data element and then retokenizes it using the new data element, as part of a single reprotect operation.

```
from appython import Protector
from datetime import datetime
protector = Protector()
session = protector.create_session("User1")
data = datetime.strptime("12/02/2019", "%d/%m/%Y").date()
print("Input date as a Date object : "+str(data))
p_out = session.protect(data, "TE_Date_DMY_S13")
print("Protected date: "+str(p_out))
r_out = session.reprotect(p_out, "TE_Date_DMY_S13", "TE_Date_DMY_S16")
print("Reprotected date: "+str(r_out))
```

Result

```
Input date as a Date object : 2019-02-12
Protected date: 1896-10-21
Reprotected date: 2130-06-19
```

2.4.3.12.26 Example - Retokenizing Bulk Date Objects

This section describes how to use the *reprotect* API for retokenizing bulk date objects. You can pass bulk date objects as a list or a tuple.

Caution: The individual elements of the list or tuple must be of the same data type.

Warning: If you are retokenizing the data using the *reprotect* API, then the old data element and the new data element must have the same tokenization type. For example, if you have used the Date (DD/MM/YYYY) data element to protect the data, then you must use only the Date (DD/MM/YYYY) data element to reprotect the data.

Note: For information about the date formats supported by Protegrity tokenization methods, refer to the section *Protegrity Tokenization* in the *Protection Methods Reference Guide 9.2.0.0*.

Example: Input as a Date Object

In the following example, the *12/02/2019* and *11/01/2018* date strings are used as the data, which are first converted to a date objects using the Python *date* method of the *datetime* module. The two date objects are then used to create a list, which is used as the input data.

The input list is then tokenized using the *TE_Date_DMY_S13* data element.

The tokenized input data, the old data element *TE_Date_DMY_S13*, and a new data element *TE_Date_DMY_S16* are then passed as inputs to the *reprotect* API. The *reprotect* API first detokenizes the protected input data using the old data element and then retokenizes it using the new data element, as part of a single reprotect operation.

```
from appython import Protector
from datetime import datetime
protector = Protector()
session = protector.create_session("User1")
data1 = datetime.strptime("12/02/2019", "%d/%m/%Y").date()
data2 = datetime.strptime("11/01/2018", "%d/%m/%Y").date()
data = [data1, data2]
print("Input data: ", str(data))
p_out = session.protect(data, "TE_Date_DMY_S13")
print("Protected data: "+str(p_out))
r_out = session.reprotect(p_out[0], "TE_Date_DMY_S13", "TE_Date_DMY_S16")
print("Reprotected date: "+str(r_out))
```

Result

```
Input data:  [datetime.date(2019, 2, 12), datetime.date(2018, 1, 11)]
Protected data: ([datetime.date(1896, 10, 21), datetime.date(696, 3, 1)], (6, 6))
Reprotected date: ([datetime.date(2130, 6, 19), datetime.date(1339, 10, 10)], (6, 6))
```

6 is the success return code for the protect operation of each element in the list.

2.4.3.12.27 Example - Retokenizing Unicode Data

This section describes how to use the *reprotect* API for retokenizing unicode data.

Warning: If you are retokenizing the data using the *reprotect* API, then the old data element and the new data element must have the same tokenization type. For example, if you have used the Alpha-Numeric data element to protect the data, then you must use only the Alpha-Numeric data element to reprotect the data.

Example

In the following example, the *u'protegrity1234ÀÁÂÃÄÅÆÇÈÉ'* unicode data is used as the input data, which is first tokenized using the *TE_A_N_S23_L2R2_Y* data element.

The tokenized input data, the old data element *TE_A_N_S23_L2R2_Y*, and a new data element *TE_AN_S23_L0R0_Y* are then passed as inputs to the *reprotect* API. The *reprotect* API first detokenizes the protected input data using the old data element and then retokenizes it using the new data element, as part of a single reprotect operation.

```
from appython import Protector
protector = Protector()
session = protector.create_session("User1")
output = session.protect(u'protegrity1234ÀÁÂÃÄÅÆÇÈÉ', "TE_A_N_S23_L2R2_Y")
print("Protected Data: %s" %output)
r_out = session.reprotect(output, "TE_A_N_S23_L2R2_Y",
        "TE_AN_S23_L0R0_Y")
print("Reprotected Data: %s" %r_out)
```

Result

```
Protected Data: prZeslalwuQQy3ÀÁÂÃÄÅÆÇÈÉ
Reprotected Data: Nw8MLVwbdcBMUaÀÁÂÃÄÅÆÇÈÉ
```

2.4.3.12.28 Example - Retokenizing Bulk Unicode Data

This section describes how to use the *reprotect* API for retokenizing bulk unicode data. You can pass bulk string data as a list or a tuple.

Caution: The individual elements of the list or tuple must be of the same data type.

Warning: If you are retokenizing the data using the *reprotect* API, then the old data element and the new data element must have the same tokenization type. For example, if you have used the Alpha-Numeric data element to protect the data, then you must use only the Alpha-Numeric data element to reprotect the data.

Example

In the following example, *u'protegrity1234ÀÁÂÃÄÅÆÇÈÉ'*, *u'Protegrity1ÆÇÈÉÀÁÂÃÄÅ'*, and *u'Protegrity56ÇÅÆÈÉÂÃ'* unicode data are stored in a list and used as bulk data, which is tokenized using the *TE_A_N_S23_L2R2_Y* data element.

The tokenized input data, the old data element *TE_A_N_S13_L1R3_N*, and a new data element *TE_A_N_S23_L2R2_Y* are then passed as inputs to the *reprotect* API. The *reprotect* API first detokenizes the protected input data using the old data element and then retokenizes it using the new data element, as part of a single reprotect operation.

```
from appython import Protector
protector = Protector()
session = protector.create_session("User1")
data = [u'protegrity1234ÀÁÂÃÄÅÆÇÈÉ', u'Protegrity1ÆÇÈÉÀÁÂÃÄÅ', u'Protegrity56ÇÅÆÈÉÂÃ']
p_out = session.protect(data, "TE_A_N_S13_L1R3_N")
print("Protected Data: ")
print(p_out)
r_out = session.reprotect(p_out[0], "TE_A_N_S13_L1R3_N",
        "TE_A_N_S23_L2R2_Y")
print("Reprotected Data: ")
print(r_out)
```

Result

```
Protected Data:
([u'p3oZN1j1PF33hz\xc0\xc1\xc2\xc3\xc4\xc5\xc6\xc7\xc8\xc9',
u'P5fjL8vdBci\xc6\xc7\xc8\xc9\xc0\xc1\xc2\xc3\xc4\xc5',
u'PIo45D7g73Sm\xc7\xc5\xc6\xc8\xc9\xc2\xc3'], (6, 6, 6))
Reprotected Data:
([u'prZeslalwuQQy3\xc0\xc1\xc2\xc3\xc4\xc5\xc6\xc7\xc8\xc9',
u'PrVt6rfyW81\xc6\xc7\xc8\xc9\xc0\xc1\xc2\xc3\xc4\xc5',
u'PrFgczleNkNG\xc7\xc5\xc6\xc8\xc9\xc2\xc3'], (6, 6, 6))
```

6 is the success return code for the protect operation of each element in the list.

2.4.4 Using AP Python in a Development Environment

You can use the AP Python in a development environment. This is also known as a mock implementation of the AP Python APIs. In this mode, the AP Python development package provides you with sample users and data elements that can be used to simulate the behavior of the actual APIs in a production environment.

Warning: When the AP Python APIs are used with the sample users and data elements provided with the development package, the output data is only a simulation of the protected or encrypted data. Do not use the AP Python APIs in the development environment to protect, unprotect, or reprotect sensitive data.

Note: For information on the syntax of the AP Python APIs, refer to the section *Using AP Python in a Production Environment*.

2.4.4.1 Sample Users

This section describes the sample users that can be used to test the AP Python APIs in a development environment.

The following table provides a list of the sample users that you can use to test the AP Python in a development environment.

Table 2-1: List of Sample Users

Sample Users	Description
ALL_USER	Simulates a user who has the required privileges to perform all security operations, including protect, unprotect, and reprotect operations. This user has been used as the sample user in the following examples: • *Using sample data elements for simulating protect, unprotect, and reprotect scenarios* • *Using sample data elements for simulating auxiliary API scenarios* • *Using sample data elements for simulating error scenarios*
NO_PROTECT_USER	Simulates a user who does not have the required privileges to protect the data.
NO_REPROTECT_USER	Simulates a user who does not have the required privileges to reprotect the data.
NO_UNPROTECT_NULL_USER	Simulates a user who does not have the required privileges to unprotect the data. If the user tries to unprotect the data, then the *unprotect* API returns a null value.
NO_UNPROTECT_EXC_USER	Simulates a user does not have the required privileges to unprotect the data. If the user tries to unprotect the data, then the *unprotect* API throws an exception.
NO_UNPROTECT_PROTECTED_USER	Simulates a user who does not have the required privileges to unprotect the data. If the user tries to unprotect the data, then the *unprotect* API returns the protected data.
NO_USER	Simulates a user who has not been defined in the security policy.

2.4.4.2 Sample Data Elements

This section describes the sample data elements that can be used to test the AP Python APIs in a development environment.

The following table provides a list of the sample data elements that you can use to test the AP Python in a development environment.

Table 2-2: List of Sample Data Elements

Sample Data Elements	Description
SUCCESS_STR	Simulates the success scenario in the following cases: • *Protecting string and unicode data* • *Protecting string data with external IV* • *Protecting string data with external IV and external tweak* • *Protecting bulk string data* • *Protecting bulk string data with external IV* • *Protecting bulk string data with external IV and external tweak* • *Unprotecting string and unicode data* • *Unprotecting string data with external IV* • *Unprotecting string data with external IV and external tweak* • *Unprotecting bulk string data*

Sample Data Elements	Description
	• *Unprotecting bulk string data with external IV* • *Unprotecting bulk string data with external IV and external tweak*
SUCCESS_REPROTECT_STR	Simulates the success scenario in the following cases: • *Reprotecting string and unicode data* • *Reprotecting string data with external IV* • *Reprotecting string data with external IV and external tweak* • *Reprotecting bulk string data* • *Reprotecting bulk string data with external IV* • *Reprotecting bulk string data with external IV and external tweak*
SUCCESS_INT	Simulates the success scenario in the following cases: • *Protecting integer data* • *Protecting integer data with external IV* • *Protecting bulk integer data* • *Protecting bulk integer data with external IV* • *Unprotecting integer data* • *Unprotecting integer data with external IV* • *Unprotecting bulk integer data* • *Unprotecting bulk integer data with external IV*
SUCCESS_REPROTECT_INT	Simulates the success scenario in the following cases: • *Reprotecting integer data* • *Reprotecting integer data with external IV* • *Reprotecting bulk integer data* • *Reprotecting bulk integer data with external IV*
SUCCESS_LONG	Simulates the success scenario in the following cases: • *Protecting long data* • *Protecting long data with external IV* • *Protecting bulk long data* • *Protecting bulk long data with external IV* • *Unprotecting long data* • *Unprotecting long data with external IV* • *Unprotecting bulk long data* • *Unprotecting bulk long data with external IV*
SUCCESS_REPROTECT_LONG	Simulates the success scenario in the following cases: • *Reprotecting long data* • *Reprotecting long data with external IV* • *Reprotecting bulk long data* • *Reprotecting bulk long data with external IV*
SUCCESS_FLOAT	Simulates the success scenario in the following cases: • *Protecting float data* • *Protecting bulk float data* • *Unprotecting float data* • *Unprotecting bulk float data*
SUCCESS_REPROTECT_FLOAT	Simulates the success scenario in the following cases: • *Reprotecting float data*

Sample Data Elements	Description
	• *Reprotecting bulk float data*
SUCCESS_BYTE	Simulates the success scenario in the following cases: • *Protecting byte data* • *Protecting byte data with external IV* • *Protecting bulk byte data* • *Protecting bulk byte data with external IV* • *Unprotecting byte data* • *Unprotecting byte data with external IV* • *Unprotecting bulk byte data* • *Unprotecting bulk byte data with external IV*
SUCCESS_REPROTECT_BYTE	Simulates the success scenario in the following cases: • *Retokenizing Bytes Data* • *Reprotecting byte data with external IV* • *Reprotecting bulk byte data* • *Reprotecting bulk byte data with external IV*
SUCCESS_DATE_DDMMYYYY	Simulates the success scenario in the following cases: • *Protecting date object in DD/MM/YYYY format* • *Protecting bulk date objects in DD/MM/YYYY format* • *Protecting date string format in DD/MM/YYYY format* • *Protecting bulk date strings in DD/MM/YYYY format* • *Unprotecting date object in DD/MM/YYYY format* • *Unprotecting bulk date objects in DD/MM/YYYY format* • *Unprotecting date string in DD/MM/YYYY format* • *Unprotecting bulk date strings in DD/MM/YYYY format*
SUCCESS_REPROTECT_DATE_DDMMYYYY	Simulates the success scenario in the following cases: • *Reprotecting date object in DD/MM/YYYY format* • *Reprotecting bulk date objects in DD/MM/YYYY format* • *Reprotecting date string in DD/MM/YYYY format* • *Reprotecting bulk date strings in DD/MM/YYYY format*
SUCCESS_DATE_MMDDYYYY	Simulates the success scenario in the following cases: • *Protecting date object in MM/DD/YYYY format* • *Protecting bulk date objects in MM/DD/YYYY format* • *Protecting date string in MM/DD/YYYY format* • *Protecting bulk date strings in MM/DD/YYYY format* • *Unprotecting date object in MM/DD/YYYY format* • *Unprotecting bulk date objects in MM/DD/YYYY format* • *Unprotecting date string in MM/DD/YYYY format* • *Unprotecting bulk date strings in MM/DD/YYYY format*
SUCCESS_REPROTECT_DATE_MMDDYYYY	Simulates the success scenario in the following cases: • *Reprotecting date object in MM/DD/YYYY format* • *Reprotecting bulk date objects in MM/DD/YYYY format* • *Reprotecting date string in MM/DD/YYYY format* • *Reprotecting bulk date strings in MM/DD/YYYY format*
SUCCESS_DATE_YYYYMMDD	Simulates the success scenario in the following cases:

Sample Data Elements	Description
	• *Protecting date object in YYYY/MM/DD format* • *Protecting bulk date objects in YYYY/MM/DD format* • *Protecting date string in YYYY/MM/DD format* • *Protecting bulk date strings in YYYY/MM/DD format* • *Unprotecting date object in YYYY/MM/DD format* • *Unprotecting bulk date objects in YYYY/MM/DD format* • *Unprotecting date string in YYYY/MM/DD format* • *Unprotecting bulk date strings in YYYY/MM/DD format*
SUCCESS_REPROTECT_DATE_YYYYMMDD	Simulates the success scenario in the following cases: • *Reprotecting date object in YYYY/MM/DD format* • *Reprotecting bulk date objects in YYYY/MM/DD format* • *Reprotecting date string in YYYY/MM/DD format* • *Reprotecting bulk date strings in YYYY/MM/DD format*
SUCCESS_DATETIME	Simulates the success scenario in the following cases: • *Protecting date and time strings* • *Protecting bulk date and time strings* • *Unprotecting date and time string* • *Unprotecting bulk date and time strings*
SUCCESS_REPROTECT_DATETIME	Simulates the success scenario in the following cases: • *Reprotecting date and time strings* • *Reprotecting bulk date and time strings*
SUCCESS_ENC	Simulates the success scenario in the following cases: • *Encrypting string and unicode data* • *Encrypting bulk string data* • *Decrypting string and unicode data* • *Decrypting bulk string data* • *Encrypting integer data* • *Encrypting bulk integer data* • *Decrypting integer data* • *Decrypting bulk integer data* • *Encrypting long data* • *Encrypting bulk long data* • *Decrypting long data* • *Decrypting bulk long data* • *Encrypting float data* • *Encrypting bulk float data* • *Decrypting float data* • *Decrypting bulk float data* • *Encrypting byte data* • *Encrypting bulk byte data* • *Decrypting byte data* • *Decrypting bulk byte data*
SUCCESS_REPROTECT_ENC	Simulates the success scenario in the following cases: • *Re-encrypting byte data* • *Re-encrypting bulk byte data*

Sample Data Elements	Description
SUCCESS_CHECK_ACCESS	Simulates the *success scenario of checking access permissions*.
FAIL_CHECK_ACCESS	Simulates the *failure scenario of checking access permissions*.
POLICY_NAME	Simulates the *success scenario for retrieving the default data element*.
EXCEPTION_INVALID_USER	Simulates the *error scenario if an invalid user is used to protect data*.
EXCEPTION_INVALID_DE	Simulates the *error scenario if an invalid data element is used to protect data*.
EXCEPTION_TWEAK_IS_NULL	Simulates the *error scenario if a null external tweak is used to protect data*.
DATA_TOO_SHORT	Simulates the *error scenario if the data to be protected is too short*.
USER_TOO_LONG	Simulates the *error scenario if the name of the user is too long*.
EXCEPTION_UNSUPPORTED_ALGORITHM	Simulates the *error scenario if the protection method used to protect the data is not supported*.
EMPTY_POLICY	Simulates the *error scenario if a policy is empty*.
LICENSE_EXPIRED	Simulates the *error scenario if the protector license has expired*.
INPUT_NOT_VALID	Simulates the *error scenario if input is invalid*.
REPROTECT_HETERO_STR	Simulates the *error scenario of reprotecting string data with heterogenous data elements*.

2.4.4.3 Using Sample Data Elements for Simulating Protect, Unprotect, and Reprotect Scenarios

This section describes how to use the sample data elements for simulating the protect, unprotect, and reprotect scenarios.

> **Note:** In the mock implementation, you must pass the *ALL_USER* username as an argument to the *create_session* API for creating a session.

2.4.4.3.1 Mock Example - Protecting String

This section describes how to use the *protect* API for protecting a string input data.

Example: Input string data

In the following example, the *Protegrity1* string is used as the input data, which is protected using the *SUCCESS_STR* data element.

```
from appython import Protector
protector = Protector()
session = protector.create_session("ALL_USER")
output = session.protect("Protegrity1", "SUCCESS_STR")
print("Protected Data: %s" %output)
```

Result

```
Protected Data: 6JPqrjJEqLX
```

2.4.4.3.2 Mock Example - Protecting String Data with External IV

This section describes how to use the *protect* API for protecting string input data using external IV.

> **Note:** If you want to pass the external IV as a keyword argument to the *protect* API, then you must pass the external IV as bytes to the API.

Example

In the following example, the *Protegrity1* string is used as the input data, which is tokenized using the *SUCCESS_STR* data element, with the help of external IV *1234* that is passed as bytes.

```
from appython import Protector
protector = Protector()
session = protector.create_session("ALL_USER")
output = session.protect("Protegrity1", "SUCCESS_STR",
         external_iv=bytes("1234"))
print("Protected Data: %s" %output)
```

Result

```
Protected Data: Ho9bgXoebxa
```

2.4.4.3.3 Mock Example - Protecting String Data Using External IV and External Tweak

This section describes how to use the *protect* API for protecting string input data using external IV and external tweak.

> **Note:** If you want to pass the external IV and external tweak as keyword arguments to the *protect* API, then you must pass the external IV and external tweak as bytes.

Example

In the following example, the *protegrity1234* string is used as the input data, which is protected using the data element *SUCCESS_STR*, with the help of external IV *1234* and external tweak *abcdef* that are passed as bytes.

```
from appython import Protector
protector = Protector()
session = protector.create_session("ALL_USER")
output = session.protect("protegrity1234", "SUCCESS_STR",
         external_iv=bytes("1234", encoding="utf-8"), external_tweak=bytes("abcdef",
encoding="utf-8"))
print("Protected Data: %s" %output)
```

Result

```
Protected Data: 9GsvVbGRvTQwxr
```

2.4.4.3.4 Mock Example - Protecting Bulk String Data

This section describes how to use the *protect* API for protecting bulk string input data. You can pass bulk string input data as a list or a tuple.

> **Caution:** The individual elements of the list or tuple must be of the same data type.

Example 1

In the following example, *protegrity1234*, *Protegrity1*, and *Protegrity56* strings are stored in a list and used as bulk data, which is protected using the *SUCCESS_STR* data element.

```
from appython import Protector
protector = Protector()
session = protector.create_session("ALL_USER")
data = ["protegrity1234", "Protegrity1", "Protegrity56"]
p_out = session.protect(data, "SUCCESS_STR")
print("Protected Data: ")
print(p_out)
```

Result

```
Protected Data:
(['pJPqrjJEqLXHaO', '6JPqrjJEqLX', '6JPqrjJEqLl5'], (6, 6, 6))
```

6 is the success return code for the protect operation of each element in the list.

Example 2

In *Example 1*, the protected output was a tuple of the tokenized data and the error list. The following example shows how you can tweak the code to ensure that you retrieve the protected output and the error list separately, and not as part of a tuple.

```
from appython import Protector
    protector = Protector()
    session = protector.create_session("ALL_USER")
    data = "protegrity1234"
    data = [data]*5
    p_out, error_list = session.protect(data, "SUCCESS_STR")
    print("Protected Data: ", end="")
    print(p_out)
    print("Error List: ", end="")
    print(error_list)
```

Result

```
Protected Data: ['pJPqrjJEqLXHaO', 'pJPqrjJEqLXHaO', 'pJPqrjJEqLXHaO',
'pJPqrjJEqLXHaO', 'pJPqrjJEqLXHaO']
Error List: (6, 6, 6, 6, 6)
```

6 is the success return code for the protect operation of each element in the list.

2.4.4.3.5 Mock Example - Protecting Bulk String Data with External IV

This section describes how to use the *protect* API for protecting bulk string input data using external IV. You can pass bulk string data as a list or a tuple.

Caution: The individual elements of the list or tuple must be of the same data type.

Note: If you want to pass the external IV as a keyword argument to the *protect* API, then you must pass the external IV as bytes to the API.

Example

In the following example, *protegrity1234*, *Protegrity1*, and *Protegrity56* strings are stored in a list and used as bulk input data, which is protected using the *SUCCESS_STR* data element, with the help of external IV *123* that is passed as bytes.

```
from appython import Protector
protector = Protector()
session = protector.create_session("ALL_USER")
data = ["protegrity1234", "Protegrity1", "Protegrity56"]
p_out = session.protect(data, "SUCCESS_STR",
        external_iv=bytes("123"))
print("Protected Data: ")
print(p_out)
```

Result

```
Protected Data:
(['nx8mEaxwmR2VSq', '1x8mEaxwmR2', '1x8mEaxwmRdF'], (6, 6, 6))
```

6 is the success return code for the protect operation of each element in the list.

2.4.4.3.6 Mock Example - Protecting Bulk String Data Using External IV and External Tweak

This section describes how to use the *protect* API for protecting bulk string input data using external IV and external tweak. You can pass bulk string data as a list or a tuple.

Caution: The individual elements of the list or tuple must be of the same data type.

Note: If you want to pass the external IV and external tweak as keyword arguments to the *protect* API, then you must pass the external IV and external tweak as bytes.

Example

In the following example, *protegrity1234ÀÁ*, *Protegrity1ÆÇÈ*, and *Protegrity56ÀÁÂÃÄÅ* strings are stored in a list and used as bulk input data. This bulk data is protected using the *SUCCESS_STR* data element, with the help of external IV *1234* and external tweak *xyz* that are both passed as bytes.

```
from appython import Protector
protector = Protector()
session = protector.create_session("ALL_USER")
data = ["protegrity1234ÀÁ", "Protegrity1ÆÇÈ", "Protegrity56ÀÁÂÃÄÅ"]
p_out = session.protect(data, "SUCCESS_STR",
        external_iv=bytes("1234", encoding="utf-8"), external_tweak=bytes("xyz",
encoding="utf-8"))
print("Protected Data: ")
print(p_out)
```

Result

```
Protected Data:
(['uc72ntca2dI896Ã\x83Â\x80Ã\x83Â\x81', 'xc72ntca2dIÃ\x83Â\x86Ã\x83Â\x87Ã\x83Â\x88',
'xc72ntca2dEBÃ\x83Â\x80Ã\x83Â\x81Ã\x83Â\x82Ã\x83Â\x83Ã\x83Â\x84Ã\x83Â\x85'], (6, 6, 6))
```

6 is the success return code for the protect operation of each element in the list.

2.4.4.3.7 Mock Example - Unprotecting String Data

This section describes how to use the *unprotect* API for retrieving the original string data from the protected data.

Example: Input string data

In the following example, the *Protegrity1* string that was protected using the *SUCCESS_STR* data element, is now unprotected using the same data element.

```
from appython import Protector
protector = Protector()
session = protector.create_session("ALL_USER")
output = session.protect("Protegrity1", "SUCCESS_STR")
print("Protected Data: %s" %output)
org = session.unprotect(output, "SUCCESS_STR")
print("Unprotected Data: %s" %org)
```

Result

```
Protected Data: 6JPqrjJEqLX
Unprotected Data: Protegrity1
```

2.4.4.3.8 Mock Example - Unprotecting String Data with External IV

This section describes how to use the *unprotect* API for retrieving the original string data from protected data, using external initialization vector (IV).

> **Note:** If you want to pass the external IV as a keyword argument to the *unprotect* API, then you must pass the external IV as bytes to the API.

Example

In the following example, the *Protegrity1* string that was protected using the *SUCCESS_STR* data element and the external IV *1234* is now unprotected using the same data element and same external IV.

```
from appython import Protector
protector = Protector()
session = protector.create_session("ALL_USER")
output = session.protect("Protegrity1", "SUCCESS_STR",
         external_iv=bytes("1234", encoding="utf-8"))
print("Protected Data: %s" %output)
org = session.unprotect(output, "SUCCESS_STR",
      external_iv=bytes("1234", encoding="utf-8"))
print("Unprotected Data: %s" %org)
```

Result

```
Protected Data: Ho9bgXoebxa
Unprotected Data: Protegrity1
```

2.4.4.3.9 Mock Example - Unprotecting String Data Using External IV and External Tweak

This section describes how to use the *unprotect* API for unprotecting string data using external IV and tweak.

> **Note:** If you want to pass the external IV and external tweak as keyword arguments to the *unprotect* API, then you must pass the external IV and external tweak as bytes.

Example

In the following example, the *protegrity1234* string that was protected using the *SUCCESS_STR* data element, is now unprotected using the same data element and the same external IV and external tweak.

```
from appython import Protector
protector = Protector()
session = protector.create_session("ALL_USER")
output = session.protect("protegrity1234", "SUCCESS_STR",
         external_iv=bytes("1234", encoding="utf-8"), external_tweak=bytes("abcdef",
encoding="utf-8"))
print("Protected Data: %s" %output)
org = session.unprotect(output, "SUCCESS_STR",
      external_iv=bytes("1234", encoding="utf-8"), external_tweak=bytes("abcdef",
encoding="utf-8"))
print("Unprotected Data: %s" %org)
```

Result

```
Protected Data: 9GsvVbGRvTQwxr
Unprotected Data: protegrity1234
```

2.4.4.3.10 Mock Example - Unprotecting Bulk String Data

This section describes how to use the *unprotect* API for retrieving the original bulk string data from the protected data.

Example 1

In the following example, *protegrity1234*, *Protegrity1*, and *Protegrity56* strings are stored in a list and used as bulk data, which is protected using the *SUCCESS_STR* data element. The bulk string data is then unprotected using the same data element.

```
from appython import Protector
protector = Protector()
session = protector.create_session("ALL_USER")
data = ["protegrity1234", "Protegrity1", "Protegrity56"]
p_out = session.protect(data, "SUCCESS_STR")
print("Protected Data: ")
print(p_out)
out = session.unprotect(p_out[0], "SUCCESS_STR")
print("Unprotected Data: ")
print(out)
```

Result

```
Protected Data:
(['pJPqrjJEqLXHaO', '6JPqrjJEqLX', '6JPqrjJEqLl5'], (6, 6, 6))
Unprotected Data:
(['protegrity1234', 'Protegrity1', 'Protegrity56'], (8, 8, 8))
```

6 is the success return code for the protect operation of each element in the list.

8 is the success return code for the unprotect operation of each element in the list.

Example 2

In *Example 1*, the unprotected output was a tuple of the unprotected data and the error list. The following example shows how you can tweak the code to ensure that you retrieve the unprotected output and the error list separately, and not as part of a tuple.

```
from appython import Protector
    protector = Protector()
    session = protector.create_session("ALL_USER")
    data = protegrity1234
    data = [data]*5
    p_out, error_list = session.protect(data, "SUCCESS_STR")
    print("Protected Data: ", end="")
    print(p_out)
    print("Error List: ", end="")
    print(error_list)
    org, error_list = session.unprotect(p_out, "SUCCESS_STR")
    print("Unprotected Data: ", end="")
    print(org)
    print("Error List: ", end="")
    print(error_list)
```

Result

```
Protected Data: ['pJPqrjJEqLXHaO', 'pJPqrjJEqLXHaO', 'pJPqrjJEqLXHaO',
'pJPqrjJEqLXHaO', 'pJPqrjJEqLXHaO']
Error List: (6, 6, 6, 6, 6)
Unprotected Data: ['protegrity1234', 'protegrity1234', 'protegrity1234',
'protegrity1234', 'protegrity1234']
Error List: (8, 8, 8, 8, 8)
```

6 is the success return code for the protect operation of each element in the list.

8 is the success return code for the unprotect operation of each element in the list.

2.4.4.3.11 Mock Example - Unprotecting Bulk String Data with External IV

This section describes how to use the *unprotect* API for retrieving the original bulk string data from protected data using external IV.

> **Note:** If you want to pass the external IV as a keyword argument to the *unprotect* API, then you must pass the external IV as bytes to the API.

Example

In the following example, *protegrity1234*, *Protegrity1*, and *Protegrity56* strings are stored in a list and used as bulk data, which is protected using the *SUCCESS_STR* data element and external IV *123*. The bulk string data is then unprotected using the same data element and same external IV.

```
from appython import Protector
protector = Protector()
session = protector.create_session("ALL_USER")
data = ["protegrity1234", "Protegrity1", "Protegrity56"]
p_out = session.protect(data, "SUCCESS_STR",
        external_iv=bytes("123"))
print("Protected Data: ")
print(p_out)
out = session.unprotect(p_out[0], "SUCCESS_STR",
      external_iv=bytes("123"))
print("Unprotected Data: ")
print(out)
```

Result

```
Protected Data:
(['nx8mEaxwmR2VSq', '1x8mEaxwmR2', '1x8mEaxwmRdF'], (6, 6, 6))
Unprotected Data:
(['protegrity1234', 'Protegrity1', 'Protegrity56'], (8, 8, 8))
```

6 is the success return code for the protect operation of each element in the list.

8 is the success return code for the unprotect operation of each element in the list.

2.4.4.3.12 Mock Example - Unprotecting Bulk String Data Using External IV and External Tweak

This section describes how to use the *unprotect* API for retrieving the original bulk string data from protected data using external IV and external tweak.

> **Note:** If you want to pass the external IV and external tweak as keyword arguments to the *protect* API, then you must pass the external IV and external tweak as bytes.

Example

In the following example, *protegrity1234ÀÁ*, *Protegrity1ÆÇÈ*, and *Protegrity56ÀÁÂÃÄÅ* strings are stored in a list and used as bulk data. This bulk data is protected using the *SUCCESS_STR* data element, with the help of external IV *1234* and external tweak *xyz* that are both passed as bytes. The protected bulk string data is then unprotected using the same data element, same external IV, and external tweak.

```
from appython import Protector
protector = Protector()
session = protector.create_session("ALL_USER")
data = ["protegrity1234ÀÁ", "Protegrity1ÆÇÈ", "Protegrity56ÀÁÂÃÄÅ"]
p_out = session.protect(data, "SUCCESS_STR",
        external_iv=bytes("1234", encoding="utf-8"), external_tweak=bytes("xyz",
encoding="utf-8"))
```

```
print("Protected Data: ")
print(p_out)
out = session.unprotect(p_out[0], "SUCCESS_STR",
      external_iv=bytes("1234", encoding="utf-8"),  external_tweak=bytes("xyz",
encoding="utf-8"))
print("Unprotected Data: ")
print(out)
```

Result

```
Protected Data:
(['uc72ntca2dI896Ã\x83Â\x80Ã\x83Â\x81', 'xc72ntca2dIÃ\x83Â\x86Ã\x83Â\x87Ã\x83Â\x88',
'xc72ntca2dEBÃ\x83Â\x80Ã\x83Â\x81Ã\x83Â\x82Ã\x83Â\x83Ã\x83Â\x84Ã\x83Â\x85'], (6, 6, 6))
Unprotected Data:
([u'protegrity1234\xc0\xc1', u'Protegrity1\xc6\xc7\xc8',
u'Protegrity56\xc0\xc1\xc2\xc3\xc4\xc5'], (8, 8, 8))
```

6 is the success return code for the protect operation of each element in the list.

8 is the success return code for the unprotect operation of each element in the list.

2.4.4.3.13 Mock Example - Reprotecting String

This section describes how to use the *reprotect* API for reprotecting string data.

Warning: If you are retokenizing the data using the *reprotect* API, then the old data element and the new data element must have the same tokenization type. For example, if you have used Alpha-Numeric data element to protect the data, then you must use only Alpha-Numeric data element to reprotect the data.

Example: Input string data

In the following example, the *Protegrity1* string is used as the input data, which is first protected using the *SUCCESS_STR* data element.

The protected input data, the old data element *SUCCESS_STR*, and a new data element *SUCCESS_REPROTECT_STR* are then passed as inputs to the *reprotect* API. The *reprotect* API first unprotects the protected input data using the old data element, and then reprotects it using the new data element, as part of a single reprotect operation.

```
from appython import Protector
protector = Protector()
session = protector.create_session("ALL_USER")
output = session.protect("Protegrity1", "SUCCESS_STR")
print("Protected Data: %s" %output)
r_out = session.reprotect(output, "SUCCESS_STR",
        "SUCCESS_REPROTECT_STR")
print("Reprotected Data: %s" %r_out)
```

Result

```
Protected Data: 6JPqrjJEqLX
Reprotected Data: JQbePhQ2eGC
```

2.4.4.3.14 Mock Example - Reprotecting String Data with External IV

This section describes how to use the *reprotect* API for reprotecting string data using external IV.

Warning: If you are retokenizing the data using the *reprotect* API, then the old data element and the new data element must have the same tokenization type. For example, if you have used Alpha-Numeric data element to protect the data, then you must use only Alpha-Numeric data element to reprotect the data.

Note: If you want to pass the external IV as a keyword argument to the *reprotect* API, then you must pass the external IV as bytes to the API.

Example

In the following example, the *Protegrity1* string is used as the input data, which is first protected using the *SUCCESS_STR* data element, with the help of external IV *1234* that is passed as bytes.

The protected input data, the old data element *SUCCESS_STR*, a new data element *SUCCESS_REPROTECT_STR*, the old external IV *1234* in bytes, and a new external IV *123456* in bytes are then passed as inputs to the *reprotect* API. As part of a single reprotect operation, the *reprotect* API first unprotects the protected input data using the given data element and old external IV, and then reprotects it using the new data element and new external IV.

```
from appython import Protector
protector = Protector()
session = protector.create_session("ALL_USER")
p_out = session.protect("Protegrity1", "SUCCESS_STR",
        external_iv=bytes("1234", encoding="utf-8"))
print("Protected Data: %s" %p_out)
r_out = session.reprotect(p_out, "SUCCESS_STR",
        "SUCCESS_REPROTECT_STR", old_external_iv=bytes("1234", encoding="utf-8"),
        new_external_iv=bytes("123456", encoding="utf-8"))
print("Reprotected Data: %s" %r_out)
```

Result

```
Protected Data: Ho9bgXoebxa
Reprotected Data: vQIqelQyqY6
```

2.4.4.3.15 Mock Example - Reprotecting String Data Using External IV and External Tweak

This section describes how to use the *reprotect* API for reprotecting string data using external IV and external tweak.

Warning: If you are using Format Preserving Encryption (FPE) with the *reprotect* API, then ensure that the plaintext alphabet type and the plaintext encoding used for FPE must be the same for both protecting and reprotecting the data. For example, if you have used FPE-Numeric data element with UTF-8 encoding to protect the data, then you must use only FPE-Numeric data element with UTF-8 encoding to reprotect the data.

Note: If you want to pass the external IV and external tweak as keyword arguments to the *reprotect* API, then you must pass the external IV and external tweak as bytes.

Example

In the following example, the *protegrity1234* string is used as the data, which is first protected using the *SUCCESS_STR* data element, with the help of external IV *1234* and external tweak *abcdef* that are both passed as bytes.

The protected input data, the *SUCCESS_STR* data element, a new data element *SUCCESS_REPROTECT_STR*, the old external IV *1234* and external tweak *abcdef* in bytes, and a new external IV *123456* and external tweak *xyz* in bytes are then passed as inputs to the *reprotect* API. As part of a single reprotect operation, the *reprotect* API first unprotects the

protected input data using the given data element, and old external IV and external tweak, and then reprotects it using the same data element, but with the new external IV and external tweak.

```
from appython import Protector
protector = Protector()
session = protector.create_session("ALL_USER")
p_out = session.protect("protegrity1234", "SUCCESS_STR",
        external_iv=bytes("1234", encoding="utf-8"), external_tweak=bytes("abcdef"))
print("Protected Data: %s" %p_out)
r_out = session.reprotect(p_out, "SUCCESS_STR",
        "SUCCESS_REPROTECT_STR", old_external_iv=bytes("1234", encoding="utf-8"),
        new_external_iv=bytes("12345", encoding="utf-8"),
old_external_tweak=bytes("abcdef", encoding="utf-8"),
        new_external_tweak=bytes("xyz"))
print("Reprotected Data: %s" %r_out)
```

Result

```
Protected Data: 9GsvVbGRvTQwxr
Reprotected Data: 3AZjIrAvjOsnwb
```

2.4.4.3.16 Mock Example - Reprotecting Bulk String Data

This section describes how to use the *reprotect* API for reprotecting bulk string data. You can pass bulk string data as a list or a tuple.

Caution: The individual elements of the list or tuple must be of the same data type.

Warning: If you are retokenizing the data using the *reprotect* API, then the old data element and the new data element must have the same tokenization type. For example, if you have used Alpha-Numeric data element to protect the data, then you must use only Alpha-Numeric data element to reprotect the data.

Example

In the following example, *protegrity1234*, *Protegrity1*, and *Protegrity56* strings are stored in a list and used as bulk data, which is protected using the *SUCCESS_STR* data element.

The protected input data, the old data element *SUCCESS_STR*, and a new data element *SUCCESS_REPROTECT_STR* are then passed as inputs to the *reprotect* API. The *reprotect* API first unprotects the protected input data using the old data element and then reprotects it using the new data element, as part of a single reprotect operation.

```
from appython import Protector
protector = Protector()
session = protector.create_session("ALL_USER")
data = ["protegrity1234", "Protegrity1", "Protegrity56"]
p_out = session.protect(data, "SUCCESS_STR")
print("Protected Data: ")
print(p_out)
r_out = session.reprotect(p_out[0], "SUCCESS_STR",
        "SUCCESS_REPROTECT_STR")
print("Reprotected Data: ")
print(r_out)
```

Result

```
Protected Data:
['pJPqrjJEqLXHaO', '6JPqrjJEqLX', '6JPqrjJEqLl5'], (6, 6, 6))
Reprotected Data:
(['gQbePhQ2eGCjqW', 'JQbePhQ2eGC', 'JQbePhQ2eGBK'], (6, 6, 6))
```

6 is the success return code for the protect operation of each element in the list.

2.4.4.3.17 Mock Example - Reprotecting Bulk String Data with External IV

This section describes how to use the *reprotect* API for reprotecting bulk string data using external IV. You can pass bulk string data as a list or a tuple.

Caution: The individual elements of the list or tuple must be of the same data type.

Warning: If you are retokenizing the data using the *reprotect* API, then the old data element and the new data element must have the same tokenization type. For example, if you have used Alpha-Numeric data element to protect the data, then you must use only Alpha-Numeric data element to reprotect the data.

Note: If you want to pass the external IV as a keyword argument to the *reprotect* API, then you must pass the external IV as bytes to the API

Example

In the following example, *protegrity1234*, *Protegrity1*, and *Protegrity56* strings are stored in a list and used as bulk data, which is protected using the *SUCCESS_STR* data element, with the help of an external IV *1234* that is passed as bytes.

The protected input data, the old data element *SUCCESS_STR*, a new data element *SUCCESS_REPROTECT_STR*, the old external IV *1234* in bytes, and a new external IV *123456* in bytes are then passed as inputs to the *reprotect* API. As part of a single reprotect operation, the *reprotect* API first unprotects the protected input data using the given data element and old external IV, and then reprotects it using the new data element and new external IV.

```
from appython import Protector
protector = Protector()
session = protector.create_session("ALL_USER")
data = ["protegrity1234", "Protegrity1", "Protegrity56"]
p_out = session.protect(data, "SUCCESS_STR",
        external_iv=bytes("1234", encoding="utf-8"))
print("Protected Data: ")
print(p_out)
r_out = session.reprotect(p_out[0], "SUCCESS_STR",
        "SUCCESS_REPROTECT_STR", old_external_iv=bytes("1234", encoding="utf-8"),
        new_external_iv=bytes("123456", encoding="utf-8"))
print("Reprotected Data: ")
print(r_out)
```

Result

```
Protected Data:
(['fo9bgXoebxaCTN', 'Ho9bgXoebxa', 'Ho9bgXoebx2q'], (6, 6, 6))
Reprotected Data:
(['cQIqelQyqY6OoN', 'vQIqelQyqY6', 'vQIqelQyqYXa'], (6, 6, 6))
```

6 is the success return code for the protect operation of each element in the list.

2.4.4.3.18 Mock Example - Reprotecting Bulk String Data Using External IV and External Tweak

This section describes how to use the *reprotect* API for reprotecting bulk string data using external IV and external tweak. You can pass bulk string data as a list or a tuple.

Caution: The individual elements of the list or tuple must be of the same data type.

Note: If you want to pass the external IV and external tweak as keyword arguments to the *reprotect* API, then you must pass the external IV and external tweak as bytes.

Example

In the following example, *protegrity1234ÀÁ*, *Protegrity1ÆÇÈ*, and *Protegrity56ÀÁÂÃÄÅ* strings are stored in a list and used as bulk data, which is first protected using the *SUCCESS_STR* data element, with the help of an external IV *1234* and external tweak *abc* that are both passed as bytes.

The protected input data, the old data element *SUCCESS_STR*, a new data element *SUCCESS_REPROTECT_STR*, the old external IV *1234* and external tweak *abc* in bytes, and a new external IV *123456* and external tweak *xyz* in bytes are then passed as inputs to the *reprotect* API. As part of a single reprotect operation, the *reprotect* API first unprotects the protected input data using the old data element, and old external IV and external tweak, and then reprotects it using the new data element, new external IV, and external tweak.

```
from appython import Protector
protector = Protector()
session = protector.create_session("ALL_USER")
data = ["protegrity1234ÀÁ", "Protegrity1ÆÇÈ", "Protegrity56ÀÁÂÃÄÅ"]
p_out = session.protect(data, "SUCCESS_STR",
        external_iv=bytes("1234", encoding="utf-8"), external_tweak=bytes("abc",
encoding="utf-8"))
print("Protected Data: ")
print(p_out)
r_out = session.reprotect(p_out[0], "SUCCESS_STR",
        "SUCCESS_REPROTECT_STR", old_external_iv=bytes("1234", encoding="utf-8"),
        new_external_iv=bytes("123456", encoding="utf-8"),
old_external_tweak=bytes("abc", encoding="utf-8"),
        new_external_tweak=bytes("xyz", encoding="utf-8"))
print("Reprotected Data: ")
print(r_out)
```

Result

```
Protected Data:
(['A2XIHe2vIEFmShÃ\x83Â\x80Ã\x83Â\x81', 'N2XIHe2vIEFÃ\x83Â\x86Ã\x83Â\x87Ã\x83Â\x88',
'N2XIHe2vIE5oÃ\x83Â\x80Ã\x83Â\x81Ã\x83Â\x82Ã\x83Â\x83Ã\x83Â\x84Ã\x83Â\x85'],
(6, 6, 6))
Reprotected Data:
(['VmDBawmlBAsfc6Ã\x83Â\x83Ã\x82Â\x83Ã\x83Â\x82Ã\x82Â\x83Ã\x83Â\x83Ã\x82Â\x82Ã\x83Â\x82Ã
\x82Â\x83Ã\x83Â\x83Ã\x82Â\x83Ã\x83Â\x82Ã\x82Â\x82Ã\x83Â\x83Â\x82Â\x82Ã\x83Â\
x82Ã\x82Â\x83Ã\x83Â\x83Ã\x82Â\x83Ã\x83Â\x82Ã\x82Â\x83Ã\x83Â\x83Ã\x82Â\x82Ã\x83Â\x82Ã\x82
Â\x82Ã\x83Â\x83Ã\x82Â\x83Ã\x83Â\x82Ã\x82Â\x82Ã\x83Â\x83Â\x82Â\x82Ã\x83Â\x82Ã\
x82Â\x80Ã\x83Â\x83Ã\x82Â\x83Ã\x83Â\x82Ã\x82Â\x83Ã\x83Â\x83Ã\x82Â\x82Ã\x83Â\x82Ã\x82Â\x83
Ã\x83Â\x83Ã\x82Â\x83Ã\x83Â\x82Ã\x82Â\x82Ã\x83Â\x83Ã\x82Â\x82Ã\x83Â\x82Ã\x82Â\
x83Ã\x83Â\x83Ã\x82Â\x83Ã\x83Â\x82Ã\x82Â\x83Ã\x83Â\x83Ã\x82Â\x82Ã\x83Â\x82Ã\x82Â\x82Ã\x83
Â\x83Ã\x82Â\x83Ã\x83Â\x82Ã\x82Â\x82Ã\x83Â\x83Ã\x82Â\x82Ã\x83Â\x82Ã\x82Â\x81',
'emDBawmlBAsÃ\x83Â\x83Ã\x82Â\x83Ã\x83Â\x82Ã\x82Â\x83Ã\x83Â\x83Ã\x82Â\x82Ã\x83Â\x82Ã\x82Â
\x83Ã\x83Â\x83Ã\x82Â\x83Ã\x83Â\x82Ã\x82Â\x82Ã\x83Â\x83Ã\x82Â\x82Ã\x83Â\x82Ã\x82Â\
x83Ã\x83Â\x83Ã\x82Â\x83Ã\x83Â\x82Ã\x82Â\x83Ã\x83Â\x83Ã\x82Â\x82Ã\x83Â\x82Ã\x82Â\x82Ã\x83
Â\x83Ã\x82Â\x83Ã\x83Â\x82Ã\x82Â\x82Ã\x83Â\x83Ã\x82Â\x82Ã\x83Â\x82Ã\x82Â\x86Ã\x83Â\
x83Ã\x82Â\x83Ã\x83Â\x82Ã\x82Â\x83Ã\x83Â\x83Ã\x82Â\x82Ã\x83Â\x82Ã\x82Â\x83Ã\x83Â\x83Ã\x82
Â\x83Ã\x83Â\x82Ã\x82Â\x82Ã\x83Â\x83Ã\x82Â\x82Ã\x83Â\x82Ã\x82Â\x83Ã\x83Â\x83Ã\x82Â\
x83Ã\x83Â\x82Ã\x82Â\x83Ã\x83Â\x83Ã\x82Â\x82Ã\x83Â\x82Ã\x82Â\x82Ã\x83Â\x83Ã\x82Â\x83Ã\x83
Â\x82Ã\x82Â\x83Ã\x83Â\x82Ã\x82Â\x82Ã\x83Â\x82Ã\x82Â\x87Ã\x83Â\x83Ã\x82Â\x83Ã\x83Â\
x82Ã\x82Â\x83Ã\x83Â\x83Ã\x82Â\x83Ã\x82Â\x83Ã\x82Â\x83Ã\x83Â\x83Ã\x82Â\x83Ã\x83Â\x82Ã\x82
Â\x82Ã\x83Â\x83Ã\x82Â\x83Ã\x82Â\x82Ã\x83Â\x82Ã\x82Â\x83Ã\x83Â\x83Ã\x82Â\x83Ã\x83Â\x82Ã\x82Â\
x83Ã\x83Â\x83Ã\x82Â\x82Ã\x83Â\x82Ã\x82Â\x82Ã\x83Â\x83Ã\x82Â\x83Ã\x83Â\x82Ã\x82Â\x82Ã\x83
Â\x83Ã\x82Â\x82Ã\x83Â\x82Ã\x82Â\x88','emDBawmlBAtIÃ\x83Â\x83Ã\x82Â\x83Ã\x83Â\x82Ã\
x82Â\x83Ã\x83Â\x83Ã\x82Â\x82Ã\x83Â\x82Ã\x82Â\x83Ã\x83Â\x83Ã\x82Â\x83Ã\x83Â\x82Ã\x82Â\x82
Ã\x83Â\x83Ã\x82Â\x82Ã\x83Â\x82Ã\x82Â\x83Ã\x83Â\x83Ã\x82Â\x83Ã\x83Â\x82Ã\x82Â\x83Ã\
x83Â\x83Ã\x82Â\x82Ã\x83Â\x82Ã\x82Â\x82Ã\x83Â\x83Ã\x82Â\x83Ã\x83Â\x82Ã\x82Â\x82Ã\x83Â\x83
Ã\x82Â\x82Ã\x83Â\x82Ã\x82Â\x80Ã\x83Â\x83Ã\x82Â\x83Ã\x83Â\x82Ã\x82Â\x83Ã\x83Â\x83Ã\
x82Â\x82Ã\x83Â\x82Ã\x82Â\x83Ã\x83Â\x83Ã\x82Â\x83Ã\x83Â\x83Ã\x82Â\x82Ã\x83Â\x83Ã\x82Â\x82
Ã\x83Â\x82Ã\x82Â\x83Ã\x83Â\x83Ã\x82Â\x83Ã\x83Â\x83Ã\x82Â\x82Ã\x83Â\x83Ã\x82Â\x82Ã\
x83Â\x82Ã\x82Â\x82Ã\x83Â\x83Ã\x82Â\x83Ã\x83Â\x83Ã\x82Â\x82Ã\x83Â\x83Ã\x82Â\x82Ã\x83Â\x82
Ã\x82Â\x81Ã\x83Â\x83Ã\x82Â\x83Ã\x83Â\x82Ã\x82Â\x82Ã\x83Â\x83Ã\x83Â\x82Ã\x82Â\x83Ã\x82Â\
x82Â\x83Ã\x83Â\x83Ã\x82Â\x83Ã\x83Â\x82Ã\x82Â\x82Ã\x83Â\x83Ã\x82Â\x82Ã\x83Â\x82Ã\x82Â\x83
Ã\x83Â\x83Ã\x82Â\x83Ã\x83Â\x82Ã\x82Â\x83Ã\x83Â\x83Ã\x82Â\x82Ã\x83Â\x82Ã\x82Â\x82Ã\
x83Â\x83Ã\x82Â\x83Ã\x83Â\x82Ã\x82Â\x82Ã\x83Â\x83Ã\x82Â\x82Ã\x83Â\x82Ã\x82Â\x82Ã\x83Â\x83
Ã\x82Â\x83Ã\x83Â\x83Ã\x82Â\x82Ã\x83Â\x83Ã\x83Â\x82Ã\x82Â\x83Ã\x82Â\x82Ã\x83Â\x83Â\x83Ã\
x82Â\x83Ã\x83Â\x83Ã\x82Â\x82Ã\x82Â\x83Ã\x83Â\x83Ã\x82Â\x82Ã\x83Â\x82Ã\x82Â\x83Ã\x83Â\x83Ã\
```

```
Ã\x83Â\x82Ã\x82Â\x83Ã\x83Â\x83Ã\x82Â\x82Ã\x83Â\x82Ã\x82Â\x82Ã\x83Â\x83Ã\x82Â\x83Ã\
x83Â\x82Ã\x82Â\x82Ã\x83Â\x83Ã\x82Â\x82Ã\x83Â\x82Ã\x82Â\x83Ã\x83Â\x83Ã\x82Â\x83Ã\x83Â\x82
Ã\x82Â\x83Ã\x83Â\x83Ã\x82Â\x82Ã\x83Â\x82Ã\x82Â\x83Ã\x83Â\x83Ã\x82Â\x83Ã\x83Â\x82Ã\
x82Â\x82Ã\x83Â\x83Ã\x82Â\x82Ã\x83Â\x82Ã\x82Â\x83Ã\x83Â\x83Ã\x82Â\x83Ã\x83Â\x82Ã\x82Â\x83
Ã\x83Â\x83Ã\x82Â\x82Ã\x83Â\x82Ã\x82Â\x82Ã\x83Â\x83Ã\x82Â\x83Ã\x83Â\x82Ã\x82Â\x82Ã\
x83Â\x83Ã\x82Â\x82Ã\x83Â\x82Ã\x84Ã\x83Â\x83Ã\x82Â\x83Ã\x83Â\x82Ã\x82Â\x83Ã\x83Â\x83
Ã\x82Â\x82Ã\x83Â\x82Ã\x82Â\x83Ã\x83Â\x83Ã\x82Â\x83Ã\x83Â\x82Ã\x82Â\x82Ã\x83Â\x83Ã\
x82Â\x82Ã\x83Â\x82Ã\x82Â\x83Ã\x83Â\x83Ã\x82Â\x83Ã\x83Â\x82Ã\x82Â\x83Ã\x83Â\x83Ã\x82Â\x82
Ã\x83Â\x82Ã\x82Â\x82Ã\x83Â\x83Ã\x82Â\x83Ã\x83Â\x82Ã\x82Â\x82Ã\x83Â\x83Ã\x82Â\x82Ã\
x83Â\x82Ã\x82Â\x85'], (6, 6, 6))
```

6 is the success return code for the protect operation of each element in the list.

2.4.4.3.19 Mock Example - Protecting Integer Data

This section describes how to use the *protect* API for protecting integer data.

> **Note:**
> The AP Python APIs support integer values only between -2147483648 and 2147483648, both inclusive.

Example

In the following example, *21* is used as the integer data, which is tokenized using the *SUCCESS_INT* data element.

```
from appython import Protector
protector = Protector()
session = protector.create_session("ALL_USER")
output = session.protect(21, "SUCCESS_INT")
print("Protected Data: %s" %output)
```

Result

```
Protected Data: 68
```

2.4.4.3.20 Mock Example - Protecting Integer Data with External Initialization Vector (IV)

This section describes how to use the *protect* API for protecting integer data using external initialization vector (IV).

> **Note:** If you want to pass the external IV as a keyword argument to the *protect* API, then you must pass the external IV as bytes to the API.

Example

In the following example, *21* is used as the integer data, which is tokenized using the *SUCCESS_INT* data element, with the help of external IV *1234* that is passed as bytes.

```
from appython import Protector
protector = Protector()
session = protector.create_session("ALL_USER")
output = session.protect(21, "SUCCESS_INT", external_iv=bytes("1234", encoding="utf-8"))
print("Protected Data: %s" %output)
```

Result

```
Protected Data: 36
```

2.4.4.3.21 Mock Example - Protecting Bulk Integer Data

This section describes how to use the *protect* API for protecting bulk integer data. You can pass bulk integer data as a list or a tuple.

Caution: The individual elements of the list or tuple must be of the same data type.

Note:
The AP Python APIs support integer values only between -2147483648 and 2147483648, both inclusive.

Example

In the following example, *21*, *42*, and *55* integers are stored in a list and used as bulk data, which is protected using the *SUCCESS_INT* data element.

```
from appython import Protector
protector = Protector()
session = protector.create_session("ALL_USER")
data = [21, 42, 55]
p_out = session.protect(data, "SUCCESS_INT")
print("Protected Data: ")
print(p_out)
```

Result

```
Protected Data:
([68, 46, 55], (6, 6, 6))
```

6 is the success return code for the protect operation of each element in the list.

2.4.4.3.22 Mock Example - Protecting Bulk Integer Data with External IV

This section describes how to use the *protect* API for protecting bulk integer data using external IV. You can pass bulk integer data as a list or a tuple.

Caution: The individual elements of the list or tuple must be of the same data type.

Note: If you want to pass the external IV as a keyword argument to the *protect* API, then you must pass the external IV as bytes to the API.

Example

In the following example, *21*, *42*, and *55* integers are stored in a list and used as bulk data, which is protected using the *SUCCESS_INT* data element, with the help of external IV *1234* that is passed as bytes.

```
from appython import Protector
protector = Protector()
session = protector.create_session("ALL_USER")
data = [21, 42, 55]
p_out = session.protect(data, "SUCCESS_INT", external_iv=bytes("1234",
encoding="utf-8"))
print("Protected Data: ")
print(p_out)
```

Result

```
Protected Data:
([36, 13, 99], (6, 6, 6))
```

6 is the success return code for the protect operation of each element in the list.

2.4.4.3.23 Mock Example - Unprotecting Integer Data

This section describes how to use the *unprotect* API for retrieving the original integer data from protected data.

> **Note:**
> The AP Python APIs support integer values only between -2147483648 and 2147483648, both inclusive.

Example

In the following example, the integer data *21* that was protected using the *SUCCESS_INT* data element, is now unprotected using the same data element.

```
from appython import Protector
protector = Protector()
session = protector.create_session("ALL_USER")
output = session.protect(21, "SUCCESS_INT")
print("Protected Data: %s" %output)
org = session.unprotect(output, "SUCCESS_INT")
print("Unprotected Data: %s" %org)
```

Result

```
Protected Data: 68
Unprotected Data: 21
```

2.4.4.3.24 Mock Example - Unprotecting Integer Data with External IV

This section describes how to use the *unprotect* API for retrieving the original integer data from protected data, using external initialization vector (IV).

> **Note:** If you want to pass the external IV as a keyword argument to the *unprotect* API, then you must pass the external IV as bytes to the API.

Example

In the following example, the integer data *21* that was protected using the *SUCCESS_INT* data element and the external IV *1234*, is now unprotected using the same data element and same external IV.

```
from appython import Protector
protector = Protector()
session = protector.create_session("ALL_USER")
output = session.protect(21, "SUCCESS_INT",
         external_iv=bytes("1234", encoding="utf-8"))
print("Protected Data: %s" %output)
org = session.unprotect(output, "SUCCESS_INT",
      external_iv=bytes("1234", encoding="utf-8"))
print("Unprotected Data: %s" %org)
```

Result

```
Protected Data: 36
Unprotected Data: 21
```

2.4.4.3.25 Mock Example - Unprotecting Bulk Integer Data

This section describes how to use the *unprotect* API for retrieving the original bulk integer data from protected data.

Example

In the following example, *21*, *42*, and *55* integers are stored in a list and used as bulk data, which is protected using the *SUCCESS_INT* data element. The bulk integer data is then unprotected using the same data element.

```
from appython import Protector
protector = Protector()
session = protector.create_session("ALL_USER")
data = [21, 42, 55]
p_out = session.protect(data, "SUCCESS_INT")
print("Protected Data: ")
print(p_out)
out = session.unprotect(p_out[0], "SUCCESS_INT")
print("Unprotected Data: ")
print(out)
```

Result

```
Protected Data:
([68, 46, 55], (6, 6, 6))
Unprotected Data:
([21, 42, 55], (8, 8, 8))
```

6 is the success return code for the protect operation of each element in the list.

2.4.4.3.26 Mock Example - Unprotecting Bulk Integer Data with External IV

This section describes how to use the *unprotect* API for retrieving the original bulk integer data from protected data using external IV.

> **Note:** If you want to pass the external IV as a keyword argument to the *unprotect* API, then you must pass the external IV as bytes to the API.

Example

In the following example, *21*, *42*, and *55* integers are stored in a list and used as bulk data, which is protected using the *SUCCESS_INT* data element and external IV *1234*. The bulk integer data is then unprotected using the same data element and external IV.

```
from appython import Protector
protector = Protector()
session = protector.create_session("ALL_USER")
data = [21, 42, 55]
p_out = session.protect(data, "SUCCESS_INT", external_iv=bytes("1234",
encoding="utf-8"))
print("Protected Data: ")
print(p_out)
out = session.unprotect(p_out[0], "SUCCESS_INT", external_iv=bytes("1234",
encoding="utf-8"))
print("Unprotected Data: ")
print(out)
```

Result

```
Protected Data:
([36, 13, 99], (6, 6, 6))
Unprotected Data:
([21, 42, 55], (8, 8, 8))
```

6 is the success return code for the protect operation of each element in the list.

8 is the success return code for the unprotect operation of each element in the list.

2.4.4.3.27 Mock Example - Reprotecting Integer Data

This section describes how to use the *reprotect* API for reprotecting integer data.

Warning: If you are retokenizing the data using the *reprotect* API, then the old data element and the new data element must have the same tokenization type. For example, if you have used Integer data element to protect the data, then you must use only Integer data element to reprotect the data.

Note:
The AP Python APIs support integer values only between -2147483648 and 2147483648, both inclusive.

Example

In the following example, *21* is used as the input integer data, which is first protected using the *SUCCESS_INT* data element.

The tokenized input data, the old data element *SUCCESS_INT*, and a new data element *SUCCESS_REPROTECT_INT* are then passed as inputs to the *reprotect* API. The *reprotect* API first unprotects the protected input data using the old data element, and then reprotects it using the new data element, as part of a single reprotect operation.

```
from appython import Protector
protector = Protector()
session = protector.create_session("ALL_USER")
output = session.protect(21, "SUCCESS_INT")
print("Protected Data: %s" %output)
r_out = session.reprotect(output, "SUCCESS_INT", "SUCCESS_REPROTECT_INT")
print("Reprotected Data: %s" %r_out)
```

Result

```
Protected Data: 68
Reprotected Data: 69
```

2.4.4.3.28 Mock Example - Reprotecting Integer Data with External IV

This section describes how to use the *reprotect* API for reprotecting integer data using external IV.

Warning: If you are retokenizing the data using the *reprotect* API, then the old data element and the new data element must have the same tokenization type. For example, if you have used Integer data element to protect the data, then you must use only Integer data element to reprotect the data.

Note: If you want to pass the external IV as a keyword argument to the *reprotect* API, then you must pass the external IV as bytes to the API.

Example

In the following example, *21* is used as the input integer data, which is first tokenized using the *SUCCESS_INT* data element, with the help of external IV *1234* that is passed as bytes.

The protected input data, the old data element *SUCCESS_INT*, a new data element *SUCCESS_REPROTECT_INT*, the old external IV *1234* in bytes, and a new external IV *123456* in bytes are then passed as inputs to the *reprotect* API. As part of a single reprotect operation, the *reprotect* API first unprotects the protected input data using the given data element and old external IV, and then reprotects it using the new data element and new external IV.

```
from appython import Protector
protector = Protector()
session = protector.create_session("ALL_USER")
p_out = session.protect(21, "SUCCESS_INT",
        external_iv=bytes("1234", encoding="utf-8"))
print("Protected Data: %s" %p_out)
r_out = session.reprotect(p_out, "SUCCESS_INT", "SUCCESS_REPROTECT_INT",
        old_external_iv=bytes("1234", encoding="utf-8"),
new_external_iv=bytes("123456", encoding="utf-8"))
print("Reprotected Data: %s" %r_out)
```

Result

```
Protected Data: 36
Reprotected Data: 14
```

2.4.4.3.29 Mock Example - Reprotecting Bulk Integer Data

This section describes how to use the *reprotect* API for reprotecting bulk integer data. You can pass bulk integer data as a list or a tuple.

Caution: The individual elements of the list or tuple must be of the same data type.

Warning: If you are retokenizing the data using the *reprotect* API, then the old data element and the new data element must have the same tokenization type. For example, if you have used Integer data element to protect the data, then you must use only Integer data element to reprotect the data.

Note:

The AP Python APIs support integer values only between -2147483648 and 2147483648, both inclusive.

Example

In the following example, *21*, *42*, and *55* integers are stored in a list and used as bulk data, which is protected using the *SUCCESS_INT* data element.

The protected input data, the old data element *SUCCESS_INT*, and a new data element *SUCCESS_REPROTECT_INT* are then passed as inputs to the *reprotect* API. The *reprotect* API first unprotects the protected input data using the old data element and then reprotects it using the new data element, as part of a single reprotect operation.

```
from appython import Protector
protector = Protector()
session = protector.create_session("ALL_USER")
data = [21, 42, 55]
p_out = session.protect(data, "SUCCESS_INT")
print("Protected Data: ")
print(p_out)
r_out = session.reprotect(p_out[0], "SUCCESS_INT", "SUCCESS_REPROTECT_INT")
print("Reprotected Data: ")
print(r_out)
```

Result

```
Protected Data:
([68, 46, 55], (6, 6, 6))
Reprotected Data:
([69, 86, 22], (6, 6, 6))
```

6 is the success return code for the protect operation of each element in the list.

2.4.4.3.30 Mock Example - Reprotecting Bulk Integer Data with External IV

This section describes how to use the *reprotect* API for rerotecting bulk integer data using external IV. You can pass bulk integer data as a list or a tuple.

Caution: The individual elements of the list or tuple must be of the same data type.

Warning: If you are retokenizing the data using the *reprotect* API, then the old data element and the new data element must have the same tokenization type. For example, if you have used Integer data element to protect the data, then you must use only Integer data element to reprotect the data.

Note: If you want to pass the external IV as a keyword argument to the *reprotect* API, then you must pass the external IV as bytes to the API.

Example

In the following example, *21*, *42*, and *55* integers are stored in a list and used as bulk data, which is protected using the *SUCCESS_INT* data element, with the help of an external IV *123* that is passed as bytes.

The tokenized input data, the old data element *SUCCESS_INT*, a new data element *SUCCESS_REPROTECT_INT*, the old external IV *123* in bytes, and a new external IV *1234* in bytes are then passed as inputs to the *reprotect* API. As part of a single reprotect operation, the *reprotect* API first unprotects the protected input data using the given data element and old external IV, and then reprotects it using the new data element and new external IV.

```
from appython import Protector
protector = Protector()
session = protector.create_session("ALL_USER")
data = [21, 42, 55]
p_out = session.protect(data, "SUCCESS_INT", external_iv=bytes("1234",
encoding="utf-8"))
print("Protected Data: ")
print(p_out)
r_out = session.reprotect(p_out[0], "SUCCESS_INT", "SUCCESS_REPROTECT_INT",
        old_external_iv=bytes("123", encoding="utf-8"), new_external_iv=bytes("1234",
encoding="utf-8"))
print("Reprotected Data: ")
print(r_out)
```

Result

```
Protected Data:
([36, 13, 99], (6, 6, 6))
Reprotected Data:
([24, 72, 33], (6, 6, 6))
```

6 is the success return code for the protect operation of each element in the list.

2.4.4.3.31 Mock Example - Protecting Long Data

This section describes how to use the *protect* API for protecting long data.

Example

In the following example, *1376235139103947* is used as the long data, which is protected using the *SUCCESS_LONG* data element.

```
from appython import Protector
protector = Protector()
session = protector.create_session("ALL_USER")
output = session.protect(1376235139103947, "SUCCESS_LONG")
print("Protected Data: %s" %output)
```

Result

```
Protected Data: 8632961867806753
```

2.4.4.3.32 Mock Example - Protecting Long Data with External IV

This section describes how to use the *protect* API for protecting long data using external IV.

> **Note:** If you want to pass the external IV as a keyword argument to the *protect* API, then you must pass the external IV as bytes to the API.

Example

In the following example, *1376235139103947* is used as the long data, which is protected using the *SUCCESS_LONG* data element, with the help of external IV *1234* that is passed as bytes.

```
from appython import Protector
protector = Protector()
session = protector.create_session("ALL_USER")
output = session.protect(1376235139103947, "SUCCESS_LONG",
         external_iv=bytes("1234", encoding="utf-8"))
print("Protected Data: %s" %output)
```

Result

```
Protected Data: 6278329624602417
```

2.4.4.3.33 Mock Example - Protecting Bulk Long Data

This section describes how to use the *protect* API for protecting bulk long data. You can pass bulk long data as a list or a tuple.

> **Caution:** The individual elements of the list or tuple must be of the same data type.

Example

In the following example, *1376235139103947*, *2396235839173981*, and *9371234126176985* long data are stored in a list and used as bulk data, which is protected using the *SUCCESS_LONG* data element.

```
from appython import Protector
protector = Protector()
session = protector.create_session("ALL_USER")
data = [1376235139103947, 2396235839173981, 9371234126176985]
p_out = session.protect(data, "SUCCESS_LONG")
print("Protected Data: ")
print(p_out)
```

Result

```
Protected Data:
([8632961867806753, 9672961467836748, 7638965892832741], (6, 6, 6))
```

6 is the success return code for the protect operation of each element in the list.

2.4.4.3.34 Mock Example - Protecting Bulk Long Data with External IV

This section describes how to use the *protect* API for protecting bulk long data using external IV. You can pass bulk long data as a list or a tuple.

Caution: The individual elements of the list or tuple must be of the same data type.

Note: If you want to pass the external IV as a keyword argument to the *protect* API, then you must pass the external IV as bytes to the API.

Example

In the following example, *1376235139103947*, *2396235839173981*, and *9371234126176985* long data are stored in a list and used as bulk data, which is protected using the *SUCCESS_LONG* data element, with the help of external IV *1234* that is passed as bytes.

```
from appython import Protector
protector = Protector()
session = protector.create_session("ALL_USER")
data = [1376235139103947, 2396235839173981, 9371234126176985]
p_out = session.protect(data, "SUCCESS_LONG", external_iv=bytes("1234",
encoding="utf-8"))
print("Protected Data: ")
print(p_out)
```

Result

```
Protected Data:
([6278329624602417, 3248329524672456, 4276321638678459], (6, 6, 6))
```

6 is the success return code for the protect operation of each element in the list.

2.4.4.3.35 Mock Example - Unprotecting Long Data

This section describes how to use the *unprotect* API for retrieving the original long data from protected data.

Example

In the following example, the long data *1376235139103947* that was tokenized using the *SUCCESS_LONG* data element, is now unprotected using the same data element.

```
from appython import Protector
protector = Protector()
session = protector.create_session("ALL_USER")
output = session.protect(1376235139103947, "SUCCESS_LONG")
print("Protected Data: %s" %output)
org = session.unprotect(output, "SUCCESS_LONG")
print("Unprotected Data: %s" %org)
```

Result

```
Protected Data: 8632961867806753
Unprotected Data: 1376235139103947
```

2.4.4.3.36 Mock Example - Unprotecting Long Data with External IV

This section describes how to use the *unprotect* API for retrieving the original long data from protected data, using external initialization vector (IV).

> **Note:** If you want to pass the external IV as a keyword argument to the *unprotect* API, then you must pass the external IV as bytes to the API.

Example

In the following example, the long data *1376235139103947* that was protected using the *SUCCESS_LONG* data element and the external IV *1234* is now unprotected using the same data element and external IV.

```
from appython import Protector
protector = Protector()
session = protector.create_session("ALL_USER")
output = session.protect(1376235139103947, "SUCCESS_LONG",
         external_iv=bytes("1234", encoding="utf-8"))
print("Protected Data: %s" %output)
org = session.unprotect(output, "SUCCESS_LONG",
      external_iv=bytes("1234", encoding="utf-8"))
print("Unprotected Data: %s" %org)
```

Result

```
Protected Data: 6278329624602417
Unprotected Data: 1376235139103947
```

2.4.4.3.37 Mock Example - Unprotecting Bulk Long Data

This section describes how to use the *unprotect* API for retrieving the original bulk long data from protected data.

Example

In the following example, *1376235139103947*, *2396235839173981*, and *9371234126176985* long data are stored in a list and used as bulk data, which is protected using the *SUCCESS_LONG* data element. The bulk long data is then unprotected using the same data element.

```
from appython import Protector
protector = Protector()
session = protector.create_session("ALL_USER")
data = [1376235139103947, 2396235839173981, 9371234126176985]
p_out = session.protect(data, "SUCCESS_LONG")
print("Protected Data: ")
print(p_out)
out = session.unprotect(p_out[0], "SUCCESS_LONG")
print("Unprotected Data: ")
print(out)
```

Result

```
Protected Data:
([8632961867806753, 9672961467836748, 7638965892832741], (6, 6, 6))
Unprotected Data:
([1376235139103947, 2396235839173981, 9371234126176985], (8, 8, 8))
```

6 is the success return code for the protect operation of each element in the list.

8 is the success return code for the unprotect operation of each element in the list.

2.4.4.3.38 Mock Example - Unprotecting Bulk Long Data with External IV

This section describes how to use the *unprotect* API for retrieving the original bulk long data from protected data using external IV.

> **Note:** If you want to pass the external IV as a keyword argument to the *unprotect* API, then you must pass the external IV as bytes to the API.

Example

In the following example, *1376235139103947*, *2396235839173981*, and *9371234126176985* long data are stored in a list and used as bulk data, which is protected using the *SUCCESS_LONG* data element and external IV *1234*. The bulk long data is then unprotected using the same data element and same external IV.

```
from appython import Protector
protector = Protector()
session = protector.create_session("ALL_USER")
data = [1376235139103947, 2396235839173981, 9371234126176985]
p_out = session.protect(data, "SUCCESS_LONG", external_iv=bytes("1234",
encoding="utf-8"))
print("Protected Data: ")
print(p_out)
out = session.unprotect(p_out[0], "SUCCESS_LONG", external_iv=bytes("1234",
encoding="utf-8"))
print("Unprotected Data: ")
print(out)
```

Result

```
Protected Data:
([6278329624602417, 3248329524672456, 4276321638678459], (6, 6, 6))
Unprotected Data:
([1376235139103947, 2396235839173981, 9371234126176985], (8, 8, 8))
```

6 is the success return code for the protect operation of each element in the list.

8 is the success return code for the unprotect operation of each element in the list.

2.4.4.3.39 Mock Example - Reprotecting Long Data

This section describes how to use the *reprotect* API for reprotecting long data.

> **Warning:** If you are retokenizing the data using the *reprotect* API, then the old data element and the new data element must have the same tokenization type. For example, if you have used Integer data element to protect the data, then you must use only Integer data element to reprotect the data.

Example

In the following example, *1376235139103947* is used as the input long data, which is first protected using the *SUCCESS_LONG* data element.

The protected input data, the old data element *SUCCESS_LONG*, and a new data element *SUCCESS_REPROTECT_LONG* are then passed as inputs to the *reprotect* API. The *reprotect* API first unprotects the protected input data using the old data element and then reprotects it using the new data element, as part of a single reprotect operation.

```
from appython import Protector
protector = Protector()
session = protector.create_session("ALL_USER")
output = session.protect(1376235139103947, "SUCCESS_LONG")
```

```
print("Protected Data: %s" %output)
r_out = session.reprotect(output, "SUCCESS_LONG", "SUCCESS_REPROTECT_LONG")
print("Reprotected Data: %s" %r_out)
```

Result

```
Protected Data: 8632961867806753
```

2.4.4.3.40 Mock Example - Reprotecting Long Data with External IV

This section describes how to use the *reprotect* API for reprotecting long data using external IV.

Warning: If you are retokenizing the data using the *reprotect* API, then the old data element and the new data element must have the same tokenization type. For example, if you have used Integer data element to protect the data, then you must use only Integer data element to reprotect the data.

Note: If you want to pass the external IV as a keyword argument to the *reprotect* API, then you must pass the external IV as bytes to the API.

Example

In the following example, *1376235139103947* is used as the input long data, which is first protected using the *SUCCESS_LONG* data element, with the help of external IV *1234* that is passed as bytes.

The protected input data, the old data element *SUCCESS_LONG*, a new data element *SUCCESS_REPROTECT_LONG*, the old external IV *1234* in bytes, and a new external IV *123456* in bytes are then passed as inputs to the *reprotect* API. As part of a single reprotect operation, the *reprotect* API first unprotects the protected input data using the given data element and old external IV, and then reprotects it using the new data element and new external IV.

```
from appython import Protector
protector = Protector()
session = protector.create_session("ALL_USER")
p_out = session.protect(1376235139103947, "SUCCESS_LONG",
        external_iv=bytes("1234", encoding="utf-8"))
print("Protected Data: %s" %p_out)
r_out = session.reprotect(p_out, "SUCCESS_LONG", "SUCCESS_REPROTECT_LONG",
        old_external_iv=bytes("1234", encoding="utf-8"),
new_external_iv=bytes("123456", encoding="utf-8"))
print("Reprotected Data: %s" %r_out)
```

Result

```
Protected Data: 6278329624602417
Reprotected Data: 4563152458405896
```

2.4.4.3.41 Mock Example - Reprotecting Bulk Long Data

This section describes how to use the *reprotect* API for reprotecting bulk long data. You can pass bulk long data as a list or a tuple.

Caution: The individual elements of the list or tuple must be of the same data type.

Warning: If you are retokenizing the data using the *reprotect* API, then the old data element and the new data element must have the same tokenization type. For example, if you have used Integer data element to protect the data, then you must use only Integer data element to reprotect the data.

Example

In the following example, *1376235139103947*, *2396235839173981*, and *9371234126176985* long data are stored in a list and used as bulk data, which is protected using the *SUCCESS_LONG* data element.

The tokenized input data, the old data element *SUCCESS_LONG*, and a new data element *SUCCESS_REPROTECT_LONG* are then passed as inputs to the *reprotect* API. The *reprotect* API first unprotects the protected input data using the old data element and then reprotects it using the new data element, as part of a single reprotect operation.

```
from appython import Protector
protector = Protector()
session = protector.create_session("ALL_USER")
data = [1376235139103947, 2396235839173981, 9371234126176985]
p_out = session.protect(data, "SUCCESS_LONG")
print("Protected Data: ")
print(p_out)
r_out = session.reprotect(p_out[0], "SUCCESS_LONG", "SUCCESS_REPROTECT_LONG")
print("Reprotected Data: ")
print(r_out)
```

Result

```
Protected Data:
([8632961867806753, 9672961467836748, 7638965892832741], (6, 6, 6))
Reprotected Data:
([4213926425402581, 9253926725412574, 5214928493413576], (6, 6, 6))
```

6 is the success return code for the protect operation of each element in the list.

2.4.4.3.42 Mock Example - Reprotecting Bulk Long Data with External IV

This section describes how to use the *reprotect* API for reprotecting bulk long data using external IV. You can pass bulk long data as a list or a tuple.

Caution: The individual elements of the list or tuple must be of the same data type.

Warning: If you are retokenizing the data using the *reprotect* API, then the old data element and the new data element must have the same tokenization type. For example, if you have used Integer data element to protect the data, then you must use only Integer data element to reprotect the data.

Note: If you want to pass the external IV as a keyword argument to the *reprotect* API, then you must pass the external IV as bytes to the API.

Example

In the following example, *1376235139103947*, *2396235839173981*, and *9371234126176985* long data are stored in a list and used as bulk data, which is protected using the *SUCCESS_LONG* data element, with the help of an external IV *1234* that is passed as bytes.

The protected input data, the old data element *SUCCESS_LONG*, a new data element *SUCCESS_REPROTECT_LONG*, the old external IV *1234* in bytes, and a new external IV *123456* in bytes are then passed as inputs to the *reprotect* API. As part of a single reprotect operation, the *reprotect* API first unprotects the protected input data using the given data element and old external IV, and then reprotects it using the new data element and new external IV.

```
from appython import Protector
protector = Protector()
session = protector.create_session("ALL_USER")
data = [1376235139103947, 2396235839173981, 9371234126176985]
p_out = session.protect(data, "SUCCESS_LONG", external_iv=bytes("1234",
```

```
encoding="utf-8"))
print("Protected Data: ")
print(p_out)
r_out = session.reprotect(p_out[0], "SUCCESS_LONG", "SUCCESS_REPROTECT_LONG",
        old_external_iv=bytes("1234", encoding="utf-8"),
new_external_iv=bytes("123456", encoding="utf-8"))
print("Reprotected Data: ")
print(r_out)
```

Result

```
Protected Data:
([6278329624602417, 3248329524672456, 4276321638678459], (6, 6, 6))
Reprotected Data:
([4563152458405896, 1583152758465874, 8564159413463872], (6, 6, 6))
```

6 is the success return code for the protect operation of each element in the list.

2.4.4.3.43 Mock Example - Protecting Float Data

This section describes how to use the *protect* API for protecting float data using a No Encryption data element. You can use this API for access control and auditing.

Example

In the following example, *22.5* is used as the float data, which is protected using the *SUCCESS_FLOAT* data element.

```
from appython import Protector
protector = Protector()
session = protector.create_session("ALL_USER")
output = session.protect(22.5, "SUCCESS_FLOAT")
print("Protected Data: %s" %output)
```

Result

```
Protected Data: 22.5
```

As we are using a No Encryption data element to protect the data, the protected output data is the same as the input data.

2.4.4.3.44 Mock Example - Protecting Bulk Float Data

This section describes how to use the *protect* API for protecting bulk float data using a No Encryption data element. You can pass bulk float data as a list or a tuple. You can use this API for access control and auditing.

> **Caution:** The individual elements of the list or tuple must be of the same data type.

Example

In the following example, *22.5*, *48.93*, and *94.14* float data are stored in a list and used as bulk data, which is protected using the *SUCCESS_FLOAT* data element.

```
from appython import Protector
protector = Protector()
session = protector.create_session("ALL_USER")
data = [22.5, 48.93, 94.31]
p_out = session.protect(data, "SUCCESS_FLOAT")
print("Protected Data: ")
print(p_out)
```

Result

```
Protected Data:
([22.5, 48.93, 94.31], (6, 6, 6))
```

6 is the success return code for the protect operation of each element in the list.

As we are using a No Encryption data element to protect the data, the protected output data is the same as the input data.

2.4.4.3.45 Mock Example - Unprotecting Float Data

This section describes how to use the *unprotect* API for unprotecting float data using a No Encryption data element. You can use this API for access control and auditing.

Example

In the following example, the long data *22.5* that was protected using the *SUCCESS_FLOAT* data element, is now unprotected using the same data element.

```
from appython import Protector
protector = Protector()
session = protector.create_session("ALL_USER")
output = session.protect(22.5, "SUCCESS_FLOAT")
print("Protected Data: %s" %output)
org = session.unprotect(output, "SUCCESS_FLOAT")
print("Unprotected Data: %s" %org)
```

Result

```
Protected Data: 22.5
Unprotected Data: 22.5
```

The input data, the protected output data, and the unprotected data are the same, as we are using a No Encryption data element to protect and unprotect the data.

2.4.4.3.46 Mock Example - Unprotecting Bulk Float Data

This section describes how to use the *unprotect* API for unprotecting bulk float data using a No Encryption data element. You can use this API for access control and auditing.

Example

In the following example, *22.5*, *48.93*, and *94.14* float data are stored in a list and used as bulk data, which is protected using the *SUCCESS_FLOAT* data element. The bulk float data is then unprotected using the same data element.

```
from appython import Protector
protector = Protector()
session = protector.create_session("ALL_USER")
data = [22.5, 48.93, 94.31]
p_out = session.protect(data, "SUCCESS_FLOAT")
print("Protected Data: ")
print(p_out)
out = session.unprotect(p_out[0], "SUCCESS_FLOAT")
print("Unprotected Data: ")
print(out)
```

Result

```
Protected Data:
([22.5, 48.93, 94.31], (6, 6, 6))
Unprotected Data:
([22.5, 48.93, 94.31], (8, 8, 8))
```

6 is the success return code for the protect operation of each element in the list.

8 is the success return code for the unprotect operation of each element in the list.

The input data, the protected output data, and the unprotected data are the same, as we are using a No Encryption data element to protect and unprotect the data.

2.4.4.3.47 Mock Example - Reprotecting Float Data

This section describes how to use the *reprotect* API for reprotecting float data using a No Encryption data element. You can use this API for access control and auditing.

Warning: If you are reprotecting the data using the *reprotect* API, then the old data element and the new data element must have the same tokenization type. For example, if you have used No Encryption data element to protect the data, then you must use only No Encryption data element to reprotect the data.

Example

In the following example, *22.5* is used as the input float data, which is first protected using the *SUCCESS_FLOAT* data element.

The protected input data, the old data element *SUCCESS_FLOAT*, and a new data element *SUCCESS_REPROTECT_FLOAT* are then passed as inputs to the *reprotect* API. The *reprotect* API first unprotects the protected input data using the old data element and then reprotects it using the new data element, as part of a single reprotect operation.

```
from appython import Protector
protector = Protector()
session = protector.create_session("ALL_USER")
output = session.protect(22.5, "SUCCESS_FLOAT")
print("Protected Data: %s" %output)
r_out = session.reprotect(output, "SUCCESS_FLOAT", "SUCCESS_REPROTECT_FLOAT")
print("Reprotected Data: %s" %r_out)
```

Result

```
Protected Data: 22.5
Reprotected Data: 22.5
```

As we are using a No Encryption data element to protect and reprotect the data, the reprotected output data is the same as the protected data.

2.4.4.3.48 Mock Example - Reprotecting Bulk Float Data

This section describes how to use the *reprotect* API for reprotecting bulk float data using a No Encryption data element. You can pass bulk long data as a list or a tuple. You can use this API for access control and auditing.

Caution: The individual elements of the list or tuple must be of the same data type.

Warning: If you are reprotecting the data using the *reprotect* API, then the old data element and the new data element must have the same tokenization type. For example, if you have used No Encryption data element to protect the data, then you must use only No Encryption data element to reprotect the data.

Example

In the following example, *22.5*, *48.93*, and *94.14* float data are stored in a list and used as bulk data, which is protected using the *SUCCESS_FLOAT* data element.

The tokenized input data, the old data element *SUCCESS_FLOAT*, and a new data element *SUCCESS_REPROTECT_FLOAT* are then passed as inputs to the *reprotect* API. The *reprotect* API first unprotects the protected input data using the old data element and then reprotects it using the new data element, as part of a single reprotect operation.

```
from appython import Protector
protector = Protector()
session = protector.create_session("ALL_USER")
data = [22.5, 48.93, 94.31]
p_out = session.protect(data, "SUCCESS_FLOAT")
print("Protected Data: ")
print(p_out)
r_out = session.reprotect(p_out[0], "SUCCESS_FLOAT", "SUCCESS_REPROTECT_FLOAT")
print("Reprotected Data: ")
print(r_out)
```

Result

```
Protected Data:
([22.5, 48.93, 94.31], (6, 6, 6))
Reprotected Data:
([22.5, 48.93, 94.31], (6, 6, 6))
```

6 is the success return code for the protect operation of each element in the list.

As we are using a No Encryption data element to protect and reprotect the data, the reprotected output data is the same as the protected data.

2.4.4.3.49 Mock Example - Protecting Bytes Data

This section describes how to use the *protect* API for protecting bytes data.

Example

In the following example, *"Protegrity1"* string is first converted to bytes using the Python *bytes()* method. The bytes data is then protected using the *SUCCESS_BYTE* data element.

```
from appython import Protector
protector = Protector()
session = protector.create_session("ALL_USER")
data=bytes("Protegrity1", encoding="utf-8")
p_out = session.protect(data, "SUCCESS_BYTE")
print("Protected Data: %s" %p_out)
```

Result

```
Protected Data: b'nLiNJRL7N2P'
```

2.4.4.3.50 Mock Example - Protecting Bytes Data with External IV

This section describes how to use the *protect* API for protecting bytes data using external IV.

Example

In the following example, *"Protegrity1"* string is first converted to bytes using the Python *bytes()* method. The bytes data is then protected using the *SUCCESS_BYTE* data element, with the help of external IV *1234* that is passed as bytes.

```
from appython import Protector
protector = Protector()
session = protector.create_session("ALL_USER")
data=bytes("Protegrity1", encoding="utf-8")
output = session.protect(data, "SUCCESS_BYTE",
        external_iv=bytes("1234", encoding="utf-8"))
print("Protected Data: %s" %output)
```

Result

```
Protected Data: b'Ho9bgXoebxa'
```

2.4.4.3.51 Mock Example - Protecting Bulk Bytes Data

This section describes how to use the *protect* API for protecting bulk bytes data. You can pass bulk bytes data as a list or a tuple.

> **Caution:** The individual elements of the list or tuple must be of the same data type.

Example

In the following example, *protegrity1234*, *Protegrity1*, and *Protegrity56* strings are first converted to bytes using the Python *bytes()* method. The converted bytes are then stored in a list and used as bulk data, which is protected using the *SUCCESS_BYTE* data element.

```
from appython import Protector
protector = Protector()
session = protector.create_session("ALL_USER")
data = [bytes("protegrity1234"), bytes("Protegrity1"), bytes("Protegrity56")]
p_out = session.protect(data, "SUCCESS_BYTE")
print("Protected Data: ")
print(p_out)
```

Result

```
Protected Data:
([b'pJPqrjJEqLXHaO', b'6JPqrjJEqLX', b'6JPqrjJEqLl5'], (6, 6, 6))
```

6 is the success return code for the protect operation of each element in the list.

2.4.4.3.52 Mock Example - Protecting Bulk Bytes Data with External IV

This section desribes how to use the *protect* API for protecting bulk bytes data using external IV. You can pass bulk bytes data as a list or a tuple.

Example

In the following example, *protegrity1234*, *Protegrity1*, and *Protegrity56* strings are first converted to bytes using the Python *bytes()* method. The converted bytes are then stored in a list and used as bulk data, which is protected using the *SUCCESS_BYTE* data element, with the help of external IV *1234* that is passed as bytes.

```
from appython import Protector
protector = Protector()
session = protector.create_session("ALL_USER")
data = [bytes("protegrity1234", encoding="utf-8"), bytes("Protegrity1",
encoding="utf-8"), bytes("Protegrity56", encoding="utf-8")]
p_out = session.protect(data, "SUCCESS_BYTE",
        external_iv=bytes("1234", encoding="utf-8"))
```

```
print("Protected Data: ")
print(p_out)
```

Result

```
Protected Data:
([b'fo9bgXoebxaCTN', b'Ho9bgXoebxa', b'Ho9bgXoebx2q'], (6, 6, 6))
```

6 is the success return code for the protect operation of each element in the list.

2.4.4.3.53 Mock Example - Unprotecting Bytes Data

This section describes how to use the *unprotect* API for retrieving the original bytes data from protected data.

Example

In the following example, the bytes data *b'Protegrity1'* that was protected using the *SUCCESS_BYTE* data element, is now unprotected using the same data element.

```
from appython import Protector
protector = Protector()
session = protector.create_session("ALL_USER")
data=bytes("Protegrity1", encoding="utf-8")
p_out = session.protect(data, "SUCCESS_BYTE")
print("Protected Data: %s" %p_out)
org = session.unprotect(p_out, "SUCCESS_BYTE")
print("Unprotected Data: %s" %org)
```

Result

```
Protected Data: b'6JPqrjJEqLX'
Unprotected Data: b'Protegrity1'
```

2.4.4.3.54 Mock Example - Unprotecting Bytes Data with External IV

This section describes how to use the *unprotect* API for retrieving the original bytes data from protected data, using external initialization vector (IV).

Example

In the following example, the bytes data *b'Protegrity1'* that was protected using the *SUCCESS_BYTE* data element and the external IV *1234* is now unprotected using the same data element and same external IV.

```
from appython import Protector
protector = Protector()
session = protector.create_session("ALL_USER")
data=bytes("Protegrity1", encoding="utf-8")
p_out = session.protect(data, "SUCCESS_BYTE", external_iv=bytes("1234",
encoding="utf-8"))
print("Protected Data:", p_out)
org = session.unprotect(p_out, "SUCCESS_BYTE",  external_iv=bytes("1234",
encoding="utf-8"))
print("Unprotected Data:", org)
```

Result

```
Protected Data: b'Ho9bgXoebxa'
Unprotected Data: b'Protegrity1'
```

2.4.4.3.55 Mock Example - Unprotecting Bulk Bytes Data

This section describes how to use the *unprotect* API for retrieving the original bulk bytes data from protected data.

Example

In the following example, *protegrity1234*, *Protegrity1*, and *Protegrity56* strings are first converted to bytes using the Python *bytes()* method. The converted bytes are then stored in a list and used as bulk data, which is protected using the *SUCCESS_BYTE* data element. The bulk bytes data is then unprotected using the same data element.

```
from appython import Protector
protector = Protector()
session = protector.create_session("ALL_USER")
data = [bytes("protegrity1234"), bytes("Protegrity1"), bytes("Protegrity56")]
p_out = session.protect(data, "SUCCESS_BYTE")
print("Protected Data: ")
print(p_out)
org = session.unprotect(p_out[0], "SUCCESS_BYTE")
print("Unprotected Data: ")
print(org)
```

Result

```
Protected Data:
([b'pJPqrjJEqLXHaO', b'6JPqrjJEqLX', b'6JPqrjJEqLl5'], (6, 6, 6))
Unprotected Data:
([b'protegrity1234', b'Protegrity1', b'Protegrity56'], (8, 8, 8))
```

6 is the success return code for the protect operation of each element in the list.

8 is the success return code for the unprotect operation of each element in the list.

2.4.4.3.56 Mock Example - Unprotecting Bulk Bytes Data with External IV

This section describes how to use the *unprotect* API for retrieving the original bulk bytes data from protected data using external IV.

Example

In the following example, *protegrity1234*, *Protegrity1*, and *Protegrity56* strings are first converted to bytes using the Python *bytes()* method. The converted bytes are then stored in a list and used as bulk data, which is protected using the *SUCCESS_BYTE* data element, with the help of external IV *1234* that is passed as bytes. The bulk bytes data is then unprotected using the same data element and same external IV.

```
from appython import Protector
protector = Protector()
session = protector.create_session("ALL_USER")
data = [bytes("protegrity1234", encoding="utf-8"), bytes("Protegrity1",
encoding="utf-8"), bytes("Protegrity56", encoding="utf-8")]
p_out = session.protect(data, "SUCCESS_BYTE",
        external_iv=bytes("1234", encoding="utf-8"))
print("Protected Data: ")
print(p_out)
org = session.unprotect(p_out[0], "SUCCESS_BYTE",
      external_iv=bytes("1234", encoding="utf-8"))
print("Unprotected Data: ")
print(org)
```

Result

```
Protected Data:
([b'fo9bgXoebxaCTN', b'Ho9bgXoebxa', b'Ho9bgXoebx2q'], (6, 6, 6))
Unprotected Data:
([b'protegrity1234', b'Protegrity1', b'Protegrity56'], (8, 8, 8))
```

6 is the success return code for the protect operation of each element in the list.

8 is the success return code for the unprotect operation of each element in the list.

2.4.4.3.57 Mock Example - Re-encrypting Bytes Data

This section describes how to use the *reprotect* API for re-encrypting bytes data.

Warning: If you are using the *reprotect* API, then the old data element and the new data element must be of the same protection method. For example, if you have used AES256 data element to protect the data, then you must use only AES256 data element to reprotect the data.

Example

In the following example, *Protegrity1* string is first converted to bytes using the Python *bytes()* method. The bytes data is then encrypted using the *SUCCESS_BYTE* data element. Therefore, the *encrypt_to* parameter is passed as a keyword argument and its value is set to *bytes*.

The protected input data, the old data element *SUCCESS_BYTE*, and a new data element *SUCCESS_REPROTECT_BYTE* are then passed as inputs to the *reprotect* API. The *reprotect* API first decrypts the protected input data using the old data element and then re-encrypts it using the new data element, as part of a single reprotect operation. Therefore, the *encrypt_to* parameter is passed as a keyword argument and its value is set to *bytes*.

```
from appython import Protector
protector = Protector()
session = protector.create_session("ALL_USER")
data=bytes("Protegrity1", encoding="utf-8")
p_out = session.protect(data, "SUCCESS_BYTE", encrypt_to=bytes)
print("Encrypted Data: %s" %p_out)
r_out = session.reprotect(p_out, "SUCCESS_BYTE", "SUCCESS_REPROTECT_BYTE",
encrypt_to=bytes)
print("Re-encrypted Data: %s" %r_out)
```

Result

```
Encrypted Data: b'6JPqrjJEqLX'
Re-encrypted Data: b'JQbePhQ2eGC'
```

2.4.4.3.58 Mock Example - Reprotecting Bytes Data with External IV

This section describes how to use the *reprotect* API for reprotecting bytes data using external IV.

Warning: If you are retokenizing the data using the *reprotect* API, then the old data element and the new data element must have the same tokenization type. For example, if you have used Alphanumeric data element to protect the data, then you must use only Alphanumeric data element to reprotect the data.

Example

In the following example, *Protegrity1* string is first converted to bytes using the Python *bytes()* method. The bytes data is then protected using the *SUCCESS_BYTE* data element, with the help of external IV *1234* that is passed as bytes.

The protected input data, the old data element *SUCCESS_BYTE*, a new data element *SUCCESS_REPROTECT_BYTE*, the old external IV *1234* in bytes, and a new external IV *123456* in bytes are then passed as inputs to the *reprotect* API. As part of a single reprotect operation, the *reprotect* API first unprotects the protected input data using the given data element and old external IV, and then reprotects it using the new data element and new external IV.

```
from appython import Protector
protector = Protector()
session = protector.create_session("ALL_USER")
data=bytes("Protegrity1", encoding="utf-8")
p_out = session.protect(data, "SUCCESS_BYTE",
```

```
        external_iv=bytes("1234", encoding="utf-8"))
print("Protected Data: %s" %p_out)
r_out = session.reprotect(p_out, "SUCCESS_BYTE",
        "SUCCESS_REPROTECT_BYTE", old_external_iv=bytes("1234", encoding="utf-8"),
        new_external_iv=bytes("123456", encoding="utf-8"))
print("Reprotected Data: %s" %r_out)
```

Result

```
Protected Data: b'Ho9bgXoebxa'
Reprotected Data: b'vQIqelQyqY6'
```

2.4.4.3.59 Mock Example - Reprotecting Bulk Bytes Data

This section describes how to use the *reprotect* API for reprotecting bulk bytes data. You can pass bulk bytes data as a list or a tuple.

Caution: The individual elements of the list or tuple must be of the same data type.

Warning: If you are retokenizing the data using the *reprotect* API, then the old data element and the new data element must have the same tokenization type. For example, if you have used Alpha-Numeric data element to protect the data, then you must use only Alpha-Numeric data element to reprotect the data.

Example

In the following example, *protegrity1234*, *Protegrity1*, and *Protegrity56* strings are first converted to bytes using the Python *bytes()* method. The converted bytes are then stored in a list and used as bulk data, which is protected using the *SUCCESS_BYTE* data element.

The tokenized input data, the old data element *SUCCESS_BYTE*, and a new data element *SUCCESS_REPROTECT_BYTE* are then passed as inputs to the *reprotect* API. The *reprotect* API first unprotects the protected input data using the old data element and then reprotects it using the new data element, as part of a single reprotect operation.

```
from appython import Protector
protector = Protector()
session = protector.create_session("ALL_USER")
data = [bytes("protegrity1234"), bytes("Protegrity1"), bytes("Protegrity56")]
p_out = session.protect(data, "SUCCESS_BYTE")
print("Protected Data: ")
print(p_out)
r_out = session.reprotect(p_out[0], "SUCCESS_BYTE",
        "SUCCESS_REPROTECT_BYTE")
print("Reprotected Data: ")
print(r_out)
```

Result

```
Protected Data:
([b'pJPqrjJEqLXHaO', b'6JPqrjJEqLX', b'6JPqrjJEqLl5'], (6, 6, 6))
Reprotected Data:
([b'gQbePhQ2eGCjqW', b'JQbePhQ2eGC', b'JQbePhQ2eGBK'], (6, 6, 6))
```

6 is the success return code for the protect operation of each element in the list.

2.4.4.3.60 Mock Example - Reprotecting Bulk Bytes Data with External IV

This section describes how to use the *reprotect* API for reprotecting bulk bytes data using external IV. You can pass bulk bytes data as a list or a tuple.

Caution: The individual elements of the list or tuple must be of the same data type.

Warning: If you are retokenizing the data using the *reprotect* API, then the old data element and the new data element must have the same tokenization type. For example, if you have used Alpha-Numeric data element to protect the data, then you must use only Alpha-Numeric data element to reprotect the data.

Example

In the following example, *protegrity1234*, *Protegrity1*, and *Protegrity56* strings are first converted to bytes using the Python *bytes()* method. The converted bytes are then stored in a list and used as bulk data, which is protected using the *SUCCESS_BYTE* data element, with the help of an external IV *1234* that is passed as bytes.

The protected input data, the *SUCCESS_BYTE* data element, a new data element *SUCCESS_REPROTECT_BYTE*, the old external IV *1234* in bytes, and a new external IV *123456* in bytes are then passed as inputs to the *reprotect* API. As part of a single reprotect operation, the *reprotect* API first unprotects the protected input data using the given data element and old external IV, and then reprotects it using the new data element and with the new external IV.

```
from appython import Protector
protector = Protector()
session = protector.create_session("ALL_USER")
data = [bytes("protegrity1234", encoding="utf-8"), bytes("Protegrity1",
encoding="utf-8"), bytes("Protegrity56", encoding="utf-8")]
p_out = session.protect(data, "SUCCESS_BYTE",
        external_iv=bytes("1234", encoding="utf-8"))
print("Protected Data: ")
print(p_out)
r_out = session.reprotect(p_out[0], "SUCCESS_BYTE",
        "SUCCESS_REPROTECT_BYTE", old_external_iv=bytes("1234", encoding="utf-8"),
        new_external_iv=bytes("123456", encoding="utf-8"))
print("Reprotected Data: ")
print(r_out)
```

Result

```
Protected Data:
([b'fo9bgXoebxaCTN', b'Ho9bgXoebxa', b'Ho9bgXoebx2q'], (6, 6, 6))
Reprotected Data:
([b'cQIqelQyqY6OoN', b'vQIqelQyqY6', b'vQIqelQyqYXa'], (6, 6, 6))
```

6 is the success return code for the protect operation of each element in the list.

2.4.4.3.61 Mock Example - Protecting Date Object in DD/MM/YYYY Format

This section describes how to use the *protect* API for protecting the date object.

Note: For information about the date formats supported by Protegrity tokenization methods, refer to the section *Protegrity Tokenization* in the *Protection Methods Reference Guide 9.1.0.0*.

Example: Input date object in DD/MM/YYYY format

In the following example, the *27/01/2019* date string is used as the data, which is first converted to a date object using the Python *date* method of the *datetime* module.

The date object is then protected using the *SUCCESS_DATE_DDMMYYYY* data element.

For information regarding the Python *datetime* module, refer to the *Python documentation*.

```
from appython import Protector
from datetime import datetime
protector = Protector()
session = protector.create_session("ALL_USER")
data = datetime.strptime("27/01/2019", "%d/%m/%Y").date()
print("\nInput date as a Date object : "+str(data))
p_out = session.protect(data, "SUCCESS_DATE_DDMMYYYY")
print("Protected date: "+str(p_out))
```

Result

```
Input date as a Date object : 2019-01-27
Protected date: 2022-06-14
```

2.4.4.3.62 Mock Example - Protecting Bulk Date Objects in DD/MM/YYYY Format

This section describes how to use the *protect* API for protecting bulk date objects. You can pass bulk date objects as a list or a tuple.

Caution: The individual elements of the list or tuple must be of the same data type.

Note: For information about the date formats supported by Protegrity tokenization methods, refer to the section *Protegrity Tokenization* in the *Protection Methods Reference Guide 9.1.0.0*.

Example

In the following example, the *27/01/2019* and *22/04/2018* date strings are used as the data, which are first converted to a date objects using the Python *date* method of the *datetime* module. The two date objects are then used to create a list, which is used as the input data.

The input list is then protected using the *SUCCESS_DATE_DDMMYYYY* data element.

For information regarding the Python *datetime* module, refer to the *Python documentation*.

```
from appython import Protector
from datetime import datetime
protector = Protector()
session = protector.create_session("ALL_USER")
data1 = datetime.strptime("27/01/2019", "%d/%m/%Y").date()
data2 = datetime.strptime("22/04/2018", "%d/%m/%Y").date()
data = [data1, data2]
print("Input data: "+str(data))
p_out = session.protect(data, "SUCCESS_DATE_DDMMYYYY")
print("Protected data: "+str(p_out))
```

Result

```
Input data: [datetime.date(2019, 1, 27), datetime.date(2018, 4, 22)]
Protected data: ([datetime.date(2022, 6, 14), datetime.date(2021, 9, 7)], (6, 6))
```

6 is the success return code for the protect operation of each element in the list.

2.4.4.3.63 Mock Example - Protecting Date String in DD/MM/YYYY Format

This section describes how to use the *protect* API for protecting a date string in DD/MM/YYYY format.

> **Note:** For information about the date formats supported by Protegrity tokenization methods, refer to the section *Protegrity Tokenization* in the *Protection Methods Reference Guide 9.1.0.0*.

Example: Input date string in DD/MM/YYYY format

In the following example, the *27/01/2019* date string is used as the input data, which is protected using the *SUCCESS_DATE_DDMMYYYY* data element.

```
from appython import Protector
protector = Protector()
session = protector.create_session("ALL_USER")
p_out = session.protect("27/01/2019", "SUCCESS_DATE_DDMMYYYY")
print("Protected date: " + p_out)
```

Result

```
Protected date: 14/06/2022
```

2.4.4.3.64 Mock Example - Protecting Bulk Date Strings in DD/MM/YYYY Format

This section describes how to use the *protect* API for protecting bulk date strings. You can pass bulk date strings as a list or a tuple.

> **Caution:** The individual elements of the list or tuple must be of the same data type.

> **Note:** For information about the date formats supported by Protegrity tokenization methods, refer to the section *Protegrity Tokenization* in the *Protection Methods Reference Guide 9.1.0.0*.

Example

In the following example, the *27/01/2019* and *22/04/2018* date strings are used to create a list, which is used as the input data. The input list is then protected using the *SUCCESS_DATE_DDMMYYYY* data element.

```
from appython import Protector
protector = Protector()
session = protector.create_session("ALL_USER")
data = ["27/01/2019", "22/04/2018"]
print("Input data: " + str(data))
p_out = session.protect(data, "SUCCESS_DATE_DDMMYYYY")
print("Protected data: " + str(p_out))
```

Result

```
Input data: ['27/01/2019', '22/04/2018']
Protected data: (['14/06/2022', '07/09/2021'], (6, 6))
```

6 is the success return code for the protect operation of each element in the list.

2.4.4.3.65 Mock Example - Unprotecting Date Objects in DD/MM/YYYY Format

This section describes how to use the *unprotect* API for retrieving the original data object from protected data.

> **Note:** For information about the date formats supported by Protegrity tokenization methods, refer to the section *Protegrity Tokenization* in the *Protection Methods Reference Guide 9.1.0.0*.

Example: Input date object in DD/MM/YYYY format

In the following example, the *27/01/2019* date string is used as the data, which is first converted to a date object using the Python *date* method of the *datetime* module.

The date object is first protected using the *SUCCESS_DATE_DDMMYYYY* data element, and is then unprotected using the same data element.

For information regarding the Python *datetime* module, refer to the *Python documentation*.

```
from appython import Protector
from datetime import datetime
protector = Protector()
session = protector.create_session("ALL_USER")
data = datetime.strptime("27/01/2019", "%d/%m/%Y").date()
print("\nInput date as a Date object : "+str(data))
p_out = session.protect(data, "SUCCESS_DATE_DDMMYYYY")
print("Protected date: "+str(p_out))
unprotected_output = session.unprotect(p_out, "SUCCESS_DATE_DDMMYYYY")
print("Unprotected date: "+str(unprotected_output))
```

Result

```
Input date as a Date object : 2019-01-27
Protected date: 2022-06-14
Unprotected date: 2019-01-27
```

2.4.4.3.66 Mock Example - Unprotecting Bulk Date Objects in DD/MM/YYYY Format

This section describes how to use the *unprotect* API for retrieving the original bulk date objects from token data.

> **Note:** For information about the date formats supported by Protegrity tokenization methods, refer to the section *Protegrity Tokenization* in the *Protection Methods Reference Guide 9.1.0.0*.

Example

In the following example, the *27/01/2019* and *22/04/2018* date strings are used as the data, which are first converted to a date objects using the Python *date* method of the *datetime* module. The two date objects are then used to create a list, which is used as the input data.

The input list is then protected using the *SUCCESS_DATE_DDMMYYYY* data element, and then unprotected using the same data element.

For information regarding the Python *datetime* module, refer to the *Python documentation*.

```
from appython import Protector
from datetime import datetime
protector = Protector()
session = protector.create_session("ALL_USER")
data1 = datetime.strptime("27/01/2019", "%d/%m/%Y").date()
data2 = datetime.strptime("22/04/2018", "%d/%m/%Y").date()
data = [data1, data2]
```

```
print("Input data: "+str(data))
p_out = session.protect(data, "SUCCESS_DATE_DDMMYYYY")
print("Protected data: "+str(p_out))
unprotected_output = session.unprotect(p_out[0], "SUCCESS_DATE_DDMMYYYY")
print("Unprotected data: "+str(unprotected_output))
```

Result

```
Input data: [datetime.date(2019, 1, 27), datetime.date(2018, 4, 22)]
Protected data: ([datetime.date(2022, 6, 14), datetime.date(2021, 9, 7)], (6, 6))
Unprotected data: ([datetime.date(2019, 1, 27), datetime.date(2018, 4, 22)], (8, 8))
```

6 is the success return code for the protect operation of each element in the list.

8 is the success return code for the unprotect operation of each element in the list.

2.4.4.3.67 Mock Example - Unprotecting Date String in DD/MM/YYYY Format

This section describes how to use the *unprotect* API for retrieving the original data string from protected data.

> **Note:** For information about the date formats supported by Protegrity tokenization methods, refer to the section *Protegrity Tokenization* in the *Protection Methods Reference Guide 9.1.0.0*.

Example: Input date string in DD/MM/YYYY format

In the following example, the *27/01/2019* date string that was protected using the *SUCCESS_DATE_DDMMYYYY* data element, is unprotected using the *SUCCESS_DATE_DDMMYYYY* data element.

```
from appython import Protector
protector = Protector()
session = protector.create_session("ALL_USER")
p_out = session.protect("27/01/2019", "SUCCESS_DATE_DDMMYYYY")
print("Protected date: "+str(p_out))
unprotected_output = session.unprotect(p_out, "SUCCESS_DATE_DDMMYYYY")
print("Unprotected date: "+str(unprotected_output))
```

Result

```
Protected date: 14/06/2022
Unprotected date: 27/01/2019
```

2.4.4.3.68 Mock Example - Unprotecting Bulk Date Strings in DD/MM/YYYY Format

This section describes how to use the *unprotect* API for retrieving the original bulk date strings from token data.

> **Note:** For information about the date formats supported by Protegrity tokenization methods, refer to the section *Protegrity Tokenization* in the *Protection Methods Reference Guide 9.1.0.0*.

Example

In the following example, the *27/01/2019* and *22/04/2018* date strings are used to create a list, which is used as the input data.

The input list is then protected using the *SUCCESS_DATE_DDMMYYYY* data element, and then unprotected using the same data element.

```
from appython import Protector
protector = Protector()
session = protector.create_session("ALL_USER")
```

```
data = ["27/01/2019", "22/04/2018"]
print("Input data: "+str(data))
p_out = session.protect(data, "SUCCESS_DATE_DDMMYYYY")
print("Protected data: "+str(p_out))
unprotected_output = session.unprotect(p_out[0], "SUCCESS_DATE_DDMMYYYY")
print("Unprotected data: "+str(unprotected_output))
```

Result

```
Input data: ['27/01/2019', '22/04/2018']
Protected data: (['14/06/2022', '07/09/2021'], (6, 6))
Unprotected data: (['27/01/2019', '22/04/2018'], (8, 8))
```

2.4.4.3.69 Mock Example - Reprotecting Date Object in DD/MM/YYYY Format

This section describes how to use the *reprotect* API for reprotecting a date object.

Warning: If you are retokenizing the data using the *reprotect* API, then the old data element and the new data element must have the same tokenization type. For example, if you have used the Date (DD/MM/YYYY) data element to protect the data, then you must use only the Date (DD/MM/YYYY) data element to reprotect the data.

Note: For information about the date formats supported by Protegrity tokenization methods, refer to the section *Protegrity Tokenization* in the *Protection Methods Reference Guide 9.1.0.0*.

Example

In the following example, the *27/01/2019* date string is used as the data, which is first converted to a date object using the Python *date* method of the *datetime* module. The date object is then protected using the *SUCCESS_DATE_DDMMYYYY* data element.

The protected input data, the old data element *SUCCESS_DATE_DDMMYYYY*, and a new data element *SUCCESS_REPROTECT_DATE_DDMMYYYY* are then passed as inputs to the *reprotect* API. The *reprotect* API first unprotects the protected input data using the old data element and then reprotects it using the new data element, as part of a single reprotect operation.

For information regarding the Python *datetime* module, refer to the *Python documentation*.

```
from appython import Protector
from datetime import datetime
protector = Protector()
session = protector.create_session("ALL_USER")
data = datetime.strptime("27/01/2019", "%d/%m/%Y").date()
print("\nInput date as a Date object : "+str(data))
p_out = session.protect(data, "SUCCESS_DATE_DDMMYYYY")
print("Protected date: "+str(p_out))
r_out = session.reprotect(p_out, "SUCCESS_DATE_DDMMYYYY",
"SUCCESS_REPROTECT_DATE_DDMMYYYY")
print("Reprotected date: "+str(r_out))
```

Result

```
Input date as a Date object : 2019-01-27
Protected date: 2022-06-14
Reprotected date: 2030-11-26
```

2.4.4.3.70 Mock Example - Reprotecting Bulk Date Objects in DD/MM/YYYY Format

This section describes how to use the *reprotect* API for reprotecting bulk date objects. You can pass bulk date objects as a list or a tuple.

Caution: The individual elements of the list or tuple must be of the same data type.

Warning: If you are retokenizing the data using the *reprotect* API, then the old data element and the new data element must have the same tokenization type. For example, if you have used the Date (DD/MM/YYYY) data element to protect the data, then you must use only the Date (DD/MM/YYYY) data element to reprotect the data.

Note: For information about the date formats supported by Protegrity tokenization methods, refer to the section *Protegrity Tokenization* in the *Protection Methods Reference Guide 9.1.0.0*.

Example

In the following example, the two date strings *27/01/2019* and *22/04/2018* are used as data, which are first converted to date objects using the Python *date* method of the *datetime* module. The two date objects are joined together to create a list, which is protected using the *SUCCESS_DATE_DDMMYYYY* data element.

The protected input data, the old data element *SUCCESS_DATE_DDMMYYYY*, and a new data element *SUCCESS_REPROTECT_DATE_DDMMYYYY* are then passed as inputs to the *reprotect* API. The *reprotect* API first unprotects the protected input data using the old data element and then reprotects it using the new data element, as part of a single reprotect operation.

For information regarding the Python *datetime* module, refer to the *Python documentation*.

```
from appython import Protector
from datetime import datetime
protector = Protector()
session = protector.create_session("ALL_USER")
data1 = datetime.strptime("27/01/2019", "%d/%m/%Y").date()
data2 = datetime.strptime("22/04/2018", "%d/%m/%Y").date()
data = [data1, data2]
print("Input data: "+str(data))
p_out = session.protect(data, "SUCCESS_DATE_DDMMYYYY")
print("Protected data: "+str(p_out))
r_out = session.reprotect(p_out[0], "SUCCESS_DATE_DDMMYYYY",
"SUCCESS_REPROTECT_DATE_DDMMYYYY")
print("Reprotected data: "+str(r_out))
```

Result

```
Input data: [datetime.date(2019, 1, 27), datetime.date(2018, 4, 22)]
Protected data: ([datetime.date(2022, 6, 14), datetime.date(2021, 9, 7)], (6, 6))
Reprotected data: ([datetime.date(2030, 11, 26), datetime.date(2030, 2, 19)], (6, 6))
```

6 is the success return code for the protect operation of each element in the list.

2.4.4.3.71 Mock Example - Reprotecting Date String in DD/MM/YYYY Format

This section describes how to use the *reprotect* API for reprotecting a date in string format.

Warning: If you are retokenizing the data using the *reprotect* API, then the old data element and the new data element must have the same tokenization type. For example, if you have used the Date (DD/MM/YYYY) data element to protect the data, then you must use only the Date (DD/MM/YYYY) data element to reprotect the data.

Note: For information about the date formats supported by Protegrity tokenization methods, refer to the section *Protegrity Tokenization* in the *Protection Methods Reference Guide 9.1.0.0*.

Example

In the following example, the *27/01/2019* date string is protected using the *SUCCESS_DATE_DDMMYYYY* data element.

The protected input data, the old data element *SUCCESS_DATE_DDMMYYYY*, and a new data element *SUCCESS_REPROTECT_DATE_DDMMYYYY* are then passed as inputs to the *reprotect* API. The *reprotect* API first unprotects the protected input data using the old data element and then reprotects it using the new data element, as part of a single reprotect operation.

```
from appython import Protector
protector = Protector()
session = protector.create_session("ALL_USER")
p_out = session.protect("27/01/2019", "SUCCESS_DATE_DDMMYYYY")
print("Protected date: "+str(p_out))
r_out = session.reprotect(p_out, "SUCCESS_DATE_DDMMYYYY",
"SUCCESS_REPROTECT_DATE_DDMMYYYY")
print("Reprotected date: "+str(r_out))
```

Result

```
Protected date: 14/06/2022
Reprotected date: 26/11/2030
```

2.4.4.3.72 Mock Example - Reprotecting Bulk Date Strings in DD/MM/YYYY Format

This section describes how to use the *reprotect* API for reprotecting bulk date strings. You can pass bulk date strings as a list or a tuple.

Caution: The individual elements of the list or tuple must be of the same data type.

Warning: If you are retokenizing the data using the *reprotect* API, then the old data element and the new data element must have the same tokenization type. For example, if you have used the Date (DD/MM/YYYY) data element to protect the data, then you must use only the Date (DD/MM/YYYY) data element to reprotect the data.

Note: For information about the date formats supported by Protegrity tokenization methods, refer to the section *Protegrity Tokenization* in the *Protection Methods Reference Guide 9.1.0.0*.

Example

In the following example, the two date strings *27/01/2019* and *22/04/2018* are used to create a list, which is protected using the *SUCCESS_DATE_DDMMYYYY* data element.

The protected input data, the old data element *SUCCESS_DATE_DDMMYYYY*, and a new data element *SUCCESS_REPROTECT_DATE_DDMMYYYY* are then passed as inputs to the *reprotect* API. The *reprotect* API first

unprotects the protected input data using the old data element and then reprotects it using the new data element, as part of a single reprotect operation.

```
from appython import Protector
protector = Protector()
session = protector.create_session("ALL_USER")
data = ["27/01/2019", "22/04/2018"]
print("Input data: "+str(data))
p_out = session.protect(data, "SUCCESS_DATE_DDMMYYYY")
print("Protected data: "+str(p_out))
r_out = session.reprotect(p_out[0], "SUCCESS_DATE_DDMMYYYY",
"SUCCESS_REPROTECT_DATE_DDMMYYYY")
print("Reprotected data: "+str(r_out))
```

Result

```
Input data: ['27/01/2019', '22/04/2018']
Protected data: (['14/06/2022', '07/09/2021'], (6, 6))
Reprotected data: (['26/11/2030', '19/02/2030'], (6, 6))
```

6 is the success return code for the protect operation of each element in the list.

2.4.4.3.73 Mock Example - Protecting Date Object in MM.DD.YYYY Format

This section describes how to use the *protect* API for protecting the date object.

> **Note:** For information about the date formats supported by Protegrity tokenization methods, refer to the section *Protegrity Tokenization* in the *Protection Methods Reference Guide 9.1.0.0*.

Example: Input date object in MM.DD.YYYY format

In the following example, the *01/27/2019* date string is used as the data, which is first converted to a date object using the Python *date* method of the *datetime* module.

The date object is then protected using the *SUCCESS_DATE_MMDDYYYY* data element.

For information regarding the Python *datetime* module, refer to the *Python documentation*.

```
from appython import Protector
from datetime import datetime
protector = Protector()
session = protector.create_session("ALL_USER")
data = datetime.strptime("01/27/2019", "%m/%d/%Y").date()
print("\nInput date as a Date object : "+str(data))
p_out = session.protect(data, "SUCCESS_DATE_MMDDYYYY")
print("Protected date: "+str(p_out))
```

Result

```
Input date as a Date object : 2019-01-27
Protected date: 2025-06-29
```

2.4.4.3.74 Mock Example - Protecting Bulk Date Objects in MM.DD.YYYY Format

This section describes how to use the *protect* API for tokenizing bulk date objects. You can pass bulk date objects as a list or a tuple.

Caution: The individual elements of the list or tuple must be of the same data type.

Note: For information about the date formats supported by Protegrity tokenization methods, refer to the section *Protegrity Tokenization* in the *Protection Methods Reference Guide 9.1.0.0*.

Example

In the following example, the *01/27/2019* and *04/22/2018* date strings are used as the data, which are first converted to a date objects using the Python *date* method of the *datetime* module. The two date objects are then used to create a list, which is used as the input data.

The input list is then protected using the *SUCCESS_DATE_MMDDYYYY* data element.

For information regarding the Python *datetime* module, refer to the *Python documentation*.

```
from appython import Protector
from datetime import datetime
protector = Protector()
session = protector.create_session("ALL_USER")
data1 = datetime.strptime("01/27/2019", "%m/%d/%Y").date()
data2 = datetime.strptime("04/22/2018", "%m/%d/%Y").date()
data = [data1, data2]
print("Input data: "+str(data))
p_out = session.protect(data, "SUCCESS_DATE_MMDDYYYY")
print("Protected data: "+str(p_out))
```

Result

```
Input data: [datetime.date(2019, 1, 27), datetime.date(2018, 4, 22)]
Protected data: ([datetime.date(2025, 6, 29), datetime.date(2024, 9, 22)], (6, 6))
```

6 is the success return code for the protect operation of each element in the list.

2.4.4.3.75 Mock Example - Protecting Date String in MM.DD.YYYY Format

This section describes how to use the *protect* API for protecting a date string in MM/DD/YYYY format.

Note: For information about the date formats supported by Protegrity tokenization methods, refer to the section *Protegrity Tokenization* in the *Protection Methods Reference Guide 9.1.0.0*.

Example: Input date string in MM.DD.YYYY format

In the following example, the *01/27/2019* date string is used as the data, which is protected using the *SUCCESS_DATE_MMDDYYYY* data element.

```
from appython import Protector
protector = Protector()
session = protector.create_session("ALL_USER")
p_out = session.protect("01/27/2019", "SUCCESS_DATE_MMDDYYYY")
print("Protected date: " + p_out)
```

Result

```
Protected date: 06/29/2025
```

2.4.4.3.76 Mock Example - Protecting Bulk Date Strings in MM.DD.YYYY Format

This section describes how to use the *protect* API for tokenizing bulk dates in string format. You can pass bulk date strings as a list or a tuple.

Caution: The individual elements of the list or tuple must be of the same data type.

Note: For information about the date formats supported by Protegrity tokenization methods, refer to the section *Protegrity Tokenization* in the *Protection Methods Reference Guide 9.1.0.0*.

Example

In the following example, the *01/27/2019* and *04/22/2018* date strings are used to create a list, which is used as the input data. The input list is then protected using the *SUCCESS_DATE_MMDDYYYY* data element.

```
from appython import Protector
protector = Protector()
session = protector.create_session("ALL_USER")
data = ["01/27/2019", "04/22/2018"]
print("Input data: "+str(data))
p_out = session.protect(data, "SUCCESS_DATE_MMDDYYYY")
print("Protected data: "+str(p_out))
```

Result

```
Input data: ['01/27/2019', '04/22/2018']
Protected data: (['06/29/2025', '09/22/2024'], (6, 6))
```

6 is the success return code for the protect operation of each element in the list.

2.4.4.3.77 Mock Example - Unprotecting Date Objects in MM.DD.YYYY Format

This section describes how to use the *unprotect* API for retrieving the original data object from protected data.

Note: For information about the date formats supported by Protegrity tokenization methods, refer to the section *Protegrity Tokenization* in the *Protection Methods Reference Guide 9.1.0.0*.

Example: Input date object in MM.DD.YYYY format

In the following example, the *01/27/2019* date string is used as the data, which is first converted to a date object using the Python *date* method of the *datetime* module.

The date object is first protected using the *SUCCESS_DATE_MMDDYYYY* data element, and is then unprotected using the same data element.

For information regarding the Python *datetime* module, refer to the *Python documentation*.

```
from appython import Protector
from datetime import datetime
protector = Protector()
session = protector.create_session("ALL_USER")
data = datetime.strptime("01/27/2019", "%m/%d/%Y").date()
print("\nInput date as a Date object : "+str(data))
p_out = session.protect(data, "SUCCESS_DATE_MMDDYYYY")
```

```
print("Protected date: "+str(p_out))
unprotected_output = session.unprotect(p_out, "SUCCESS_DATE_MMDDYYYY")
print("Unprotected date: "+str(unprotected_output))
```

Result

```
Input date as a Date object : 2019-01-27
Protected date: 2025-06-29
Unprotected date: 2019-01-27
```

2.4.4.3.78 Mock Example - Unprotecting Bulk Date Objects in MM.DD.YYYY Format

This section describes how to use the *unprotect* API for retrieving the original bulk date objects from token data.

> **Note:** For information about the date formats supported by Protegrity tokenization methods, refer to the section *Protegrity Tokenization* in the *Protection Methods Reference Guide 9.1.0.0*.

Example

In the following example, the *01/27/2019* and *04/22/2018* date strings are used as the data, which are first converted to a date objects using the Python *date* method of the *datetime* module. The two date objects are then used to create a list, which is used as the input data.

The input list is then protected using the *SUCCESS_DATE_MMDDYYYY* data element, and then unprotected using the same data element.

For information regarding the Python *datetime* module, refer to the *Python documentation*.

```
from appython import Protector
from datetime import datetime
protector = Protector()
session = protector.create_session("ALL_USER")
data1 = datetime.strptime("01/27/2019", "%m/%d/%Y").date()
data2 = datetime.strptime("04/22/2018", "%m/%d/%Y").date()
data = [data1, data2]
print("Input data: "+str(data))
p_out = session.protect(data, "SUCCESS_DATE_MMDDYYYY")
print("Protected data: "+str(p_out))
unprotected_output = session.unprotect(p_out[0], "SUCCESS_DATE_MMDDYYYY")
print("Unprotected data: "+str(unprotected_output))
```

Result

```
Input data: [datetime.date(2019, 1, 27), datetime.date(2018, 4, 22)]
Protected data: ([datetime.date(2025, 6, 29), datetime.date(2024, 9, 22)], (6, 6))
Unprotected data: ([datetime.date(2019, 1, 27), datetime.date(2018, 4, 22)], (8, 8))
```

6 is the success return code for the protect operation of each element in the list.

8 is the success return code for the unprotect operation of each element in the list.

2.4.4.3.79 Mock Example - Unprotecting Date Objects in MM.DD.YYYY Format

This section describes how to use the *unprotect* API for retrieving the original data object from protected data.

> **Note:** For information about the date formats supported by Protegrity tokenization methods, refer to the section *Protegrity Tokenization* in the *Protection Methods Reference Guide 9.1.0.0*.

Example: Input date object in MM.DD.YYYY format

In the following example, the *01/27/2019* date string that was protected using the *SUCCESS_DATE_MMDDYYYY* data element, is unprotected using the same data element.

For information regarding the Python *datetime* module, refer to the *Python documentation*.

```
from appython import Protector
protector = Protector()
session = protector.create_session("ALL_USER")
p_out = session.protect("01/27/2019", "SUCCESS_DATE_MMDDYYYY")
print("Protected date: "+str(p_out))
unprotected_output = session.unprotect(p_out, "SUCCESS_DATE_MMDDYYYY")
print("Unprotected date: "+str(unprotected_output))
```

Result

```
Protected date: 06/29/2025
Unprotected date: 01/27/2019
```

2.4.4.3.80 Mock Example - Unprotecting Bulk Date Strings in MM.DD.YYYY Format

This section describes how to use the *unprotect* API for retrieving the original bulk date strings from token data.

> **Note:** For information about the date formats supported by Protegrity tokenization methods, refer to the section *Protegrity Tokenization* in the *Protection Methods Reference Guide 9.1.0.0*.

Example

In the following example, the *01/27/2019* and *04/22/2018* date strings are used to create a list, which is used as the input data.

The input list is then protected using the *SUCCESS_DATE_MMDDYYYY* data element, and then unprotected using the same data element.

```
from appython import Protector
protector = Protector()
session = protector.create_session("ALL_USER")
data = ["01/27/2019", "04/22/2018"]
print("Input data: "+str(data))
p_out = session.protect(data, "SUCCESS_DATE_MMDDYYYY")
print("Protected data: "+str(p_out))
unprotected_output = session.unprotect(p_out[0], "SUCCESS_DATE_MMDDYYYY")
print("Unprotected data: "+str(unprotected_output))
```

Result

```
Input data: ['01/27/2019', '04/22/2018']
Protected data: (['06/29/2025', '09/22/2024'], (6, 6))
Unprotected data: (['01/27/2019', '04/22/2018'], (8, 8))
```

6 is the success return code for the protect operation of each element in the list.

8 is the success return code for the unprotect operation of each element in the list.

2.4.4.3.81 Mock Example - Reprotecting Date Object in MM.DD.YYYY Format

This section describes how to use the *reprotect* API for reprotecting date object.

Warning: If you are retokenizing the data using the *reprotect* API, then the old data element and the new data element must have the same tokenization type. For example, if you have used the Date (DD/MM/YYYY) data element to protect the data, then you must use only the Date (DD/MM/YYYY) data element to reprotect the data.

Note: For information about the date formats supported by Protegrity tokenization methods, refer to the section *Protegrity Tokenization* in the *Protection Methods Reference Guide 9.1.0.0*.

Example

In the following example, the *01/27/2019* date string is used as the data, which is first converted to a date object using the Python *date* method of the *datetime* module. The datetime object is then protected using the *SUCCESS_DATE_MMDDYYYY* data element.

The protected input data, the old data element *SUCCESS_DATE_MMDDYYYY*, and a new data element *SUCCESS_REPROTECT_DATE_MMDDYYYY* are then passed as inputs to the *reprotect* API. The *reprotect* API first unprotects the protected input data using the old data element and then reprotects it using the new data element, as part of a single reprotect operation.

For information regarding the Python *datetime* module, refer to the *Python documentation*.

```
from appython import Protector
from datetime import datetime
protector = Protector()
session = protector.create_session("ALL_USER")
data = datetime.strptime("01/27/2019", "%m/%d/%Y").date()
print("\nInput date as a Date object : "+str(data))
p_out = session.protect(data, "SUCCESS_DATE_MMDDYYYY")
print("Protected date: "+str(p_out))
r_out = session.reprotect(p_out, "SUCCESS_DATE_MMDDYYYY",
"SUCCESS_REPROTECT_DATE_MMDDYYYY")
print("Reprotected date: "+str(r_out))
```

Result

```
Input date as a Date object : 2019-01-27
Protected date: 2025-06-29
Reprotected date: 2033-12-11
```

2.4.4.3.82 Mock Example - Reprotecting Bulk Date Objects in MM.DD.YYYY Format

This section describes how to use the *reprotect* API for reprotecting bulk date objects. You can pass bulk date objects as a list or a tuple.

Caution: The individual elements of the list or tuple must be of the same data type.

Warning: If you are retokenizing the data using the *reprotect* API, then the old data element and the new data element must have the same tokenization type. For example, if you have used the Date (DD/MM/YYYY) data element to protect the data, then you must use only the Date (DD/MM/YYYY) data element to reprotect the data.

Note: For information about the date formats supported by Protegrity tokenization methods, refer to the section *Protegrity Tokenization* in the *Protection Methods Reference Guide 9.1.0.0*.

Example

In the following example, the two date strings *01/27/2019* and *04/22/2018* are used as data, which are first converted to date objects using the Python *date* method of the *datetime* module. The two date objects are joined together to create a list, which is protected using the *SUCCESS_DATE_MMDDYYYY* data element.

The protected input data, the old data element *SUCCESS_DATE_MMDDYYYY*, and a new data element *SUCCESS_REPROTECT_DATE_MMDDYYYY* are then passed as inputs to the *reprotect* API. The *reprotect* API first unprotects the protected input data using the old data element and then reprotects it using the new data element, as part of a single reprotect operation.

For information regarding the Python *datetime* module, refer to the *Python documentation*.

```
from appython import Protector
from datetime import datetime
protector = Protector()
session = protector.create_session("ALL_USER")
data1 = datetime.strptime("01/27/2019", "%m/%d/%Y").date()
data2 = datetime.strptime("04/22/2018", "%m/%d/%Y").date()
data = [data1, data2]
print("Input data: "+str(data))
p_out = session.protect(data, "SUCCESS_DATE_MMDDYYYY")
print("Protected data: "+str(p_out))
r_out = session.reprotect(p_out[0], "SUCCESS_DATE_MMDDYYYY",
"SUCCESS_REPROTECT_DATE_MMDDYYYY")
print("Reprotected data: "+str(r_out))
```

Result

```
Input data: [datetime.date(2019, 1, 27), datetime.date(2018, 4, 22)]
Protected data: ([datetime.date(2025, 6, 29), datetime.date(2024, 9, 22)], (6, 6))
Reprotected data: ([datetime.date(2033, 12, 11), datetime.date(2033, 3, 6)], (6, 6))
```

6 is the success return code for the protect operation of each element in the list.

2.4.4.3.83 Mock Example - Reprotecting Date String in MM.DD.YYYY Format

This section describes how to use the *reprotect* API for reprotecting a date in string format.

Warning: If you are retokenizing the data using the *reprotect* API, then the old data element and the new data element must have the same tokenization type. For example, if you have used the Date (DD/MM/YYYY) data element to protect the data, then you must use only the Date (DD/MM/YYYY) data element to reprotect the data.

Note: For information about the date formats supported by Protegrity tokenization methods, refer to the section *Protegrity Tokenization* in the *Protection Methods Reference Guide 9.1.0.0*.

Example

In the following example, the *01/27/2019* date string is used as the data, which is protected using the *SUCCESS_DATE_MMDDYYYY* data element.

The protected input data, the old data element *SUCCESS_DATE_MMDDYYYY*, and a new data element *SUCCESS_REPROTECT_DATE_MMDDYYYY* are then passed as inputs to the *reprotect* API. The *reprotect* API first unprotects the protected input data using the old data element and then reprotects it using the new data element, as part of a single reprotect operation.

```
from appython import Protector
protector = Protector()
session = protector.create_session("ALL_USER")
```

```
p_out = session.protect("01/27/2019", "SUCCESS_DATE_MMDDYYYY")
print("Protected date: "+str(p_out))
r_out = session.reprotect(p_out, "SUCCESS_DATE_MMDDYYYY",
"SUCCESS_REPROTECT_DATE_MMDDYYYY")
print("Reprotected date: "+str(r_out))
```

Result

```
Protected date: 06/29/2025
Reprotected date: 12/11/2033
```

2.4.4.3.84 Mock Example - Reprotecting Bulk Date Strings in MM.DD.YYYY Format

This section describes how to use the *reprotect* API for reprotecting bulk dates in string format. You can pass bulk date strings as a list or a tuple.

Caution: The individual elements of the list or tuple must be of the same data type.

Warning: If you are retokenizing the data using the *reprotect* API, then the old data element and the new data element must have the same tokenization type. For example, if you have used the Date (DD/MM/YYYY) data element to protect the data, then you must use only the Date (DD/MM/YYYY) data element to reprotect the data.

Note: For information about the date formats supported by Protegrity tokenization methods, refer to the section *Protegrity Tokenization* in the *Protection Methods Reference Guide 9.1.0.0*.

Example

In the following example, the two date strings *01/27/2019* and *04/22/2018* are used to create a list, which is protected using the *SUCCESS_DATE_MMDDYYYY* data element.

The protected input data, the old data element *SUCCESS_DATE_MMDDYYYY*, and a new data element *SUCCESS_REPROTECT_DATE_MMDDYYYY* are then passed as inputs to the *reprotect* API. The *reprotect* API first unprotects the protected input data using the old data element and then reprotects it using the new data element, as part of a single reprotect operation.

```
from appython import Protector
protector = Protector()
session = protector.create_session("ALL_USER")
data = ["01/27/2019", "04/22/2018"]
print("Input data: "+str(data))
p_out = session.protect(data, "SUCCESS_DATE_MMDDYYYY")
print("Protected data: "+str(p_out))
r_out = session.reprotect(p_out[0], "SUCCESS_DATE_MMDDYYYY",
"SUCCESS_REPROTECT_DATE_MMDDYYYY")
print("Reprotected data: "+str(r_out))
```

Result

```
Input data: ['01/27/2019', '04/22/2018']
Protected data: (['06/29/2025', '09/22/2024'], (6, 6))
Reprotected data: (['12/11/2033', '03/06/2033'], (6, 6))
```

6 is the success return code for the protect operation of each element in the list.

2.4.4.3.85 Mock Example - Reprotecting Bulk Date Strings in MM.DD.YYYY Format

This section describes how to use the *reprotect* API for reprotecting bulk dates in string format. You can pass bulk date strings as a list or a tuple.

Caution: The individual elements of the list or tuple must be of the same data type.

Warning: If you are retokenizing the data using the *reprotect* API, then the old data element and the new data element must have the same tokenization type. For example, if you have used the Date (DD/MM/YYYY) data element to protect the data, then you must use only the Date (DD/MM/YYYY) data element to reprotect the data.

Note: For information about the date formats supported by Protegrity tokenization methods, refer to the section *Protegrity Tokenization* in the *Protection Methods Reference Guide 9.1.0.0*.

Example

In the following example, the two date strings *01/27/2019* and *04/22/2018* are used to create a list, which is protected using the *SUCCESS_DATE_MMDDYYYY* data element.

The protected input data, the old data element *SUCCESS_DATE_MMDDYYYY*, and a new data element *SUCCESS_REPROTECT_DATE_MMDDYYYY* are then passed as inputs to the *reprotect* API. The *reprotect* API first unprotects the protected input data using the old data element and then reprotects it using the new data element, as part of a single reprotect operation.

```
from appython import Protector
protector = Protector()
session = protector.create_session("ALL_USER")
data = ["01/27/2019", "04/22/2018"]
print("Input data: "+str(data))
p_out = session.protect(data, "SUCCESS_DATE_MMDDYYYY")
print("Protected data: "+str(p_out))
r_out = session.reprotect(p_out[0], "SUCCESS_DATE_MMDDYYYY",
"SUCCESS_REPROTECT_DATE_MMDDYYYY")
print("Reprotected data: "+str(r_out))
```

Result

```
Input data: ['01/27/2019', '04/22/2018']
Protected data: (['06/29/2025', '09/22/2024'], (6, 6))
Reprotected data: (['12/11/2033', '03/06/2033'], (6, 6))
```

6 is the success return code for the protect operation of each element in the list.

2.4.4.3.86 Mock Example - Protecting Bulk Date Objects in YYYY/MM/DD Format

This section explains how to use the *protect* API for protecting bulk date objects. You can pass bulk date objects as a list or a tuple.

Caution: The individual elements of the list or tuple must be of the same data type.

Example

In the following example, the *2019/01/27* and *2018/04/22* date strings are used as the data, which are first converted to a date object using the Python *date* method of the *datetime* module. The two date objects are then used to create a list, which is used as the input data.

The input list is then protected using the *SUCCESS_DATE_ YYYYMMDD* data element.

For information regarding the Python *datetime* module, refer to the *Python documentation*.

```
from appython import Protector
from datetime import datetime
protector = Protector()
session = protector.create_session("ALL_USER")
data1 = datetime.strptime("2019/01/27", "%Y/%m/%d").date()
data2 = datetime.strptime("2018/04/22", "%Y/%m/%d").date()
data = [data1, data2]
print("Input data: "+str(data))
p_out = session.protect(data, "SUCCESS_DATE_YYYYMMDD")
print("Protected data: "+str(p_out))
```

Result

```
Input data: [datetime.date(2019, 1, 27), datetime.date(2018, 4, 22)]
Protected data: ([datetime.date(2028, 7, 14), datetime.date(2027, 10, 8)], (6, 6))
```

6 is the success return code for the protect operation of each element in the list.

2.4.4.3.87 Mock Example - Protecting Date Object in YYYY-MM-DD Format

This section describes how to use the *protect* API for protecting the date object.

> **Note:** For information about the date formats supported by Protegrity tokenization methods, refer to the section *Protegrity Tokenization* in the *Protection Methods Reference Guide 9.1.0.0*.

Example: Input date object in YYYY-MM-DD format

In the following example, the *2019/01/27* date string is used as the data, which is first converted to a date object using the Python *date* method of the *datetime* module.

The date object is then protected using the *SUCCESS_DATE_ YYYYMMDD* data element.

For information regarding the Python *datetime* module, refer to the *Python documentation*.

```
from appython import Protector
from datetime import datetime
protector = Protector()
session = protector.create_session("ALL_USER")
data = datetime.strptime("2019/01/27", "%Y/%m/%d").date()
print("\nInput date as a Date object : "+str(data))
p_out = session.protect(data, "SUCCESS_DATE_YYYYMMDD")
print("Protected date: "+str(p_out))
```

Result

```
Input date as a Date object : 2019-01-27
Protected date: 2028-07-14
```

2.4.4.3.88 Mock Example - Protecting Bulk Date Objects in YYYY-MM-DD Format

This section describes how to use the *protect* API for protecting bulk date objects. You can pass bulk date objects as a list or a tuple.

Caution: The individual elements of the list or tuple must be of the same data type.

Note: For information about the date formats supported by Protegrity tokenization methods, refer to the section *Protegrity Tokenization* in the *Protection Methods Reference Guide 9.1.0.0*.

Example

In the following example, the *2019/01/27* and *2018/04/22* date strings are used as the data, which are first converted to a date object using the Python *date* method of the *datetime* module. The two date objects are then used to create a list, which is used as the input data.

The input list is then protected using the *SUCCESS_DATE_ YYYYMMDD* data element.

For information regarding the Python *datetime* module, refer to the *Python documentation*.

```
from appython import Protector
from datetime import datetime
protector = Protector()
session = protector.create_session("ALL_USER")
data1 = datetime.strptime("2019/01/27", "%Y/%m/%d").date()
data2 = datetime.strptime("2018/04/22", "%Y/%m/%d").date()
data = [data1, data2]
print("Input data: "+str(data))
p_out = session.protect(data, "SUCCESS_DATE_YYYYMMDD")
print("Protected data: "+str(p_out))
```

Result

```
Input data: [datetime.date(2019, 1, 27), datetime.date(2018, 4, 22)]
Protected data: ([datetime.date(2028, 7, 14), datetime.date(2027, 10, 8)], (6, 6))
```

6 is the success return code for the protect operation of each element in the list.

2.4.4.3.89 Mock Example - Unprotecting Date Objects in YYYY-MM-DD Format

This section describes how to use the *unprotect* API for retrieving the original data object from protected data.

Note: For information about the date formats supported by Protegrity tokenization methods, refer to the section *Protegrity Tokenization* in the *Protection Methods Reference Guide 9.1.0.0*.

Example: Input date object in YYYY-MM-DD format

In the following example, the *2019/01/27* date string is used as the data, which is first converted to a date object using the Python *date* method of the *datetime* module.

The date object is first protected using the *SUCCESS_DATE_ YYYYMMDD* data element, and is then unprotected using the same data element.

For information regarding the Python *datetime* module, refer to the *Python documentation*.

```
from appython import Protector
from datetime import datetime
```

```
protector = Protector()
session = protector.create_session("ALL_USER")
data = datetime.strptime("2019/01/27", "%Y/%m/%d").date()
print("\nInput date as a Date object : "+str(data))
p_out = session.protect(data, "SUCCESS_DATE_YYYYMMDD")
print("Protected date: "+str(p_out))
unprotected_output = session.unprotect(p_out, "SUCCESS_DATE_YYYYMMDD")
print("Unprotected date: "+str(unprotected_output))
```

Result

```
Input date as a Date object : 2019-01-27
Protected date: 2028-07-14
Unprotected date: 2019-01-27
```

2.4.4.3.90 Mock Example - Unprotecting Bulk Date Objects in YYYY-MM-DD Format

This section describes how to use the *unprotect* API for retrieving the original bulk date objects from token data.

> **Note:** For information about the date formats supported by Protegrity tokenization methods, refer to the section *Protegrity Tokenization* in the *Protection Methods Reference Guide 9.1.0.0*.

Example

In the following example, the *2019/01/27* and *2018/04/22* date strings are used as the data, which are first converted to date objects using the Python *date* method of the *datetime* module. The two date objects are then used to create a list, which is used as the input data.

The input list is then protected using the *SUCCESS_DATE_YYYYMMDD* data element, and then unprotected using the same data element.

For information regarding the Python *datetime* module, refer to the *Python documentation*.

```
from appython import Protector
from datetime import datetime
protector = Protector()
session = protector.create_session("ALL_USER")
data1 = datetime.strptime("2019/01/27", "%Y/%m/%d").date()
data2 = datetime.strptime("2018/04/22", "%Y/%m/%d").date()
data = [data1, data2]
print("Input data: "+str(data))
p_out = session.protect(data, "SUCCESS_DATE_YYYYMMDD")
print("Protected data: "+str(p_out))
unprotected_output = session.unprotect(p_out[0], "SUCCESS_DATE_YYYYMMDD")
print("Unprotected data: "+str(unprotected_output))
```

Result

```
Input data: [datetime.date(2019, 1, 27), datetime.date(2018, 4, 22)]
Protected data: ([datetime.date(2028, 7, 14), datetime.date(2027, 10, 8)], (6, 6))
Unprotected data: ([datetime.date(2019, 1, 27), datetime.date(2018, 4, 22)], (8, 8))
```

6 is the success return code for the protect operation of each element in the list.

8 is the success return code for the unprotect operation of each element in the list.

2.4.4.3.91 Mock Example - Unprotecting Date String in YYYY-MM-DD Format

This section describes how to use the *unprotect* API for retrieving the original data string from protected data.

> **Note:** For information about the date formats supported by Protegrity tokenization methods, refer to the section *Protegrity Tokenization* in the *Protection Methods Reference Guide 9.1.0.0*.

Example: Input date string in YYYY-MM-DD format

In the following example, the *2019/01/27* date string that was protected using the *SUCCESS_DATE_ YYYYMMDD* data element, is unprotected using the same data element.

```
from appython import Protector
protector = Protector()
session = protector.create_session("ALL_USER")
p_out = session.protect("2019/01/27", "SUCCESS_DATE_YYYYMMDD")
print("Protected date: "+str(p_out))
unprotected_output = session.unprotect(p_out, "SUCCESS_DATE_YYYYMMDD")
print("Unprotected date: "+str(unprotected_output))
```

Result

```
Protected date: 2028/07/14
Unprotected date: 2019/01/27
```

2.4.4.3.92 Mock Example - Unprotecting Bulk Date Strings in YYYY-MM-DD Format

This section describes how to use the *unprotect* API for retrieving the original bulk date strings from token data.

> **Note:** For information about the date formats supported by Protegrity tokenization methods, refer to the section *Protegrity Tokenization* in the *Protection Methods Reference Guide 9.1.0.0*.

Example

In the following example, the *2019/01/27* and *2018/04/22* date strings are used to create a list, which is used as the input data.

The input list is then protected using the *SUCCESS_DATE_ YYYYMMDD* data element, and then unprotected using the same data element.

```
from appython import Protector
protector = Protector()
session = protector.create_session("ALL_USER")
data = ["2019/01/27", "2018/04/22"]
print("Input data: "+str(data))
p_out = session.protect(data, "SUCCESS_DATE_YYYYMMDD")
print("Protected data: "+str(p_out))
unprotected_output = session.unprotect(p_out[0], "SUCCESS_DATE_YYYYMMDD")
print("Unprotected data: "+str(unprotected_output))
```

Result

```
Input data: ['2019/01/27', '2018/04/22']
Protected data: (['2028/07/14', '2027/10/08'], (6, 6))
Unprotected data: (['2019/01/27', '2018/04/22'], (8, 8))
```

6 is the success return code for the protect operation of each element in the list.

8 is the success return code for the unprotect operation of each element in the list.

2.4.4.3.93 Mock Example - Reprotecting Date Object in YYYY-MM-DD Format

This section describes how to use the *reprotect* API for reprotecting date object.

Warning: If you are retokenizing the data using the *reprotect* API, then the old data element and the new data element must have the same tokenization type. For example, if you have used the Date (DD/MM/YYYY) data element to protect the data, then you must use only the Date (DD/MM/YYYY) data element to reprotect the data.

Note: For information about the date formats supported by Protegrity tokenization methods, refer to the section *Protegrity Tokenization* in the *Protection Methods Reference Guide 9.1.0.0*.

Example

In the following example, the *2019/01/27* date string is used as the data, which is first converted to a date object using the Python *date* method of the *datetime* module. The date object is then protected using the *SUCCESS_DATE_ YYYYMMDD* data element.

The protected input data, the old data element *SUCCESS_DATE_ YYYYMMDD*, and a new data element *SUCCESS_REPROTECT_DATE_ YYYYMMDD* are then passed as inputs to the *reprotect* API. The *reprotect* API first unprotects the protected input data using the old data element and then reprotects it using the new data element, as part of a single reprotect operation.

For information regarding the Python *datetime* module, refer to the *Python documentation*.

```
from appython import Protector
from datetime import datetime
protector = Protector()
session = protector.create_session("ALL_USER")
data = datetime.strptime("2019/01/27", "%Y/%m/%d").date()
print("\nInput date as a Date object : "+str(data))
p_out = session.protect(data, "SUCCESS_DATE_YYYYMMDD")
print("Protected date: "+str(p_out))
r_out = session.reprotect(p_out, "SUCCESS_DATE_YYYYMMDD",
"SUCCESS_REPROTECT_DATE_YYYYMMDD")
print("Reprotected date: "+str(r_out))
```

Result

```
Input date as a Date object : 2019-01-27
Protected date: 2028-07-14
Reprotected date: 2036-12-26
```

2.4.4.3.94 Mock Example - Reprotecting Bulk Date Objects in YYYY-MM-DD Format

This section describes how to use the *reprotect* API for reprotecting bulk date objects. You can pass bulk date objects as a list or a tuple.

Caution: The individual elements of the list or tuple must be of the same data type.

Warning: If you are retokenizing the data using the *reprotect* API, then the old data element and the new data element must have the same tokenization type. For example, if you have used the Date (DD/MM/YYYY) data element to protect the data, then you must use only the Date (DD/MM/YYYY) data element to reprotect the data.

Note: For information about the date formats supported by Protegrity tokenization methods, refer to the section *Protegrity Tokenization* in the *Protection Methods Reference Guide 9.1.0.0*.

Example

In the following example, the two date strings *2019/01/27* and *2018/04/22* are used as data, which are first converted to date objects using the Python *date* method of the *datetime* module. The two date objects are joined together to create a list, which is protected using the *SUCCESS_DATE_YYYYMMDD* data element.

The protected input data, the old data element *SUCCESS_DATE_YYYYMMDD*, and a new data element *SUCCESS_REPROTECT_DATE_YYYYMMDD* are then passed as inputs to the *reprotect* API. The *reprotect* API first unprotects the protected input data using the old data element and then reprotects it using the new data element, as part of a single reprotect operation.

For information regarding the Python *datetime* module, refer to the *Python documentation*.

```
from appython import Protector
from datetime import datetime
protector = Protector()
session = protector.create_session("ALL_USER")
data1 = datetime.strptime("2019/01/27", "%Y/%m/%d").date()
data2 = datetime.strptime("2018/04/22", "%Y/%m/%d").date()
data = [data1, data2]
print("Input data: "+str(data))
p_out = session.protect(data, "SUCCESS_DATE_YYYYMMDD")
print("Protected data: "+str(p_out))
r_out = session.reprotect(p_out[0], "SUCCESS_DATE_YYYYMMDD",
"SUCCESS_REPROTECT_DATE_YYYYMMDD")
print("Reprotected data: "+str(r_out))
```

Result

```
Input data: [datetime.date(2019, 1, 27), datetime.date(2018, 4, 22)]
Protected data: ([datetime.date(2028, 7, 14), datetime.date(2027, 10, 8)], (6, 6))
Reprotected data: ([datetime.date(2036, 12, 26), datetime.date(2036, 3, 21)], (6, 6))
```

6 is the success return code for the protect operation of each element in the list.

2.4.4.3.95 Mock Example - Reprotecting Date String in YYYY-MM-DD Format

This section describes how to use the *reprotect* API for reprotecting a date in string format.

Warning: If you are retokenizing the data using the *reprotect* API, then the old data element and the new data element must have the same tokenization type. For example, if you have used the Date (DD/MM/YYYY) data element to protect the data, then you must use only the Date (DD/MM/YYYY) data element to reprotect the data.

Note: For information about the date formats supported by Protegrity tokenization methods, refer to the section *Protegrity Tokenization* in the *Protection Methods Reference Guide 9.1.0.0*.

Example

In the following example, the *2019/01/27* date string is protected using the *SUCCESS_DATE_YYYYMMDD* data element.

The protected input data, the old data element *SUCCESS_DATE_YYYYMMDD*, and a new data element *SUCCESS_REPROTECT_DATE_YYYYMMDD* are then passed as inputs to the *reprotect* API. The *reprotect* API first unprotects the protected input data using the old data element and then reprotects it using the new data element, as part of a single reprotect operation.

```
from appython import Protector
protector = Protector()
session = protector.create_session("ALL_USER")
```

```
p_out = session.protect("2019/01/27", "SUCCESS_DATE_YYYYMMDD")
print("Protected date: "+str(p_out))
r_out = session.reprotect(p_out, "SUCCESS_DATE_YYYYMMDD",
"SUCCESS_REPROTECT_DATE_YYYYMMDD")
print("Reprotected date: "+str(r_out))
```

Result

```
Protected date: 2028/07/14
Reprotected date: 2036/12/26
```

2.4.4.3.96 Mock Example - Reprotecting Bulk Date Strings in YYYY-MM-DD Format

This section describes how to use the *reprotect* API for reprotecting bulk dates in string format. You can pass bulk date strings as a list or a tuple.

Caution: The individual elements of the list or tuple must be of the same data type.

Warning: If you are retokenizing the data using the *reprotect* API, then the old data element and the new data element must have the same tokenization type. For example, if you have used the Date (DD/MM/YYYY) data element to protect the data, then you must use only the Date (DD/MM/YYYY) data element to reprotect the data.

Note: For information about the date formats supported by Protegrity tokenization methods, refer to the section *Protegrity Tokenization* in the *Protection Methods Reference Guide 9.1.0.0*.

Example

In the following example, the two date strings *2019/01/27* and *2018/04/22* are used to create a list, which is protected using the *SUCCESS_DATE_YYYYMMDD* data element.

The protected input data, the old data element *SUCCESS_DATE_YYYYMMDD*, and a new data element *SUCCESS_REPROTECT_DATE_YYYYMMDD* are then passed as inputs to the *reprotect* API. The *reprotect* API first unprotects the protected input data using the old data element and then reprotects it using the new data element, as part of a single reprotect operation.

```
from appython import Protector
protector = Protector()
session = protector.create_session("ALL_USER")
data = ["2019/01/27", "2018/04/22"]
print("Input data: "+str(data))
p_out = session.protect(data, "SUCCESS_DATE_YYYYMMDD")
print("Protected data: "+str(p_out))
r_out = session.reprotect(p_out[0], "SUCCESS_DATE_YYYYMMDD",
"SUCCESS_REPROTECT_DATE_YYYYMMDD")
print("Reprotected data: "+str(r_out))
```

Result

```
Input data: ['2019/01/27', '2018/04/22']
Protected data: (['2028/07/14', '2027/10/08'], (6, 6))
Reprotected data: (['2036/12/26', '2036/03/21'], (8, 8))
```

6 is the success return code for the protect operation of each element in the list.

2.4.4.3.97 Mock Example - Protecting Date and Time String

This section describes how to use the *protect* API for protecting the date and time string.

Warning: If you are providing the input as a Datetime object, then you must use the data element with the tokenization type as Datetime to protect the data.

Note: For information about the date formats supported by Protegrity tokenization methods, refer to the section *Protegrity Tokenization* in the *Protection Methods Reference Guide 9.1.0.0*.

Example: Input date and time string in YYYY-MM-DD HH:MM:SS MMM format

In the following example, the *2019/01/27 02:34:54.123* date and time string is protected using the *SUCCESS_DATETIME* data element.

```
from appython import Protector
protector = Protector()
session = protector.create_session("ALL_USER")
p_out = session.protect("2019/01/27 02:34:54.123", "SUCCESS_DATETIME")
print("Protected date: "+str(p_out))
```

Result

```
Protected date: 2021/10/27 08:16:34.123000
```

2.4.4.3.98 Mock Example - Protecting Bulk Date and Time Strings

This section describes how to use the *protect* API for protecting bulk date and time strings. You can pass bulk date and time strings as a list or a tuple.

Caution: The individual elements of the list or tuple must be of the same data type.

Note: For information about the date formats supported by Protegrity tokenization methods, refer to the section *Protegrity Tokenization* in the *Protection Methods Reference Guide 9.1.0.0*.

Example

In the following example, the *2019/01/27 02:34:54.123* and *2018/04/22 01:24:35.123* date and time strings are used to create a list, which is used as the input data.

The input list is then tokenized using the *SUCCESS_DATETIME* data element.

```
from appython import Protector
protector = Protector()
session = protector.create_session("ALL_USER")
data = ["2019/01/27 02:34:54.123", "2018/04/22 01:24:35.123"]
print("Input data: "+str(data))
p_out = session.protect(data, "SUCCESS_DATETIME")
print("Protected data: "+str(p_out))
```

Result

```
Input data: ['2019/01/27 02:34:54.123', '2018/04/22 01:24:35.123']
Protected data: (['2021/10/27 08:16:34.123000', '2021/01/20 07:06:15.123000'], (6, 6))
```

6 is the success return code for the protect operation of each element in the list.

2.4.4.3.99 Mock Example - Unprotecting Date and Time String

This section describes how to use the *unprotect* API for retrieving the original bulk data and time string from protected data.

> **Note:** For information about the date formats supported by Protegrity tokenization methods, refer to the section *Protegrity Tokenization* in the *Protection Methods Reference Guide 9.1.0.0*.

Example: Input date and time string in YYYY-MM-DD HH:MM:SS MMM format

In the following example, the *2019/01/27 02:34:54.123* date and time string that was protected using the *SUCCESS_DATETIME* data element, is unprotected using the same data element.

```
from appython import Protector
protector = Protector()
session = protector.create_session("ALL_USER")
p_out = session.protect("2019/01/27 02:34:54.123", "SUCCESS_DATETIME")
print("Protected date: "+str(p_out))
unprotected_output = session.unprotect(p_out, "SUCCESS_DATETIME")
print("Unprotected data: "+str(unprotected_output))
```

Result

```
Protected date: 2021/10/27 08:16:34.123000
Unprotected data: 2019/01/27 02:34:54.123000
```

2.4.4.3.100 Mock Example - Unprotecting Bulk Date and Time Strings

This section describes how to use the *unprotect* API for retrieving the original bulk date and time strings from token data.

> **Note:** For information about the date formats supported by Protegrity tokenization methods, refer to the section *Protegrity Tokenization* in the *Protection Methods Reference Guide 9.1.0.0*.

Example

In the following example, the *2019/01/27 02:34:54.123* and *2018/04/22 01:24:35.123* date and time strings are used to create a list, which is used as the input data.

The input list is then protected using the *SUCCESS_DATETIME* data element, and then unprotected using the same data element.

```
from appython import Protector
protector = Protector()
session = protector.create_session("ALL_USER")
data = ["2019/01/27 02:34:54.123", "2018/04/22 01:24:35.123"]
print("Input data: "+str(data))
p_out = session.protect(data, "SUCCESS_DATETIME")
print("Protected data: "+str(p_out))
unprotected_output = session.unprotect(p_out[0], "SUCCESS_DATETIME")
print("Unprotected data: "+str(unprotected_output))
```

Result

```
Protected data: (['2021/10/27 08:16:34.123000', '2021/01/20 07:06:15.123000'], (6, 6))
Unprotected data: (['2019/01/27 02:34:54.123000', '2018/04/22 01:24:35.123000'], (8, 8))
```

6 is the success return code for the protect operation of each element in the list.

8 is the success return code for the unprotect operation of each element in the list.

2.4.4.3.101 Mock Example - Reprotecting Date and Time String

This section describes how to use the *reprotect* API for reprotecting date and time string.

Warning: If you are retokenizing the data using the *reprotect* API, then the old data element and the new data element must have the same tokenization type. For example, if you have used the Datetime (YYYY-MM-DD HH:MM:SS MMM) data element to protect the data, then you must use only the Datetime (YYYY-MM-DD HH:MM:SS MMM) data element to reprotect the data.

Note: For information about the date formats supported by Protegrity tokenization methods, refer to the section *Protegrity Tokenization* in the *Protection Methods Reference Guide 9.1.0.0*.

Example: Input date and time in YYYY-MM-DD HH:MM:SS MMM format

In the following example, the *2019/01/27 02:34:54.123* date string is protected using the *SUCCESS_DATETIME* data element.

The protected input data, the old data element *SUCCESS_DATETIME*, and a new data element *SUCCESS_REPROTECT_DATETIME* are then passed as inputs to the *reprotect* API. The *reprotect* API first unprotects the protected input data using the old data element and then reprotects it using the new data element, as part of a single reprotect operation.

```
from appython import Protector
protector = Protector()
session = protector.create_session("ALL_USER")
p_out = session.protect("2019/01/27 02:34:54.123", "SUCCESS_DATETIME")
print("Protected date: "+str(p_out))
r_out = session.reprotect(p_out, "SUCCESS_DATETIME", "SUCCESS_REPROTECT_DATETIME")
print("Reprotected date: "+str(r_out))
```

Result

```
Protected date: 2021/10/27 08:16:34.123000
Reprotected date: 2022/06/24 02:27:30.123000
```

2.4.4.3.102 Mock Example - Reprotecting Bulk Date and Time Strings

This section describes how to use the *reprotect* API for reprotecting bulk date and time strings. You can pass bulk date and time strings as a list or a tuple.

Caution: The individual elements of the list or tuple must be of the same data type.

Warning: If you are retokenizing the data using the *reprotect* API, then the old data element and the new data element must have the same tokenization type. For example, if you have used the Datetime (YYYY-MM-DD HH:MM:SS MMM) data element to protect the data, then you must use only the Datetime (YYYY-MM-DD HH:MM:SS MMM) data element to reprotect the data.

Note: For information about the date formats supported by Protegrity tokenization methods, refer to the section *Protegrity Tokenization* in the *Protection Methods Reference Guide 9.1.0.0*.

Example

In the following example, the *2019/01/27 02:34:54.123* and *2018/04/22 01:24:35.123* date and time strings are used to create a list, which is used as the input data.

The input list is then protected using the *SUCCESS_DATETIME* data element.

The protected input data, the old data element *SUCCESS_DATETIME*, and a new data element *SUCCESS_REPROTECT_DATETIME* are then passed as inputs to the *reprotect* API. The *reprotect* API first unprotects the protected input data using the old data element and then reprotects it using the new data element, as part of a single reprotect operation.

```
from appython import Protector
protector = Protector()
session = protector.create_session("ALL_USER")
data = ["2019/01/27 02:34:54.123", "2018/04/22 01:24:35.123"]
print("Input data: "+str(data))
p_out = session.protect(data, "SUCCESS_DATETIME")
print("Protected data: "+str(p_out))
r_out = session.reprotect(p_out[0], "SUCCESS_DATETIME", "SUCCESS_REPROTECT_DATETIME")
print("Reprotected date: "+str(r_out))
```

Result

```
Protected data: (['2021/10/27 08:16:34.123000', '2021/01/20 07:06:15.123000'], (6, 6))
Reprotected date: (['2022/06/24 02:27:30.123000', '2021/09/17 01:17:11.123000'], (6, 6))
```

6 is the success return code for the protect operation of each element in the list.

2.4.4.3.103 Mock Example - Encrypting String Data

This section describes how to use the *protect* API for encrypting string data.

> **Note:**
> If you want to encrypt the data, then you must use bytes in the *encrypt_to* keyword.

Example: Input string data

In the following example, the *Protegrity1* string is used as the data, which is encrypted using the *SUCCESS_ENC* data element. Therefore, the *encrypt_to* parameter is passed as a keyword argument, and its value is set to *bytes*.

```
from appython import Protector
protector = Protector()
session = protector.create_session("ALL_USER")
output = session.protect("Protegrity1", "SUCCESS_ENC",
         encrypt_to=bytes)
print("Encrypted Data: %s" %output)
```

Result

```
Encrypted Data: b'1\x05\n\x07\n\n\x17\x19\x15\nB'
```

2.4.4.3.104 Mock Example - Encrypting Bulk String Data

This section describes how to use the *protect* API for encrypting bulk string data. You can pass bulk string data as a list or a tuple.

> **Caution:** The individual elements of the list or tuple must be of the same data type.

> **Note:**
> If you want to encrypt the data, then you must must use bytes in the *encrypt_to* keyword.

Example

In the following example, *protegrity1234*, *Protegrity1*, and *Protegrity56* strings are stored in a list and used as bulk data, which is encrypted using the *SUCCESS_ENC* data element. Therefore, the *encrypt_to* parameter is passed as a keyword argument, and its value is set to *bytes*.

```
from appython import Protector
protector = Protector()
session = protector.create_session("ALL_USER")
data = ["protegrity1234", "Protegrity1", "Protegrity56"]
p_out = session.protect(data, "SUCCESS_ENC",
        encrypt_to=bytes)
print("Encrypted Data: ")
print(p_out)
```

Result

```
Encrypted Data:
([b'\x11\x05\n\x07\n\n\x17\x19\x15\nBE\\F', b'1\x05\n\x07\n\n\x17\x19\x15\nB',
b'1\x05\n\x07\n\n\x17\x19\x15\nFA'], (6, 6, 6))
```

6 is the success return code for the protect operation of each element in the list.

2.4.4.3.105 Mock Example - Decrypting String Data

This section describes how to use the *unprotect* API for decrypting string data.

> **Note:**
> If you want to encrypt the data, then you must use bytes in the *encrypt_to* keyword.

Example: Input string data

In the following example, the *Protegrity1* string that was encrypted using the *SUCCESS_ENC* data element is now decrypted using the same data element. Therefore, the *decrypt_to* parameter is passed as a keyword argument, and its value is set to *str*.

```
from appython import Protector
protector = Protector()
session = protector.create_session("ALL_USER")
output = session.protect("Protegrity1", "SUCCESS_ENC",
         encrypt_to=bytes)
print("Encrypted Data: %s" %output)
org = session.unprotect(output, "SUCCESS_ENC",
      decrypt_to=str)
print("Decrypted Data: %s" %org)
```

Result

```
Encrypted Data: b'1\x05\n\x07\n\n\x17\x19\x15\nB'
Decrypted Data: Protegrity1
```

2.4.4.3.106 Mock Example - Decrypting Bulk String Data

This section describes how to use the *unprotect* API for decrypting bulk string data.

> **Note:**
> If you want to encrypt the data, then you must use bytes in the *encrypt_to* keyword.

Example

In the following example, *protegrity1234*, *Protegrity1*, and *Protegrity56* strings are stored in a list and used as bulk data, which is encrypted using the *SUCCESS_STR* data element. The bulk string data is then decrypted using the same data element. Therefore, the *decrypt_to* parameter is passed as a keyword argument, and its value is set to *str*.

```
from appython import Protector
protector = Protector()
session = protector.create_session("ALL_USER")
data = ["protegrity1234", "Protegrity1", "Protegrity56"]
p_out = session.protect(data, "SUCCESS_STR", encrypt_to=bytes)
print("Encrypted Data: ")
print(p_out)
out = session.unprotect(p_out[0], "SUCCESS_STR", decrypt_to=str)
print("Decrypted Data: ")
print(out)
```

Result

```
Encrypted Data:
([b'pJPqrjJEqLXHaO', b'6JPqrjJEqLX', b'6JPqrjJEqLl5'], (6, 6, 6))
Decrypted Data:
(['protegrity1234', 'Protegrity1', 'Protegrity56'], (8, 8, 8))
```

6 is the success return code for the protect operation of each element in the list.

8 is the success return code for the unprotect operation of each element in the list.

2.4.4.3.107 Mock Example - Encrypting Integer Data

This section describes how to use the *protect* API for encrypting integer data.

> **Note:**
> If you want to encrypt the data, then you must use bytes in the *encrypt_to* keyword.

Example

In the following example, *21* is used as the integer data, which is encrypted using the *SUCCESS_ENC* data element. Therefore, the *encrypt_to* parameter is passed as a keyword argument, and its value is set to *bytes*.

```
from appython import Protector
protector = Protector()
session = protector.create_session("ALL_USER")
output = session.protect(21, "SUCCESS_ENC", encrypt_to=bytes)
print("Encrypted Data: %s" %output)
```

Result

```
Encrypted Data: b'twes'
```

2.4.4.3.108 Mock Example - Encrypting Bulk Integer Data

This section describes how to use the *protect* API for encrypting bulk integer data. You can pass bulk integer data as a list or a tuple.

> **Note:**
> If you want to encrypt the data, then you must use bytes in the *encrypt_to* keyword.

Example

In the following example, *21*, *42*, and *55* integers are stored in a list and used as bulk data, which is encrypted using the *SUCCESS_ENC* data element. Therefore, the *encrypt_to* parameter is passed as a keyword argument, and its value is set to *bytes*.

```
from appython import Protector
protector = Protector()
session = protector.create_session("ALL_USER")
data = [21, 42, 55]
p_out = session.protect(data, "SUCCESS_ENC", encrypt_to=bytes)
print("Encrypted Data: ")
print(p_out)
```

Result

```
Encrypted Data:
([b'twes', b'Kwes', b'Vwes'], (6, 6, 6))
```

6 is the success return code for the protect operation of each element in the list.

2.4.4.3.109 Mock Example - Decrypting Integer Data

This section describes how to use the *unprotect* API for decrypting integer data.

> **Note:**
> If you want to encrypt the data, then you must use bytes in the *encrypt_to* keyword.

Example

In the following example, the integer data *21* that was encrypted using the *SUCCESS_ENC* data element is now decrypted using the same data element. Therefore, the *decrypt_to* parameter is passed as a keyword argument, and its value is set to *int*.

```
from appython import Protector
protector = Protector()
session = protector.create_session("ALL_USER")
output = session.protect(21, "SUCCESS_ENC", encrypt_to=bytes)
print("Encrypted Data: %s" %output)
org = session.unprotect(output, "SUCCESS_ENC", decrypt_to=int)
print("Decrypted Data: %s" %org)
```

Result

```
Encrypted Data: b'twes'
Decrypted Data: 21
```

2.4.4.3.110 Mock Example - Decrypting Bulk Integer Data

This section descibes how to use the *unprotect* API for decrypting bulk integer data.

> **Note:**
> If you want to encrypt the data,then you must use bytes in the *encrypt_to* keyword.

Example

In the following example, *21*, *42*, and *55* integers are stored in a list and used as bulk data, which is encrypted using the *SUCCESS_ENC* data element. The bulk integer data is then decrypted using the same data element. Therefore, the *decrypt_to* parameter is passed as a keyword argument, and its value is set to *int*.

```
from appython import Protector
protector = Protector()
session = protector.create_session("ALL_USER")
data = [21, 42, 55]
p_out = session.protect(data, "SUCCESS_ENC", encrypt_to=bytes)
print("Encrypted Data: ")
print(p_out)
out = session.unprotect(p_out[0], "SUCCESS_ENC", decrypt_to=int)
print("Decrypted Data: ")
print(out)
```

Result

```
Encrypted Data:
([b'twes', b'Kwes', b'Vwes'], (6, 6, 6))
Decrypted Data:
([21, 42, 55], (6, 6, 6))
```

6 is the success return code for the protect operation of each element in the list.

8 is the success return code for the unprotect operation of each element in the list.

2.4.4.3.111 Mock Example - Encrypting Long Data

This section describes how to use the *protect* API for encrypting long data.

> **Note:**
> If you want to encrypt the data, then you must use bytes in the *encrypt_to* keyword.

Example

In the following example, *1376235139103947* is used as the long data, which is encrypted using the *SUCCESS_ENC* data element. Therefore, the *encrypt_to* parameter is passed as a keyword argument, and its value is set to *bytes*.

```
from appython import Protector
protector = Protector()
session = protector.create_session("ALL_USER")
output = session.protect(1376235139103947, "SUCCESS_ENC", encrypt_to=bytes)
print("Encrypted Data: %s" %output)
```

Result

```
Encrypted Data: b'\xaa\x8b\xf2\xc5\xc2\x8eap'
```

2.4.4.3.112 Mock Example - Encrypting Bulk Long Data

This section describes how to use the *protect* API for encrypting bulk long data. You can pass bulk long data as a list or a tuple.

> **Caution:** The individual elements of the list or tuple must be of the same data type.

> **Note:**

If you want to encrypt the data, then you must use bytes in the *encrypt_to* keyword.

Example

In the following example, *1376235139103947*, *2396235839173981*, and *9371234126176985* long data are stored in a list and used as bulk data, which is encrypted using the *SUCCESS_ENC* data element. Therefore, the *encrypt_to* parameter is passed as a keyword argument, and its value is set to *bytes*.

```
from appython import Protector
protector = Protector()
session = protector.create_session("ALL_USER")
data = [1376235139103947, 2396235839173981, 9371234126176985]
p_out = session.protect(data, "SUCCESS_ENC", encrypt_to=bytes)
print("Encrypted Data: ")
print(p_out)
```

Result

```
Encrypted Data:
([b'\xaa\x8b\xf2\xc5\xc2\x8eap', b'<\x82\x98R2\xeemp', b'\xb8\xd5W\ny&Dp'], (6, 6, 6))
```

6 is the success return code for the protect operation of each element in the list.

2.4.4.3.113 Mock Example - Decrypting Long Data

This section describes how to use the *unprotect* API for decrypting long data.

Note:
If you want to encrypt the data, then you must use bytes in the *encrypt_to* keyword.

Example

In the following example, the long data *1376235139103947* that was encrypted using the *SUCCESS_ENC* data element is now decrypted using the same data element. Therefore, the *decrypt_to* parameter is passed as a keyword argument, and its value is set to *long*.

```
from appython import Protector
protector = Protector()
session = protector.create_session("ALL_USER")
output = session.protect(1376235139103947, "SUCCESS_ENC", encrypt_to=bytes)
print("Encrypted Data: %s" %output)
org = session.unprotect(output, "SUCCESS_ENC", decrypt_to=int)
print("Decrypted Data: %s" %org)
```

Result

```
Encrypted Data: b'\xaa\x8b\xf2\xc5\xc2\x8eap'
Decrypted Data: 1376235139103947
```

2.4.4.3.114 Mock Example - Decrypting Bulk Long Data

This section describes how to use the *unprotect* API for decrypting bulk long data.

Note:
If you want to encrypt the data, then you must use bytes in the *encrypt_to* keyword.

Example

In the following example, *1376235139103947*, *2396235839173981*, and *9371234126176985* long data are stored in a list and used as bulk data, which is encrypted using the *SUCCESS_ENC* data element. The bulk long data is then decrypted using the same data element. Therefore, the *decrypt_to* parameter is passed as a keyword argument, and its value is set to *long*.

```
from appython import Protector
protector = Protector()
session = protector.create_session("ALL_USER")
data = [1376235139103947, 2396235839173981, 9371234126176985]
p_out = session.protect(data, "SUCCESS_ENC", encrypt_to=bytes)
print("Encrypted Data: ")
print(p_out)
out = session.unprotect(p_out[0], "SUCCESS_ENC", decrypt_to=int)
print("Decrypted Data: ")
print(out)
```

Result

```
Encrypted Data:
([b'\xaa\x8b\xf2\xc5\xc2\x8eap', b'<\x82\x98R2\xeemp', b'\xb8\xd5W\ny&Dp'], (6, 6, 6))
Decrypted Data:
([1376235139103947, 2396235839173981, 9371234126176985], (6, 6, 6))
```

6 is the success return code for the protect operation of each element in the list.

8 is the success return code for the unprotect operation of each element in the list.

2.4.4.3.115 Mock Example - Encrypting Float Data

This section describes how to use the *protect* API for encrypting float data.

> **Note:**
> If you want to encrypt the data, then you must use bytes in the *encrypt_to* keyword.

Example

In the following example, *22.5* is used as the float data, which is encrypted using the *SUCCESS_ENC* data element. Therefore, the *encrypt_to* parameter is passed as a keyword argument, and its value is set to *bytes*.

```
from appython import Protector
protector = Protector()
session = protector.create_session("ALL_USER")
output = session.protect(22.5, "SUCCESS_ENC", encrypt_to=bytes)
print("Encrypted Data: %s" %output)
```

Result

```
Encrypted Data: b'SEKF'
```

2.4.4.3.116 Mock Example - Encrypting Bulk Float Data

This section describes how to use the *protect* API for encrypting bulk float data. You can pass bulk float data as a list or a tuple.

> **Caution:** The individual elements of the list or tuple must be of the same data type.

> **Note:**

If you want to encrypt the data, then you must use bytes in the *encrypt_to* keyword.

Example

In the following example, *22.5*, *48.93*, and *94.14* float data are stored in a list and used as bulk data, which is encrypted using the *SUCCESS_ENC* data element. Therefore, the *encrypt_to* parameter is passed as a keyword argument, and its value is set to *bytes*.

```
from appython import Protector
protector = Protector()
session = protector.create_session("ALL_USER")
data = [22.5, 48.93, 94.31]
p_out = session.protect(data, "SUCCESS_ENC", encrypt_to=bytes)
print("Encrypted Data: ")
print(p_out)
```

Result

```
Encrypted Data:
([b'SEKF', b'UOKJ\\', b'XCK@^'], (6, 6, 6))
```

6 is the success return code for the protect operation of each element in the list.

2.4.4.3.117 Mock Example - Decrypting Float Data

This section describes how to use the *unprotect* API for decrypting float data.

Note:
If you want to encrypt the data, then you must use bytes in the *encrypt_to* keyword.

Example

In the following example, the float data *22.5* that was encrypted using the *SUCCESS_ENC* data element is now decrypted using the same data element. Therefore, the *decrypt_to* parameter is passed as a keyword argument and its value is set to *float*.

```
from appython import Protector
protector = Protector()
session = protector.create_session("ALL_USER")
output = session.protect(22.5, "SUCCESS_ENC", encrypt_to=bytes)
print("Encrypted Data: %s" %output)
org = session.unprotect(output, "SUCCESS_ENC", decrypt_to=float)
print("Decrypted Data: %s" %org)
```

Result

```
Encrypted Data: b'SEKF'
Decrypted Data: 22.5
```

2.4.4.3.118 Mock Example - Decrypting Bulk Float Data

This section describes how to use the *unprotect* API for decrypting bulk float data.

Note:
If you want to encrypt the data, then you must use bytes in the *encrypt_to* keyword.

Example

In the following example, *22.5*, *48.93*, and *94.14* float data are stored in a list and used as bulk data, which is encrypted using the *SUCCESS_ENC* data element. The bulk float data is then decrypted using the same data element. Therefore, the *decrypt_to* parameter is passed as a keyword argument and its value is set to *float*.

```
from appython import Protector
protector = Protector()
session = protector.create_session("ALL_USER")
data = [22.5, 48.93, 94.31]
p_out = session.protect(data, "SUCCESS_ENC", encrypt_to=bytes)
print("Encrypted Data: ")
print(p_out)
out = session.unprotect(p_out[0], "SUCCESS_ENC", decrypt_to=float)
print("Decrypted Data: ")
print(out)
```

Result

```
Encrypted Data:
([b'SEKF', b'UOKJ\\', b'XCK@^'], (6, 6, 6))
Decrypted Data:
([22.5, 48.93, 94.31], (6, 6, 6))
```

6 is the success return code for the protect operation of each element in the list.

8 is the success return code for the unprotect operation of each element in the list.

2.4.4.3.119 Mock Example - Encrypting Bytes Data

This section describes how to use the *protect* API for encrypting bytes data.

> **Note:**
> If you want to encrypt the data, then you must use bytes in the *encrypt_to* keyword.

Example

In the following example, *Protegrity1* string is first converted to bytes using the Python *bytes()* method. The bytes data is then encrypted using the *SUCCESS_BYTE* data element. Therefore, the *encrypt_to* parameter is passed as a keyword argument and its value is set to *bytes*.

```
from appython import Protector
protector = Protector()
session = protector.create_session("ALL_USER")
data=bytes("Protegrity1", encoding="utf-8")
p_out = session.protect(data, "SUCCESS_BYTE", encrypt_to=bytes)
print("Encrypted Data: %s" %p_out)
```

Result

```
Encrypted Data: b'6JPqrjJEqLX'
```

2.4.4.3.120 Mock Example - Encrypting Bulk Bytes Data

This section describes how to use the *protect* API for encrypting bulk bytes data. You can pass bulk bytes data as a list or a tuple.

> **Caution:** The individual elements of the list or tuple must be of the same data type.

> **Note:**
> If you want to encrypt the data, then you must use bytes in the *encrypt_to* keyword.

Example

In the following example, *Protegrity1* string is first converted to bytes using the Python *bytes()* method. The bytes data is then repeated five times in a list and used as bulk data, which is encrypted using the *SUCCESS_BYTE* data element. Therefore, the *encrypt_to* parameter is passed as a keyword argument and its value is set to *bytes*.

```
from appython import Protector
protector = Protector()
session = protector.create_session("ALL_USER")
data=[bytes("Protegrity1")]*5
p_out = session.protect(data, "SUCCESS_BYTE", encrypt_to=bytes)
print("Encrypted Data: ")
print(p_out)
```

Result

```
Encrypted Data:
([b'6JPqrjJEqLX', b'6JPqrjJEqLX', b'6JPqrjJEqLX', b'6JPqrjJEqLX', b'6JPqrjJEqLX'], (6,
6, 6, 6, 6))
```

6 is the success return code for the protect operation of each element in the list.

2.4.4.3.121 Mock Example - Decrypting Bytes Data

This section describes how to use the *protect* API for decrypting bytes data.

Example

In the following example, *Protegrity1* string is first converted to bytes using the Python *bytes()* method. The bytes data is then encrypted using the *SUCCESS_BYTE* data element. Therefore, the *encrypt_to* parameter is passed as a keyword argument, and its value is set to *bytes*.

The encrypted data is then decrypted using the same data element. Therefore, the *decrypt_to* parameter is passed as a keyword argument and its value is set to *bytes*.

```
from appython import Protector
protector = Protector()
session = protector.create_session("ALL_USER")
data=bytes("Protegrity1", encoding="utf-8")
p_out = session.protect(data, "SUCCESS_BYTE", encrypt_to=bytes)
print("Encrypted Data: %s" %p_out)
org = session.unprotect(p_out, "SUCCESS_BYTE", decrypt_to=bytes)
print("Decrypted Data: %s" org)
```

Result

```
Encrypted Data: b'6JPqrjJEqLX'
Decrypted Data: %s b'Protegrity1'
```

2.4.4.3.122 Mock Example - Decrypting Bulk Bytes Data

This section describes how to use the *protect* API for encrypting bulk bytes data. You can pass bulk bytes data as a list or a tuple.

> **Caution:** The individual elements of the list or tuple must be of the same data type.

Example

In the following example, *Protegrity1* string is first converted to bytes using the Python *bytes()* method. The bytes data is then repeated five times in a list and used as bulk data, which is encrypted using the *SUCCESS_BYTE* data element. Therefore, the *encrypt_to* parameter is passed as a keyword argument and its value is set to *bytes*.

The encrypted bulk data is then decrypted using the same data element. Therefore, the *decrypt_to* parameter is passed as a keyword argument and its value is set to *bytes*.

```
from appython import Protector
protector = Protector()
session = protector.create_session("ALL_USER")
data=[bytes("Protegrity1")]*5
p_out = session.protect(data, "SUCCESS_BYTE", encrypt_to=bytes)
print("Encrypted Data: ")
print(p_out)
org = session.unprotect(p_out[0], "SUCCESS_BYTE", decrypt_to=bytes)
print("Decrypted Data: ")
print(org)
```

Result

```
Encrypted Data:
([b'6JPqrjJEqLX', b'6JPqrjJEqLX', b'6JPqrjJEqLX', b'6JPqrjJEqLX', b'6JPqrjJEqLX'], (6,
6, 6, 6, 6))
Decrypted Data:
([b'Protegrity1', b'Protegrity1', b'Protegrity1', b'Protegrity1', b'Protegrity1'], (8,
8, 8, 8, 8))
```

6 is the success return code for the protect operation of each element in the list.

8 is the success return code for the unprotect operation of each element in the list.

2.4.4.3.123 Mock Example - Re-encrypting Bytes Data

This section describes how to use the *reprotect* API for re-encrypting bytes data.

> **Warning:** If you are using the *reprotect* API, then the old data element and the new data element must be of the same protection method. For example, if you have used AES256 data element to protect the data, then you must use only AES256 data element to reprotect the data.

Example

In the following example, *Protegrity1* string is first converted to bytes using the Python *bytes()* method. The bytes data is then encrypted using the *SUCCESS_BYTE* data element. Therefore, the *encrypt_to* parameter is passed as a keyword argument and its value is set to *bytes*.

The protected input data, the old data element *SUCCESS_BYTE*, and a new data element *SUCCESS_REPROTECT_BYTE* are then passed as inputs to the *reprotect* API. The *reprotect* API first decrypts the protected input data using the old data element and then re-encrypts it using the new data element, as part of a single reprotect operation. Therefore, the *encrypt_to* parameter is passed as a keyword argument and its value is set to *bytes*.

```
from appython import Protector
protector = Protector()
```

```
session = protector.create_session("ALL_USER")
data=bytes("Protegrity1", encoding="utf-8")
p_out = session.protect(data, "SUCCESS_BYTE", encrypt_to=bytes)
print("Encrypted Data: %s" %p_out)
r_out = session.reprotect(p_out, "SUCCESS_BYTE", "SUCCESS_REPROTECT_BYTE",
encrypt_to=bytes)
print("Re-encrypted Data: %s" %r_out)
```

Result

```
Encrypted Data: b'6JPqrjJEqLX'
Re-encrypted Data: b'JQbePhQ2eGC'
```

2.4.4.3.124 Example - Re-Encrypting Bulk Bytes Data

This section describes how to use the *reprotect* API for re-encrypting bulk bytes data. You can pass bulk bytes data as a list or a tuple.

Caution: The individual elements of the list or tuple must be of the same data type.

Warning: If you are using the *reprotect* API, then the old data element and the new data element must be of the same protection method. For example, if you have used AES256 data element to protect the data, then you must use only AES256 data element to reprotect the data.

Example

In the following example, *Protegrity1* string is first converted to bytes using the Python *bytes()* method. The bytes data is then repeated five times in a list and used as bulk data, which is encrypted using the *SUCCESS_BYTE* data element. Therefore, the *encrypt_to* parameter is passed as a keyword argument and its value is set to *bytes*.

The encrypted input data, the old data element *SUCCESS_BYTE*, and a new data element *SUCCESS_REPROTECT_BYTE* are then passed as inputs to the *reprotect* API. The *reprotect* API first decrypts the protected input data using the old data element and then re-encrypts it using the new data element, as part of a single reprotect operation. Therefore, the *encrypt_to* parameter is passed as a keyword argument and its value is set to *bytes*.

```
from appython import Protector
protector = Protector()
session = protector.create_session("ALL_USER")
data=[bytes("Protegrity1")]*5
p_out = session.protect(data, "SUCCESS_BYTE", encrypt_to=bytes)
print("Encrypted Data: ")
print(p_out)
r_out = session.reprotect(p_out[0], "SUCCESS_BYTE", "SUCCESS_REPROTECT_BYTE",
encrypt_to=bytes)
print("Re-encrypted Data: ")
print(r_out)
```

Result

```
Encrypted Data:
([b'6JPqrjJEqLX', b'6JPqrjJEqLX', b'6JPqrjJEqLX', b'6JPqrjJEqLX', b'6JPqrjJEqLX'], (6,
6, 6, 6, 6))
Re-encrypted Data:
([b'JQbePhQ2eGC', b'JQbePhQ2eGC', b'JQbePhQ2eGC', b'JQbePhQ2eGC', b'JQbePhQ2eGC'], (6,
6, 6, 6, 6))
```

2.4.4.4 Using Sample Data Elements for Simulating Auxiliary API Scenarios

This section describes how to use the sample data elements for simulating the following auxiliary API scenarios:

- Retrieving the default data element
- Retrieving the key ID for a specific data element
- Checking access permissions with success output

- Checking access permissions with failure output

> **Note:** In the mock implementation, you must pass the *ALL_USER* user name as an argument to the *create_session* API for creating a session.

2.4.4.4.1 Mock Example - Success Scenario for Checking Access Permissions

This section lists the success scenario when you check the access permission status of the user for a specified data element.

Example

In the following example, the *check_access* API returns *True* when you check the permission of *User1* for protecting the data using the *SUCCESS_CHECK_ACCESS* data element.

```
from appython import Protector
from appython import CheckAccessType
protector = Protector()
session = protector.create_session("ALL_USER")
print(session.check_access("SUCCESS_CHECK_ACCESS", CheckAccessType.PROTECT))
```

Result

```
True
```

2.4.4.4.2 Mock Example - Failure Scenario for Checking Access Permissions

This section lists the failure scenario when you check the access permission status of the user for a specified data element.

Example

In the following example, the *check_access* API returns *True* when you check the permission of *User1* for protecting the data using the *FAIL_CHECK_ACCESS* data element.

```
from appython import Protector
from appython import CheckAccessType
protector = Protector()
session = protector.create_session("ALL_USER")
print(session.check_access("FAIL_CHECK_ACCESS", CheckAccessType.PROTECT))
```

Result

```
False
```

2.4.4.4.3 Mock Example - Retrieving Default Data Element

This section describes how to use the *get_default_de* API to retrieve the default data element for the policy provided as an input parameter. A data element becomes a default for a policy when you select it as default during policy creation.

Example

In the following example, the *get_default_de* API returns the default data element when you pass the policy name *POLICY_NAME* as an input parameter.

```
from appython import Protector
protector = Protector()
session = protector.create_session("ALL_USER")
default_de = session.get_default_de("POLICY_NAME")
print("Default data element: "+default_de)
```

Result

```
Default data element: ALPHANUM
```

2.4.4.4.4 Mock Example - Retrieving Key ID for Data Element

This section describes how to use the *get_current_key_id_for_dataelement* API to return the key ID for a data element that is passed as an input parameter. The data elements can be of type 3DES, AES-128, or AES-256.

Example

In the following example, the *get_current_key_id_for_dataelement* API returns the key ID when you pass the *SUCCESS_PROTECT_ENC_INT* as an input parameter.

```
from appython import Protector
protector = Protector()
session = protector.create_session("ALL_USER")
current_key = session.get_current_key_id_for_dataelement("SUCCESS_PROTECT_ENC_INT")
print(current_key)
```

Result

```
123
```

2.4.4.5 Using Sample Data Elements for Simulating Error Scenarios

This section describes how to use the sample data elements for simulating the error scenarios while protecting, unprotecting, and reprotecting the data.

> **Note:** In the mock implementation, you must pass the *ALL_USER* user name as an argument to the *create_session* API for creating a session.

2.4.4.5.1 Mock Example - Invalid User Exception

This section describes an example of the scenario if a user who is not defined in a policy is used to protect single or bulk data.

Example: Single Data

In the following example, the *Protegrity1* string is used as the data, which is being protected using the *EXCEPTION_INVALID_USER* data element.

```
from appython import Protector
from appython.exceptions import ProtectError
protector = Protector()
session = protector.create_session("ALL_USER")
try:
    output = session.protect("Protegrity1", "EXCEPTION_INVALID_USER")
    print("protect:    "+output)
except ProtectError as error:
    print(error)
```

Result

```
1, The username could not be found in the policy in shared memory.
```

Example: Bulk Data

In the following example, the *Protegrity1* string is repeated five times in a list, which is used as the input bulk data. The input data is being protected using the *EXCEPTION_INVALID_USER* data element.

```
from appython import Protector
protector = Protector()
session = protector.create_session("ALL_USER")
data = ["Protegrity1"]*5
output = session.protect(data, "EXCEPTION_INVALID_USER")
print(output)
```

Result

```
([None, None, None, None, None], ('1', '1', '1', '1', '1'))
```

2.4.4.5.2 Mock Example - Invalid Data Element Exception

This section describes an example of the scenario if a data element that is not defined in a policy is used to protect single or bulk data.

Example: Single Data

In the following example, the *Protegrity1* string is used as the data, which is being protected using the *EXCEPTION_INVALID_DE* data element.

```
from appython import Protector
from appython.exceptions import ProtectError
protector = Protector()
session = protector.create_session("ALL_USER")
try:
    output = session.protect("Protegrity1", "EXCEPTION_INVALID_DE")
    print("protect:    "+output)
except ProtectError as error:
    print(error)
```

Result

```
2, The data element could not be found in the policy in shared memory.
```

Example: Bulk Data

In the following example, the *Protegrity1* string is repeated five times in a list, which is used as the input bulk data. The input data is being protected using the *EXCEPTION_INVALID_DE* data element.

```
from appython import Protector
protector = Protector()
session = protector.create_session("ALL_USER")
data = ["Protegrity1"]*5
output = session.protect(data, "EXCEPTION_INVALID_DE")
print(output)
```

Result

```
([None, None, None, None, None], ('2', '2', '2', '2', '2'))
```

2.4.4.5.3 Mock Example - External Tweak is Null

This section describes an example of the scenario if a null external tweak is used to protect single or bulk data.

Example: Single Data

In the following example, the *Protegrity1* string is used as the data, which is being protected using the *EXCEPTION_TWEAK_IS_NULL* data element.

```
from appython import Protector
protector = Protector()
session = protector.create_session("ALL_USER")
try:
    output = session.protect("Protegrity1", "EXCEPTION_TWEAK_IS_NULL")
    print("protect:    "+output)
except ProtectError as error:
    print(error)
```

Result

```
4, Tweak is null.
```

Example: Bulk Data

In the following example, the *Protegrity1* string is repeated five times in a list, which is used as the input bulk data. The input data is being protected using the *EXCEPTION_TWEAK_IS_NULL* data element.

```
from appython import Protector
from appython.exceptions import ProtectError
protector = Protector()
session = protector.create_session("ALL_USER")
data = ["Protegrity1"]*5
output = session.protect(data, "EXCEPTION_TWEAK_IS_NULL")
print(output)
```

Result

```
([None, None, None, None, None], ('4', '4', '4', '4', '4'))
```

2.4.4.5.4 Mock Example - Data Too Short

This section describes an example of the scenario if the data to be protected or unprotected is too short.

Example: Single Data

In the following example, the *Protegrity1* string is used as the data, which is being protected using the *DATA_TOO_SHORT* data element.

```
from appython import Protector
from appython.exceptions import ProtectError
protector = Protector()
session = protector.create_session("ALL_USER")
try:
    output = session.protect("Protegrity1", "DATA_TOO_SHORT")
    print("protect:    "+output)
except ProtectError as error:
    print(error)
```

Result

```
22, Data is too short to be protected/unprotected.
```

Example: Bulk Data

In the following example, the *Protegrity1* string is repeated five times in a list, which is used as the input bulk data. The input data is being protected using the *DATA_TOO_SHORT* data element.

```
from appython import Protector
protector = Protector()
session = protector.create_session("ALL_USER")
```

```
data = ["Protegrity1"]*5
output = session.protect(data, "DATA_TOO_SHORT")
print(output)
```

Result

```
([None, None, None, None, None], ('22', '22', '22', '22', '22'))
```

2.4.4.5.5 Mock Example - Long User Name

This section describes an example of the scenario if the name of the user, who is protecting or unprotecting the data, is too long.

Example: Single Data

In the following example, the *Protegrity1* string is used as the data, which is being protected using the *USER_TOO_LONG* data element.

```
from appython import Protector
from appython.exceptions import ProtectError
protector = Protector()
session = protector.create_session("ALL_USER")
try:
    output = session.protect("Protegrity1", "USER_TOO_LONG")
    print("protect:    "+output)
except ProtectError as error:
    print(error)
```

Result

```
25, Username too long.
```

Example: Bulk Data

In the following example, the *Protegrity1* string is repeated five times in a list, which is used as the input bulk data. The input data is being protected using the *USER_TOO_LONG* data element.

```
from appython import Protector
protector = Protector()
session = protector.create_session("ALL_USER")
data = ["Protegrity1"]*5
output = session.protect(data, "USER_TOO_LONG")
print(output)
```

Result

```
([None, None, None, None, None], ('25', '25', '25', '25', '25'))
```

2.4.4.5.6 Mock Example - Unsupported Algorithm

This section describes an example of the scenario if the protection method used to protect the data is not supported by the API.

Example: Single Data

In the following example, the *Protegrity1* string is used as the data, which is being protected using the *EXCEPTION_UNSUPPORTED_ALGORITHM* data element.

```
from appython import Protector
from appython.exceptions import ProtectError
protector = Protector()
session = protector.create_session("ALL_USER")
try:
    output = session.protect("Protegrity1", "EXCEPTION_UNSUPPORTED_ALGORITHM")
    print("protect:    "+output)
```

```
except ProtectError as error:
    print(error)
```

Result

```
26, Unsupported algorithm or unsupported action for the specific data element.
```

Example: Bulk Data

In the following example, the *Protegrity1* string is repeated five times in a list, which is used as the input bulk data. The input data is being protected using the *EXCEPTION_UNSUPPORTED_ALGORITHM* data element.

```
from appython import Protector
protector = Protector()
session = protector.create_session("ALL_USER")
data = ["Protegrity1"]*5
output = session.protect(data, "EXCEPTION_UNSUPPORTED_ALGORITHM")
print(output)
```

Result

```
([None, None, None, None, None], ('26', '26', '26', '26', '26'))
```

2.4.4.5.7 Mock Example - Empty Policy

This section describes an example of the scenario if the data is protected without the policy being present in shared memory.

Example: Single Data

In the following example, the *Protegrity1* string is used as the data, which is being protected using the *EMPTY_POLICY* data element.

```
from appython import Protector
from appython.exceptions import ProtectError
protector = Protector()
session = protector.create_session("ALL_USER")
try:
    output = session.protect("Protegrity1", "EMPTY_POLICY")
    print("protect:    "+output)
except ProtectError as error:
    print(error)
```

Result

```
31, The policy in shared memory is empty.
```

Example: Bulk Data

In the following example, the *Protegrity1* string is repeated five times in a list, which is used as the input bulk data. The input data is being protected using the *EMPTY_POLICY* data element.

```
from appython import Protector
protector = Protector()
session = protector.create_session("ALL_USER")
data = ["Protegrity1"]*5
output = session.protect(data, "EMPTY_POLICY")
print(output)
```

Result

```
([None, None, None, None, None], ('31', '31', '31', '31', '31'))
```

2.4.4.5.8 Mock Example - License Expired

This section describes an example of the scenario if the protector license has expired.

Example: Single Data

In the following example, the *Protegrity1* string is used as the data, which is being protected using the *LICENSE_EXPIRED* data element.

```
from appython import Protector
from appython.exceptions import ProtectError
protector = Protector()
session = protector.create_session("ALL_USER")
try:
    output = session.protect("Protegrity1", "LICENSE_EXPIRED")
    print("protect:    "+output)
except ProtectError as error:
    print(error)
```

Result

```
40, No valid license or current date is beyond the license expiration date.
```

Example: Bulk Data

In the following example, the *Protegrity1* string is repeated five times in a list, which is used as the input bulk data. The input data is being protected using the *LICENSE_EXPIRED* data element.

```
from appython import Protector
protector = Protector()
session = protector.create_session("ALL_USER")
data = ["Protegrity1"]*5
output = session.protect(data, "LICENSE_EXPIRED")
print(output)
```

Result

```
([None, None, None, None, None], ('40', '40', '40', '40', '40'))
```

2.4.4.5.9 Mock Example - Invalid Input

This section describes an example of the scenario if the data to be protected is invalid.

Example: Single Data

In the following example, the *Protegrity1* string is used as the data, which is being protected using the *INPUT_NOT_VALID* data element.

```
from appython import Protector
from appython.exceptions import ProtectError
protector = Protector()
session = protector.create_session("ALL_USER")
try:
    output = session.protect("Protegrity1", "INPUT_NOT_VALID")
    print("protect:    "+output)
except ProtectError as error:
    print(error)
```

Result

```
44, The content of the input data is not valid.
```

Example: Bulk Data

In the following example, the *Protegrity1* string is repeated five times in a list, which is used as the input bulk data. The input data is being protected using the *INPUT_NOT_VALID* data element.

```
from appython import Protector
protector = Protector()
session = protector.create_session("ALL_USER")
data = ["Protegrity1"]*5
output = session.protect(data, "INPUT_NOT_VALID")
print(output)
```

Result

```
([None, None, None, None, None], ('44', '44', '44', '44', '44'))
```

2.4.4.5.10 Mock Example - Reprotecting Data with Heterogenous Data Elements

This section describes the error when the new data element used to reprotect the data does not have the same tokenization type or the protection method as that of the old data element.

Example: Single Data

In the following example, the *Protegrity1* string is used as the data, which is being protected using the *SUCCESS_STR* data element.

The protected input data, the old data element *SUCCESS_STR*, and a new data element *REPROTECT_HETERO_STR* are then passed as inputs to the *reprotect* API. The *reprotect* API returns an error as the old and new data elements do not have the same tokenization type or the protection method.

```
from appython import Protector
from appython.exceptions import ProtectError
protector = Protector()
session = protector.create_session("ALL_USER")
output = session.protect("Protegrity1", "SUCCESS_STR" )
try:
    org = session.reprotect(output, "SUCCESS_STR", "REPROTECT_HETERO_STR" )
    print("Reprotected data:    "+org)
except Exception as error:
    print(error)
```

Result

```
26, Unsupported algorithm or unsupported action for the specific data element.
```

Example: Bulk Data

In the following example, the *Protegrity1* string is repeated five times in a list, which is used as the input bulk data. The input data is being protected using the *SUCCESS_STR* data element.

The protected input data, the old data element *SUCCESS_STR*, and a new data element *REPROTECT_HETERO_STR* are then passed as inputs to the *reprotect* API. The *reprotect* API returns an error as the old and new data elements do not have the same tokenization type or the protection method.

```
from appython import Protector
protector = Protector()
session = protector.create_session("ALL_USER")
data=["Protegrity1"]*5
output = session.protect(data, "SUCCESS_STR" )
try:
    org = session.reprotect(output[0], "SUCCESS_STR", "REPROTECT_HETERO_STR" )
    print("Reprotected data:")
    print(org)
except Exception as error:
    print(error)
```

Result

```
26, Unsupported algorithm or unsupported action for the specific data element.
```

2.4.4.6 Using Sample Users for Simulating Error Scenarios

This section describes how to use sample users for simulating the user-related error scenarios while protecting, unprotecting, and reprotecting the data.

2.4.4.6.1 Mock Example - No Protect User

This section describes an example of the scenario in which a user does not have privileges to protect data.

Example: Single Data

In the following example, the *NO_PROTECT_USER* user is used to try and protect the *Protegrity1* string using the *SUCCESS_STR* data element.

```
from appython import Protector
from appython.exceptions import ProtectError
protector = Protector()
session = protector.create_session("NO_PROTECT_USER")
try:
    output = session.protect("Protegrity1", "SUCCESS_STR")
    print("Protected data:    "+output)
except ProtectError as error:
    print(error)
```

Result

```
3, The user does not have the appropriate permissions to perform the requested
operation.
```

Example: Bulk Data

In the following example, the *Protegrity1* string is repeated five times in a list, which is used as the input bulk data. The *NO_PROTECT_USER* is used to try and protect the input data using the *SUCCESS_STR* data element.

```
from appython import Protector
from appython.exceptions import ProtectError
protector = Protector()
session = protector.create_session("NO_PROTECT_USER")
data = ["Protegrity1"]*5
output = session.protect(data, "SUCCESS_STR")
print(output)
```

Result

```
([None, None, None, None, None], ('3', '3', '3', '3', '3'))
```

2.4.4.6.2 Mock Example - No Reprotect User

This section describes an example of the scenario in which a user does not have privileges to reprotect data.

Example: Single Data

In the following example, the *NO_REPROTECT_USER* user is used to try and reprotect the *Protegrity1* string using the *SUCCESS_REPROTECT_STR* data element.

```
from appython import Protector
from appython.exceptions import ReprotectError
protector = Protector()
session = protector.create_session("NO_REPROTECT_USER")
```

```
try:
    org = session.reprotect("Protegrity1", "SUCCESS_STR", "SUCCESS_REPROTECT_STR")
    print("reprotect:    " + org)
except ReprotectError as e:
    print(e)
```

Result

```
3, The user does not have the appropriate permissions to perform the requested
operation.
```

Example: Bulk Data

In the following example, the *Protegrity1* string is repeated five times in a list, which is used as the input bulk data. The *NO_REPROTECT_USER* is used to try and reprotect the input data using the *SUCCESS_REPROTECT_STR* data element.

```
from appython import Protector
from appython.exceptions import ReprotectError
protector = Protector()
session = protector.create_session("NO_REPROTECT_USER")
data = ["Protegrity1"]*5
org = session.reprotect(data, "SUCCESS_STR", "SUCCESS_REPROTECT_STR")
print(org)
```

Result

```
([None, None, None, None, None], ('3', '3', '3', '3', '3'))
```

2.4.4.6.3 Mock Example - No Unprotect Null User

This section describes an example of the scenario in which a user does not have privileges to unprotect data. In this case, if the user tries to unprotect the data, then the *unprotect* API returns a null value.

Example: Single Data

In the following example, the *NO_UNPROTECT_NULL_USER* user is first used to protect the *Protegrity1* string using the *SUCCESS_STR* data element. Then, the *NO_UNPROTECT_NULL_USER* user is used to try and unprotect the protected input data using the same data element. However, the user is unable to unprotect the data and the API returns a null value.

```
from appython import Protector
from appython.exceptions import UnprotectError
protector = Protector()
session = protector.create_session("NO_UNPROTECT_NULL_USER")
p_out = session.protect("Protegrity1", "SUCCESS_STR")
print("Protected data: " + p_out)
org = session.unprotect(p_out, "SUCCESS_STR")
print("Unprotected data: ")
print(org)
```

Result

```
Protected data: lSvH5dvO5l5vvH5zvOvzaX
Unprotected data:
None
```

Example: Bulk Data

In the following example, the *Protegrity1* string is repeated five times in a list, which is used as the input bulk data. The *NO_UNPROTECT_NULL_USER* user is first used to protect the input data using the *SUCCESS_STR* data element.

Then, the *NO_UNPROTECT_NULL_USER* user is used to try and unprotect the protected input data using the same data element.

```
from appython import Protector
from appython.exceptions import UnprotectError
protector = Protector()
session = protector.create_session("NO_UNPROTECT_NULL_USER")
data = ["Protegrity1"]*5
p_out = session.protect(data, "SUCCESS_STR")
print("Protected data: ")
print(p_out)
org = session.unprotect(p_out[0], "SUCCESS_STR")
print("Unprotected data: ")
print(org)
```

Result

```
Protected data:
(['lSvH5dvO5l5vvH5zvOvzaX', 'lSvH5dvO5l5vvH5zvOvzaX', 'lSvH5dvO5l5vvH5zvOvzaX',
'lSvH5dvO5l5vvH5zvOvzaX', 'lSvH5dvO5l5vvH5zvOvzaX'], (6, 6, 6, 6, 6))
Unprotected data:
([None, None, None, None, None], ('3', '3', '3', '3', '3'))
```

2.4.4.6.4 Mock Example - No Unprotect Exception User

This section describes an example of the scenario in which a user does not have privileges to unprotect data. In this case, if the user tries to unprotect the data, then the *unprotect* API throws an exception.

Example: Single Data

In the following example, the *NO_UNPROTECT_NULL_USER* user is first used to protect the *Protegrity1* string using the *SUCCESS_STR* data element. Then, the *NO_UNPROTECT_NULL_USER* user is used to try and unprotect the protected input data using the same data element. However, the user is unable to unprotect the data and the API throws an exception.

```
from appython import Protector
from appython.exceptions import UnprotectError
protector = Protector()
session = protector.create_session("NO_UNPROTECT_EXC_USER")
p_out = session.protect("Protegrity1", "SUCCESS_STR")
print("Protected data: " + p_out)
try:
    org = session.unprotect(p_out, "SUCCESS_STR")
    print("Unprotected data: " + org)
except UnprotectError as e:
    print(e)
```

Result

```
Protected data: lSvH5dvO5l5vvH5zvOvzaX
3, The user does not have the appropriate permissions to perform the requested
operation.
```

Example: Bulk Data

In the following example, the *Protegrity1* string is repeated five times in a list, which is used as the input bulk data. The *NO_UNPROTECT_NULL_USER* user is first used to protect the input data using the *SUCCESS_STR* data element. Then, the *NO_UNPROTECT_NULL_USER* user is used to try and unprotect the protected input data using the same data element.

```
from appython import Protector
from appython.exceptions import UnprotectError
protector = Protector()
session = protector.create_session("NO_UNPROTECT_EXC_USER")
data = ["Protegrity1"]*5
```

```
p_out = session.protect(data, "SUCCESS_STR")
print("Protected data: ")
print(p_out)
org = session.unprotect(p_out[0], "SUCCESS_STR")
print("Unprotected data: ")
print(org)
```

Result

```
Protected data:
(['lSvH5dvO5l5vvH5zvOvzaX', 'lSvH5dvO5l5vvH5zvOvzaX', 'lSvH5dvO5l5vvH5zvOvzaX',
'lSvH5dvO5l5vvH5zvOvzaX', 'lSvH5dvO5l5vvH5zvOvzaX'], (6, 6, 6, 6, 6))
Unprotected data:
([None, None, None, None, None], ('3', '3', '3', '3', '3'))
```

2.4.4.6.5 Mock Example - No Unprotect Protected User

This section provides an example of the scenario in which a user does not have privileges to unprotect data. In this case, if the user tries to unprotect the data, then the *unprotect* API returns the protected input data.

Example: Single Data

In the following example, the *NO_UNPROTECT_PROTECTED_USER* user is first used to protect the *Protegrity1* string using the *SUCCESS_STR* data element. Then, the *NO_UNPROTECT_NULL_USER* user is used to try and unprotect the protected input data using the same data element. However, the user is unable to unprotect the data, and the API returns the protected input data.

```
from appython import Protector
from appython.exceptions import UnprotectError
protector = Protector()
session = protector.create_session("NO_UNPROTECT_PROTECTED_USER")
p_out = session.protect("Protegrity1", "SUCCESS_STR")
print("Protected data: " + p_out)
org = session.unprotect(p_out, "SUCCESS_STR")
print("Unprotected data: " + org)
```

Result

```
Protected data: lSvH5dvO5l5vvH5zvOvzaX
Unprotected data: lSvH5dvO5l5vvH5zvOvzaX
```

Example: Bulk Data

In the following example, the *Protegrity1* string is repeated five times in a list, which is used as the input bulk data. The *NO_UNPROTECT_PROTECTED_USER* user is first used to protect the input data using the *SUCCESS_STR* data element. Then, the *NO_UNPROTECT_PROTECTED_USER* user is used to try and unprotect the protected input data using the same data element.

```
from appython import Protector
from appython.exceptions import UnprotectError
protector = Protector()
session = protector.create_session("NO_UNPROTECT_PROTECTED_USER")
data = ["Protegrity1"]*5
p_out = session.protect(data, "SUCCESS_STR")
print("Protected data: ")
print(p_out)
org = session.unprotect(p_out[0], "SUCCESS_STR")
print("Unprotected data: ")
print(org)
```

Result

```
Protected data:
(['lSvH5dvO5l5vvH5zvOvzaX', 'lSvH5dvO5l5vvH5zvOvzaX', 'lSvH5dvO5l5vvH5zvOvzaX',
'lSvH5dvO5l5vvH5zvOvzaX', 'lSvH5dvO5l5vvH5zvOvzaX'], (6, 6, 6, 6, 6))
Unprotected data:
```

```
(['lSvH5dvO5l5vvH5zvOvzaX', 'lSvH5dvO5l5vvH5zvOvzaX', 'lSvH5dvO5l5vvH5zvOvzaX',
'lSvH5dvO5l5vvH5zvOvzaX', 'lSvH5dvO5l5vvH5zvOvzaX'], (3, 3, 3, 3, 3))
```

2.4.4.6.6 Mock Example - No User

This section describes an example of the scenario in which a user has not been defined in the security policy.

Example: Single Data

In the following example, the *NO_USER* user is used to try and protect the *Protegrity1* string using the *SUCCESS_STR* data element. However, the user is unable to perform the requested operation because the user has not been defined in the security policy.

```
from appython import Protector
from appython.exceptions import ProtectError, ReprotectError, UnprotectError
protector = Protector()
session = protector.create_session("NO_USER")
try:
    output = session.protect("Protegrity1", "SUCCESS_STR")
    print("Protected data: " + output)
except ProtectError as e:
    print(e)
```

Result

```
1, The username could not be found in the policy in shared memory.
```

Example: Bulk Data

In the following example, the *Protegrity1* string is repeated five times in a list, which is used as the input bulk data. The *NO_USER* user is used to try and protect the *Protegrity1* string using the *SUCCESS_STR* data element. However, the user is unable to perform the requested operation because the user has not been defined in the security policy.

```
from appython import Protector
from appython.exceptions import ProtectError, ReprotectError, UnprotectError
protector = Protector()
session = protector.create_session("NO_USER")
data = ["Protegrity1"]*5
output = session.protect(data, "SUCCESS_STR")
print(output)
```

Result

```
([None, None, None, None, None], ('1', '1', '1', '1', '1'))
```

2.5 Application Protector (AP) NodeJS APIs

A Trusted Application must be added in the datastore for running AP NodeJS. The AP NodeJS accesses the information on the Trusted Application from the policy stored in the memory. If the application is trusted, then the user can invoke the protect, unprotect, or reprotect APIs, depending on the requirements. You can flush the audits at the point where the application exits.

> **Note:** The AP NodeJS APIs can be invoked by a valid *Policy User* or a *Trusted Application* user.

> **Note:**
> When a short running application has completed its execution (in less than a second), the audit logs will not be seen. In such cases, the *flushAudits()* API needs to be invoked.
>
> For more information about the flushAudits API, refer to the section *flushAudits API*.

The following diagram represents the basic flow of the AP NodeJS.

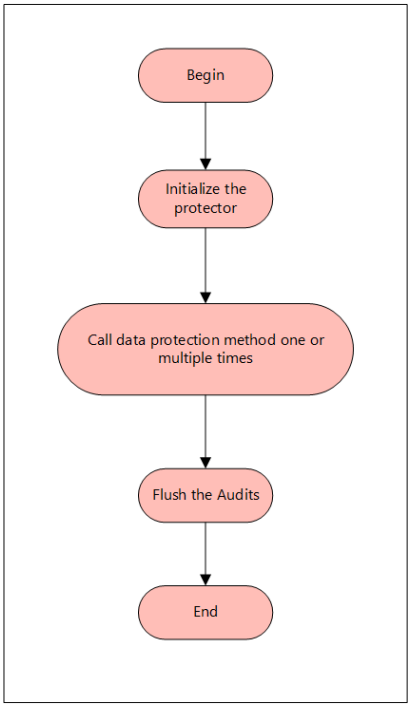

Figure 2-5: Flowchart for AP NodeJS

> **Note:**
> The AP NodeJS supports only bytes and string data type.

The following sections provide detailed information for the various methods used by the Protegrity Application Protector NodeJS.

2.5.1 initialize API

The *initialize* API initializes the Application Protector NodeJS. This should be performed only once in the lifecycle of each application that uses the AP NodeJS. The protection operations can be performed only on the successful initialization of the AP NodeJS.

function initialize(communicationID = 0)

Parameters

communicationID: The Communication ID in integer format that is used by the PEP server. This value must match the value specified by the Communication ID parameter in the *pepserver.cfg* file. This parameter is optional. The default value of the *communicationID* parameter is *0*.

> **Note:** Ensure that the *communicationID* parameter value is equal to the *Communication ID* parameter value in the *pepserver.cfg* file.

Returns

Promise<boolean>: Returns a promise in boolean format when the protector has been initialized successfully.

Exceptions

Error: If the initialization of the protector fails, then an exception is thrown.

Example

```
protector.initialize().then(result => {
    if (result) {
        /* The Protector has been initialized successfully */
    }
}).catch(error => {
    //handle exceptions
    console.error(error)
})
```

2.5.2 getVersion API

The *getVersion* API returns the extended version of the AP NodeJS in use. The extended version consists of the AP NodeJS version number and the PEP server version.

function getVersion()

Parameters

None

Returns

Object: Returns an object with product version of the installed AP NodeJS and the PEP server version.

Exceptions

None

Example

```
protector.getVersion()
```

2.5.3 checkAccess API

The *checkAccess* API returns the access permission status of the user for a specified data element.

function checkAccess(userName, dataElement, checkAccess)

Parameters

userName: String containing the username defined in the policy.

dataElement: String containing the name of the data element.

checkAccess: Type of the access permission of the user for the specified data element. You can specify a value for this parameter from the accessType constants, such as, protectAccess, unprotectAccess, or reprotectAccess.

Returns

boolean: Returns *true* if the user has the requested access on the data element.

Exceptions

Error: If the checkAccess operation is unsuccessful, then an exception is thrown.

Example

```
protector.checkAccess('userName', 'dataElement', protector.accessType.protectAccess);
protector.checkAccess('userName', 'dataElement', protector.accessType.unprotectAccess);
protector.checkAccess('userName', 'dataElement', protector.accessType.reprotectAccess);
```

2.5.4 flushAudits API

The *flushAudits* API is used for flushing the audit logs at any point in the application. This API is required for a short running process that lasts less than a second, to get the audit logs.

> **Note:**
> It is recommended to invoke *flushAudits* API at the point where the application exits.

function flushAudits()

Parameters

None

Returns

None

Exceptions

None

Example

```
protector.flushAudits()
```

2.5.5 protect API

The *protect* API protects the data using tokenization, data type preserving encryption, No Encryption, or encryption data element. It supports single and bulk protection without a maximum bulk size limit. However, you are recommended not to pass more than 1 MB of input data for each protection call.

function protect(inputData, userName, dataElement, exiv = null, extweak = null, charset = null)

Parameters

inputData: Data to be protected. You can provide the input data of byte or string type. However, you cannot provide the data from multiple data types at the same time in a bulk call.

userName: String containing the username defined in the policy.

dataElement: String containing the data element name defined in the policy.

exiv: External IV is an optional argument. It is a buffer containing data that is used as an initialization vector and accepts input in byte format. When the External IV is null, its value is ignored.

extweak: External Tweak is an optional argument that is used only for the FPE data elements. It is a buffer containing data that is used as an external tweak and accepts input in byte format. When the external tweak is empty, its value is ignored.

charset: Charset is an optional argument. It indicates the byte order of the input buffer. You can specify a value for this argument from the charset constants, such as, *utf8*, *utf16le*, or *utf16be*.

> **Note:** The default value for the *charset* argument is *UTF-8*.

> **Note:** The *charset* argument is only applicable for the input data of byte type.

Returns

- *For single data*: Returns an object with protected data in the following format.
  - For string input:

    ```
    { returncode: (int), output: (string) }
    ```

  - For byte input:

    ```
    { returncode: (int), output: (Buffer) }
    ```

- *For bulk data*: Returns an object with protected data in the following format:
  - For string input:

    ```
    {
      returncode: []int
      output:[]string,
      isSuccess: bool
    }
    ```

- For byte input:

```
{
  returncode: []int
  output:[]Buffer,
  isSuccess: bool
}
```

Exception

`Error`: If the protect operation is unsuccessful, then an exception is thrown. For String or Byte array, an exception is not thrown for error codes 22, 23, and 44. In such cases, *isSuccess* is returned as false.

For more information about the AP NodeJS error return codes, refer to the section *Application Protectors API Return Codes*.

Example

The following table provides examples of the API usage for tokenizing and encrypting the data for each input.

Input	Usage	Refer to
String	Tokenizing string data	*Example - Tokenizing String Data*
	Tokenizing string data with external IV	*Example - Tokenizing String Data with External IV*
	Protecting string data using FPE	*Example - Protecting String Data Using Format Preserving Encryption (FPE)*
	Protecting string data using FPE with external IV	*Example - Protecting String Data Using FPE with External IV*
	Protecting string data using FPE with external tweak	*Example - Protecting String Data Using FPE with External Tweak*
Bytes	Tokenizing bytes data	*Example - Tokenizing Bytes Data*
	Tokenizing bytes data with external IV	*Example - Tokenizing Bytes Data with External IV*
	Encrypting bytes data	*Example - Encrypting Bytes Data*

2.5.5.1 Example - Tokenizing String Data

This section describes how to use the *protect* API for tokenizing the string data.

Input

In this example, the following input data are used, which are tokenized using the *AlphaNum* Alpha-Numeric data element.

Single input data	input = 'Protegrity1'
Bulk input data	input = ['Protegrity1', 'Protegrity2', 'Protegrity3']

Operation

```
const protectedData = protector.protect(input, 'user1', 'AlphaNum')
```

Result

Example output for single input data:

```
{ returncode: 6, output: 'P8PCmC8gty1' }
```

Example output for bulk input data:

```
{
  returncode: [ 6, 6, 6 ],
  output: [ 'P8PCmC8gty1', 'PUVrjFb7ty2', 'PGWGOrhWty3' ],
```

```
  isSuccess: true
}
```

2.5.5.2 Example - Tokenizing String Data with External IV

This section describes how to use the *protect* API for tokenizing the string data with external initialization vector (IV).

> **Note:** If you want to pass external IV as an argument to the *protect* API, then you must convert it to UTF-8 bytes before passing it to the API.

Input

In this example, the following input data are used, which are tokenized using the *AlphaNum* Alpha-Numeric data element, with the help of the external IV *1234* that is passed as bytes.

Single input data	input = 'Protegrity1'
Bulk input data	input = ['Protegrity1', 'Protegrity2', 'Protegrity3']

Operation

```
const protectedData = protector.protect(input, 'user1', 'AlphaNum', Buffer.from('1234'))
```

Result

Example output for single input data:

```
{ returncode: 6, output: 'Ppfabceaty1' }
```

Example output for bulk input data:

```
{
  returncode: [ 6, 6, 6 ],
  output: [ 'Ppfabceaty1', 'PiCZ1qBQty2', 'PxFmR99yty3' ],
  isSuccess: true
}
```

2.5.5.3 Example - Protecting String Data Using Format Preserving Encryption (FPE)

This section describes how to use the *protect* API to protect the string data using Format Preserving Encryption (FPE) (FF1).

Input

In this example, the following input data are used, which are protected using the *FPE_Alpha* FPE data element.

Single input data	input = 'protegrity1234ÀÁÂÃÄÅÆÇÈÉ'
Bulk input data	input = ['protegrity1234ÀÁÂÃÄÅÆÇÈÉ', 'protegrity1234ÀÁÂÃÄ', 'protegrity1234ÀÁÂÃÄÅ']

Operation

```
const protectedData = protector.protect(input, 'user1', 'FPE_Alpha')
```

Result

Example output for single input data:

```
{ returncode: 6, output: 'pMLzkFabyp1234ÀÁÂÃÄÅÆÇÈÉ' }
```

Example output for bulk input data:

```
{
  returncode: [ 6, 6, 6 ],
```

```
  output: [
    'pMLzkFabyp1234ÀÁÂÃÄÅÆÇÈÉ',
    'pYFUEwnxeB1234ÀÁÂÃÄ',
    'pKnERbuJEH1234ÀÁÂÃÄÅ'
  ],
  isSuccess: true
}
```

2.5.5.4 Example - Protecting String Data Using FPE with External IV

This section describes how to use the *protect* API to protect the string data using FPE (FF1) with external IV.

> **Note:** If you want to pass external IV as an argument to the *protect* API, then you must convert them to UTF-8 bytes before passing them to the API.

Input

In this example, the following input data are used, which are protected using the *FPE_Num* FPE data element, with the help of external IV *1234* that are passed as bytes.

Single input data	input = '372875647747447'
Bulk input data	input = ['372875647747447', '562875647747412', '702875647747434']

Operation

```
const protectedData = protector.protect(input, 'user1', 'FPE_Num', Buffer.from('1234'))
```

Result

Example output for single input data:

```
{ returncode: 6, output: '376445170344927' }
```

Example output for bulk input data:

```
{
  returncode: [ 6, 6, 6 ],
  output: [ '376932013690437', '539755103178822', '762007695745544' ],
  isSuccess: true
}
```

2.5.5.5 Example - Protecting String Data Using FPE with External Tweak

This section describes how to use the *protect* API to protect the string data using FPE (FF1) with external tweak.

> **Note:** If you want to pass external tweak as an argument to the *protect* API, then you must convert them to UTF-8 bytes before passing them to the API.

Input

In this example, the following input data are used, which are protected using the *FPE_Num* FPE data element, with the help of external tweak *abcdef* that are passed as bytes.

Single input data	input = '372875647747447'
Bulk input data	input = ['372875647747447', '562875647747412', '702875647747434']

Operation

```
const protectedData = protector.protect(input, 'user1', 'FPE_Num', null,
Buffer.from('abcdef'))
```

Result

Example output for single input data:

```
{ returncode: 6, output: '376445170344927' }
```

Example output for bulk input data:

```
{
  returncode: [ 6, 6, 6 ],
  output: [ '376932013690437', '539755103178822', '762007695745544' ],
  isSuccess: true
}
```

2.5.5.6 Example - Tokenizing Bytes Data

This section describes how to use the *protect* API for tokenizing the bytes data.

> **Note:** The default value for the charset argument is *UTF8*. If the data elements have default encoding or Plaintext Encoding set as either *UTF16LE* or *UTF16BE*, then the respective charset argument must be passed to the API.

> **Note:** Ensure that the input data is of the same byte format as selected in the Plaintext Encoding or default encoding drop-down for the required data element.

Input

In this example, the following input data are used, which are tokenized using the *AlphaNum* Alpha-Numeric data element and *UnicodeG2_UTF16LE* Unicode Gen2 data element.

Single input data for Alpha-Numeric data element	input = Buffer.from('Protegrity1')
Single input data for Unicode Gen2 data element	byteInputUTF16LE = Buffer.from('Protegrity1', 'utf16le')
Bulk input data for Alpha-Numeric data element	input = [Buffer.from('Protegrity1'), Buffer.from('Protegrity2'), Buffer.from('Protegrity3')]
Bulk input data for Unicode Gen2 data element	byteInputUTF16LE = [Buffer.from('Protegrity1', 'utf16le'), Buffer.from('Protegrity2', 'utf16le'), Buffer.from('Protegrity3', 'utf16le')]

Operation

The following example shows how to perform the protect operation using the *AlphaNum* Alpha-Numeric token data element.

```
const protectedData = protector.protect( input, 'user1', 'AlphaNum')
```

The following example shows how to perform the protect operation using the *UnicodeG2_UTF16LE* Unicode Gen2 token data element.

```
const protectedData = protector.protect( byteInputUTF16LE, 'user1',
'UnicodeG2_UTF16LE', null, null, protector.charset.utf16le)
```

Result

Example output for single input data for Alpha-Numeric data element:

```
{ returncode: 6, output: <Buffer 50 38 50 43 6d 43 38 67 74 79 31> }
```

Example output for single input data for Unicode Gen2 data element:

```
{ returncode: 6, output: <Buffer 5000 3800 5000 4300 6d00 4300 3800 6700 7400 7900
3100> }
```

Example output for bulk input data for Alpha-Numeric data element:

```
{
  returncode: [ 6, 6, 6 ],
  output: [
    <Buffer 50 38 50 43 6d 43 38 67 74 79 31>,
    <Buffer 50 55 56 72 6a 46 62 37 74 79 32>,
    <Buffer 50 47 57 47 4f 72 68 57 74 79 33>
  ],
  isSuccess: true
}
```

Example output for bulk input data for Unicode Gen2 data element:

```
{
  returncode: [ 6, 6, 6 ],
  output: [
    <Buffer 5000 3800 5000 4300 6d00 4300 3800 6700 7400 7900 3100>,
    <Buffer 5000 5500 5600 7200 6a00 4600 6200 3700 7400 7900 3200>,
    <Buffer 5000 4700 5700 4700 4f00 7200 6800 5700 7400 7900 3300>
  ],
  isSuccess: true
}
```

2.5.5.7 Example - Tokenizing Bytes Data with External IV

This section describes how to use the *protect* API for tokenizing the bytes data using external IV.

Input

In this example, the following input data are used, which are tokenized using the *AlphaNum* Alpha-Numeric data element and *UnicodeG2_UTF16LE* Unicode Gen2 data element, with the help of the external IV *1234* that is passed as bytes

Single input data for Alpha-Numeric data element	input = Buffer.from('Protegrity1')
Single input data for Unicode Gen2 data element	byteInputUTF16LE = Buffer.from('Protegrity1', 'utf16le')
Bulk input data for Alpha-Numeric data element	input = [Buffer.from('Protegrity1'), Buffer.from('Protegrity2'), Buffer.from('Protegrity3')]
Bulk input data for Unicode Gen2 data element	byteInputUTF16LE = [Buffer.from('Protegrity1', 'utf16le'), Buffer.from('Protegrity2', 'utf16le'), Buffer.from('Protegrity3', 'utf16le')]

Operation

The following example shows how to perform the protect operation using the *AlphaNum* Alpha-Numeric token data element.

```
const protectedData = protector.protect( input, 'user1', 'AlphaNum',
Buffer.from('1234'))
```

The following example shows how to perform the protect operation using the *UnicodeG2_UTF16LE* Unicode Gen2 token data element.

```
const protectedData = protector.protect( byteInputUTF16LE, 'user1',
'UnicodeG2_UTF16LE', Buffer.from('1234'), null, protector.charset.utf16le)
```

Result

Example output for single input data for Alpha-Numeric data element:

```
{ returncode: 6, output: <Buffer 50 43 76 47 34 78 34 30 74 79 31> }
```

Example output for single input data for Unicode Gen2 data element:

```
{ returncode: 6, output: <Buffer 5000 4300 7600 4700 3400 7800 3400 3000 7400 7900
3100> }
```

Example output for bulk input data for Alpha-Numeric data element:

```
{
  returncode: [ 6, 6, 6 ],
  output: [
    <Buffer 50 43 76 47 34 78 34 30 74 79 31>,
    <Buffer 50 62 65 53 6f 31 43 48 74 79 32>,
    <Buffer 50 59 61 6b 59 71 50 53 74 79 33>
  ],
  isSuccess: true
}
```

Example output for bulk input data for Unicode Gen2 data element:

```
{
  returncode: [ 6, 6, 6 ],
  output: [
    <Buffer 5000 4300 7600 4700 3400 7800 3400 3000 7400 7900 3100>,
    <Buffer 5000 6200 6500 5300 6f00 3100 4300 4800 7400 7900 3200>,
    <Buffer 5000 5900 6100 6b00 5900 7100 5000 5300 7400 7900 3300>
  ],
  isSuccess: true
}
```

2.5.5.8 Example - Encrypting Bytes Data

This section describes how to use the *protect* API for encrypting the bytes data.

Warning: To avoid data corruption, do not convert the encrypted bytes data into string format. It is recommended to convert the encrypted bytes data to a Hexadecimal, Base 64, or any other appropriate format.

Input

In this example, the following input data are used, which are first converted to bytes using the Buffer class in NodeJS. The bytes data is then encrypted using the *AES256* data element.

Single input data	input = Buffer.from('Protegrity1')
Bulk input data	input = [Buffer.from('Protegrity1'), Buffer.from('Protegrity2'), Buffer.from('Protegrity3')]

Operation

```
const protectedData = protector.protect( input, 'user1', 'AES256')
```

Result

Example output for single input data:

```
{
  returncode: 6,
  output: <Buffer 74 80 f5 8d 9e 0b 2b 34 4c 71 8a 97 db 8f 78 16>
}
```

Example output for bulk input data:

```
{
  returncode: [ 6, 6, 6 ],
  output: [
    <Buffer 74 80 f5 8d 9e 0b 2b 34 4c 71 8a 97 db 8f 78 16>,
    <Buffer 5b 09 28 e3 1e d3 c1 1a 59 01 0c 58 63 dd c7 b6>,
    <Buffer c0 6d 5e fc 87 f3 4b 15 00 92 fb 3f 50 ff c8 6f>
  ],
  isSuccess: true
}
```

2.5.6 unprotect API

The *unprotect* API unprotects the data using tokenization, data type preserving encryption, No Encryption, or encryption data element. It supports single and bulk unprotection without a maximum bulk size limit. However, you are recommended not to pass more than 1 MB of input data for each protection call.

function unprotect(inputData, userName, dataElement, exiv = null, extweak = null, charset = null)

Parameters

inputData: Data to be unprotected. You can provide the input data of byte or string type. However, you cannot provide the data from multiple data types at the same time in a bulk call.

userName: String containing the username defined in the policy.

dataElement: String containing the data element name defined in the policy.

exiv: External IV is an optional argument. It is a buffer containing data that is used as an initialization vector and accepts input in byte format. When the external IV is null, its value is ignored.

extweak: External Tweak is an optional argument that is used only for the FPE data elements. It is a buffer containing data that is used as an external tweak and accepts input in byte format. When the external tweak is empty, its value is ignored.

charset: Charset is an optional argument. It indicates the byte order of the input buffer. You can specify a value for this argument from the charset constants, such as, *utf8*, *utf16le*, or *utf16be*.

> **Note:** The default value for the *charset* argument is *UTF-8*.

> **Note:** The *charset* argument is only applicable for the input data of byte type.

Returns

- *For single data*: Returns an object with unprotected data in the following format.
  - For string data type:

    ```
    { returncode: (int), output: (string) }
    ```

  - For buffer data type:

    ```
    { returncode: (int), output: (Buffer) }
    ```

- *For bulk data*: Returns an object with unprotected data in the following format:
  - For string input:

    ```
    {
      returncode: []int
      output:[]string,
      isSuccess: bool
    }
    ```

- For byte input:

```
{
  returncode: []int
  output:[]Buffer,
  isSuccess: bool
}
```

Exceptions

`Error`: If the unprotect operation is unsuccessful, then an exception is thrown. For String or Byte array, an exception is not thrown for error codes 22, 23, and 44. In such cases, *isSuccess* is returned as false.

For more information about the AP NodeJS error return codes, refer to the section *Application Protectors API Return Codes*.

Example

The following table provides examples of the API usage for detokenizing and decrypting the data for each input.

Input	Usage	Refer to
String	Detokenizing string data	*Example - Detokenizing String Data*
	Detokenizing string data with external IV	*Example - Detokenizing String Data with External IV*
	Unprotecting string data using FPE	*Example - Unprotecting String Data Using Format Preserving Encryption (FPE)*
	Unprotecting string data using FPE with external IV	*Example - Unprotecting String Data Using FPE with External IV*
	Unprotecting string data using FPE with external tweak	*Example - Unprotecting String Data Using FPE with External Tweak*
Bytes	Detokenizing bytes data	*Example - Detokenizing Bytes Data*
	Detokenizing bytes data with external IV	*Example - Detokenizing Bytes Data with External IV*
	Decrypting bytes data	*Example - Decrypting Bytes Data*

2.5.6.1 Example - Detokenizing String Data

This section describes how to use the *unprotect* API retrieving the original string data from the token data.

Input

In this example, the following input data are used, which are tokenized using the *AlphaNum* Alpha-Numeric data element. The protected output is detokenized using the same data element.

Single input data	input = 'Protegrity1'
Bulk input data	input = ['Protegrity1', 'Protegrity2', 'Protegrity3']

Operation

```
const protectedData = protector.protect(input, 'user1', 'AlphaNum')

const unprotectedData = protector.unprotect(protectedData.output, 'user1', 'AlphaNum')
```

Result

Example output for single input data:

```
{ returncode: 8, output: 'Protegrity1' }
```

Example output for bulk input data:

```
{
  returncode: [ 8, 8, 8 ],
  output: [ 'Protegrity1', 'Protegrity2', 'Protegrity3' ],
  isSuccess: true
}
```

2.5.6.2 Example - Detokenizing String Data with External IV

This section describes how to use the *unprotect* API retrieving the original string data from the token data, using external IV.

> **Note:** If you want to pass external IV as an argument to the *unprotect* API, then you must convert it to UTF-8 bytes before passing it to the API.

Input

In this example, the following input data are used, which are tokenized using the *AlphaNum* Alpha-Numeric data element, with the help of the external IV *1234* that is passed as bytes. The protected output is detokenized using the same data element.

Single input data	input = 'Protegrity1'
Bulk input data	input = ['Protegrity1', 'Protegrity2', 'Protegrity3']

Operation

```
const protectedData = protector.protect(input, 'user1', 'AlphaNum', Buffer.from('1234'))

const unprotectedData = protector.unprotect(protectedData.output, 'user1', 'AlphaNum',
Buffer.from('1234'))
```

Result

Example output for single input data:

```
{ returncode: 8, output: 'Protegrity1' }
```

Example output for bulk input data:

```
{
  returncode: [ 8, 8, 8 ],
  output: [ 'Protegrity1', 'Protegrity2', 'Protegrity3' ],
  isSuccess: true
}
```

2.5.6.3 Example - Unprotecting String Data Using Format Preserving Encryption (FPE)

This section describes how to use the *unprotect* API to unprotect the string data using Format Preserving Encryption (FPE) (FF1).

Input

In this example, the following input data are used, which are protected using the *FPE_Alpha* FPE data element. The protected output is unprotected using the same data element.

Single input data	input = 'protegrity1234ÀÁÂÃÄÅÆÇÈÉ'
Bulk input data	input = ['protegrity1234ÀÁÂÃÄÅÆÇÈÉ', 'protegrity1234ÀÁÂÃÄ', 'protegrity1234ÀÁÂÃÄÅ']

Operation

```
const protectedData = protector.protect(input, 'user1', 'FPE_Alpha')

const unprotectedData = protector.unprotect(protectedData.output, 'user1', 'FPE_Alpha')
```

Result

Example output for single input data:

```
{ returncode: 8, output: 'protegrity1234ÀÁÂÃÄÅÆÇÈÉ' }
```

Example output for bulk input data:

```
{
  returncode: [ 8, 8, 8 ],
  output: [
    'protegrity1234ÀÁÂÃÄÅÆÇÈÉ',
    'protegrity1234ÀÁÂÃÄ',
    'protegrity1234ÀÁÂÃÄÅ'
  ],
  isSuccess: true
}
```

2.5.6.4 Example - Unprotecting String Data Using FPE with External IV

This section describes how to use the *unprotect* API to unprotect the string data using FPE (FF1) with external IV.

> **Note:** If you want to pass external IV as an argument to the *unprotect* API, then you must convert them to UTF-8 bytes before passing them to the API.

Input

In this example, the following input data are used, which are protected using the *FPE_Num* FPE data element, with the help of external IV *1234* that are passed as bytes. The protected output is unprotected using the same data element.

Single input data	input = '372875647747447'
Bulk input data	input = ['372875647747447', '562875647747412', '702875647747434']

Operation

```
const protectedData = protector.protect(input, 'user1', 'FPE_Num', Buffer.from('1234'))

const unprotectedData = protector.unprotect(protectedData.output, 'user1', 'FPE_Num',
Buffer.from('1234'))
```

Result

Example output for single input data:

```
{ returncode: 8, output: '372875647747447' }
```

Example output for bulk input data:

```
{
  returncode: [ 8, 8, 8 ],
  output: [ '372875647747447', '562875647747412', '702875647747434' ],
  isSuccess: true
}
```

2.5.6.5 Example - Unprotecting String Data Using FPE with External Tweak

This section describes how to use the *unprotect* API to unprotect the string data using FPE (FF1) with external tweak.

> **Note:** If you want to pass external tweak as an argument to the *unprotect* API, then you must convert them to UTF-8 bytes before passing them to the API.

Input

In this example, the following input data are used, which are protected using the *FPE_Num* FPE data element, with the help of external tweak *abcdef* that are passed as bytes. The protected output is unprotected using the same data element.

Single input data	input = '372875647747447'
Bulk input data	input = ['372875647747447', '562875647747412', '702875647747434']

Operation

```
const protectedData = protector.protect(input, 'user1', 'FPE_Num', null,
Buffer.from('abcdef'))

const unprotectedData = protector.unprotect(protectedData.output, 'user1', 'FPE_Num',
null, Buffer.from('abcdef'))
```

Result

Example output for single input data:

```
{ returncode: 8, output: '372875647747447' }
```

Example output for bulk input data:

```
{
  returncode: [ 8, 8, 8 ],
  output: [ '372875647747447', '562875647747412', '702875647747434' ],
  isSuccess: true
}
```

2.5.6.6 Example - Detokenizing Bytes Data

This section describes how to use the *unprotect* API for detokenizing the bytes data.

> **Note:** The default value for the charset argument is *UTF8*. If the data elements have default encoding or Plaintext Encoding set as either *UTF16LE* or *UTF16BE*, then the respective charset argument must be passed to the API.

> **Note:** Ensure that the input data is of the same byte format as selected in the Plaintext Encoding or default encoding drop-down for the required FPE or Unicode Gen2 data element.

Input

In this example, the following input data are used, which are tokenized using the *AlphaNum* Alpha-Numeric data element and *UnicodeG2_UTF16LE* Unicode Gen2 data element. The protected output is detokenized using the same data element.

Single input data for Alpha-Numeric data element	input = Buffer.from('Protegrity1')
Single input data for Unicode Gen2 data element	byteInputUTF16LE = Buffer.from('Protegrity1', 'utf16le')
Bulk input data for Alpha-Numeric data element	input = [Buffer.from('Protegrity1'), Buffer.from('Protegrity2'), Buffer.from('Protegrity3')]
Bulk input data for Unicode Gen2 data element	byteInputUTF16LE = [Buffer.from('Protegrity1', 'utf16le'), Buffer.from('Protegrity2', 'utf16le'), Buffer.from('Protegrity3', 'utf16le')]

Operation

The following example shows how to perform the unprotect operation using the *AlphaNum* Alpha-Numeric token data element.

```
const protectedData = protector.protect( input, 'user1', 'AlphaNum')

const unprotectedData = protector.unprotect( protectedData.output, 'user1', 'AlphaNum')
```

The following example shows how to perform the unprotect operation using the *UnicodeG2_UTF16LE* Unicode Gen2 token data element.

```
const protectedData = protector.protect( byteInputUTF16LE, 'user1',
'UnicodeG2_UTF16LE', null, null, protector.charset.utf16le)

const unprotectedData = protector.unprotect( protectedData.output, 'user1',
'UnicodeG2_UTF16LE', null, null, protector.charset.utf16le)
```

Result

Example output for single input data for Alpha-Numeric data element:

```
{ returncode: 8, output: <Buffer 50 72 6f 74 65 67 72 69 74 79 31> }
```

Example output for single input data for Unicode Gen2 data element:

```
{ returncode: 8, output: <Buffer 5000 7200 6f00 7400 6500 6700 7200 6900 7400 7900
3100> }
```

Example output for bulk input data for Alpha-Numeric data element:

```
{
  returncode: [ 8, 8, 8 ],
  output: [
    <Buffer 50 72 6f 74 65 67 72 69 74 79 31>,
    <Buffer 50 72 6f 74 65 67 72 69 74 79 32>,
    <Buffer 50 72 6f 74 65 67 72 69 74 79 33>
  ],
  isSuccess: true
}
```

Example output for bulk input data for Unicode Gen2 data element:

```
{
  returncode: [ 8, 8, 8 ],
  output: [
    <Buffer 5000 7200 6f00 7400 6500 6700 7200 6900 7400 7900 3100>,
    <Buffer 5000 7200 6f00 7400 6500 6700 7200 6900 7400 7900 3200>,
    <Buffer 5000 7200 6f00 7400 6500 6700 7200 6900 7400 7900 3300>
  ],
  isSuccess: true
}
```

2.5.6.7 Example - Detokenizing Bytes Data with External IV

This section describes how to use the *unprotect* API for detokenizing the bytes data using external IV.

Input

In this example, the following input data are used, which are tokenized using the *AlphaNum* Alpha-Numeric data element and *UnicodeG2_UTF16LE* Unicode Gen2 data element. The protected output is detokenized using the same data element.

Single input data for Alpha-Numeric data element	input = Buffer.from('Protegrity1')
Single input data for Unicode Gen2 data element	byteInputUTF16LE = Buffer.from('Protegrity1', 'utf16le')

Bulk input data for Alpha-Numeric data element	input = [Buffer.from('Protegrity1'), Buffer.from('Protegrity2'), Buffer.from('Protegrity3')]
Bulk input data for Unicode Gen2 data element	byteInputUTF16LE = [Buffer.from('Protegrity1', 'utf16le'), Buffer.from('Protegrity2', 'utf16le'), Buffer.from('Protegrity3', 'utf16le')]

Operation

The following example shows how to perform the unprotect operation using the *AlphaNum* Alpha-Numeric token data element.

```
const protectedData = protector.protect( input, 'user1', 'AlphaNum',
Buffer.from('1234'))

const unprotectedData = protector.unprotect( protectedData.output, 'user1', 'AlphaNum',
Buffer.from('1234'))
```

The following example shows how to perform the unprotect operation using the *UnicodeG2_UTF16LE* Unicode Gen2 token data element.

```
const protectedData = protector.protect( byteInputUTF16LE, 'user1',
'UnicodeG2_UTF16LE', Buffer.from('1234'), null, protector.charset.utf16le)

const unprotectedData = protector.unprotect( protectedData.output, 'user1',
'UnicodeG2_UTF16LE', Buffer.from('1234'), null, protector.charset.utf16le)
```

Result

Example output for single input data for Alpha-Numeric data element:

```
{ returncode: 8, output: <Buffer 50 72 6f 74 65 67 72 69 74 79 31> }
```

Example output for single input data for Unicode Gen2 data element:

```
{ returncode: 8, output: <Buffer 5000 7200 6f00 7400 6500 6700 7200 6900 7400 7900
3100> }
```

Example output for bulk input data for Alpha-Numeric data element:

```
{
  returncode: [ 8, 8, 8 ],
  output: [
    <Buffer 50 72 6f 74 65 67 72 69 74 79 31>,
    <Buffer 50 72 6f 74 65 67 72 69 74 79 32>,
    <Buffer 50 72 6f 74 65 67 72 69 74 79 33>
  ],
  isSuccess: true
}
```

Example output for bulk input data for Unicode Gen2 data element:

```
{
  returncode: [ 8, 8, 8 ],
  output: [
    <Buffer 5000 7200 6f00 7400 6500 6700 7200 6900 7400 7900 3100>,
    <Buffer 5000 7200 6f00 7400 6500 6700 7200 6900 7400 7900 3200>,
    <Buffer 5000 7200 6f00 7400 6500 6700 7200 6900 7400 7900 3300>
  ],
  isSuccess: true
}
```

2.5.6.8 Example - Decrypting Bytes Data

This section describes how to use the *unprotect* API for decrypting the bytes data.

Warning: To avoid data corruption, do not convert the encrypted bytes data into string format. It is recommended to convert the encrypted bytes data to a Hexadecimal, Base 64, or any other appropriate format.

Input

In this example, the following input data are used, which are first converted to bytes using the Buffer class in NodeJS. The bytes data is then encrypted using the *AES256* data element. The protected output is decrypted using the same data element.

Single input data	input = Buffer.from('Protegrity1')
Bulk input data	input = [Buffer.from('Protegrity1'), Buffer.from('Protegrity2'), Buffer.from('Protegrity3')]

Operation

```
const protectedData = protector.protect( input, 'user1', 'AES256')

const unprotectedData = protector.unprotect( protectedData.output, 'user1', 'AES256')
```

Result

Example output for single input data:

```
{ returncode: 8, output: <Buffer 50 72 6f 74 65 67 72 69 74 79 31> }
```

Example output for bulk input data:

```
{
  returncode: [ 8, 8, 8 ],
  output: [
    <Buffer 50 72 6f 74 65 67 72 69 74 79 31>,
    <Buffer 50 72 6f 74 65 67 72 69 74 79 32>,
    <Buffer 50 72 6f 74 65 67 72 69 74 79 33>
  ],
  isSuccess: true
}
```

2.5.7 reprotect API

The *reprotect* API protects the data using tokenization, data type preserving encryption, No Encryption, or encryption data element. It supports single and bulk protection without a maximum bulk size limit. However, you are recommended not to pass more than 1 MB of input data for each protection call.

function reprotect(inputData, userName, oldDataElement, newDataElement, oldexiv = null, newexiv = null, oldextweak = null, newextweak = null, charset = null)

Parameters

inputData: Data to be reprotected. You can provide the input data of byte or string type. However, you cannot provide the data from multiple data types at the same time in a bulk call.

userName: String containing the username defined in the policy.

oldDataElement: String containing the old data element name defined in the policy.

newDataElement: String containing the new data element name defined in the policy.

oldexiv: External IV is an optional argument. It is a buffer containing old external IV and accepts input in byte format. When the old external IV is null, its value is ignored.

newexiv: External IV is an optional argument. It is a buffer containing new external IV and accepts input in byte format. When the new external IV is null, its value is ignored.

oldextweak: External Tweak is an optional argument that is used only for the FPE data elements. It is a buffer containing data that is used as an old external tweak and accepts input in byte format. When the old external tweak is empty, its value is ignored.

newextweak: External Tweak is an optional argument that is used only for the FPE data elements. It is a buffer containing data that is used as an new external tweak and accepts input in byte format. When the new external tweak is empty, its value is ignored.

charset: Charset is an optional argument. It indicates the byte order of the input buffer. You can specify a value for this argument from the charset constants, such as, *utf8*, *utf16le*, or *utf16be*.

> **Note:** The default value for the *charset* argument is *UTF-8*.

> **Note:** The *charset* argument is only applicable for the input data of byte type.

Returns

- *For single data*: Returns an object with reprotected data in the following format.
  - For string input:

    ```
    { returncode: (int), output: (string) }
    ```

  - For byte input:

    ```
    { returncode: (int), output: (Buffer) }
    ```

- *For bulk data*: Returns an object with reprotected data in the following format:
  - For string input:

    ```
    {
      returncode: []int
      output:[]string,
      isSuccess: bool
    }
    ```

  - For byte input:

    ```
    {
      returncode: []int
      output:[]Buffer,
      isSuccess: bool
    }
    ```

Exceptions

Error: If the reprotect operation is unsuccessful, then an exception is thrown. For String or Byte array, an exception is not thrown for error codes 22, 23, and 44. In such cases, *isSuccess* is returned as false.

For more information about the AP NodeJS error return codes, refer to the section *Application Protectors API Return Codes*.

Example

The following table provides examples of the API usage for retokenizing and re-encrypting the data for each input.

Input	Usage	Refer to
String	Retokenizing string data	*Example - Retokenizing String Data*
	Retokenizing string data with external IV	*Example - Retokenizing String Data with External IV*
	Reprotecting string data using FPE	*Example - Reprotecting String Data Using FPE*

Input	Usage	Refer to
	Reprotecting string data using FPE with external IV	*Example - Reprotecting String Data Using FPE with External IV*
	Reprotecting string data using FPE with external tweak	*Example - Reprotecting String Data Using FPE with External Tweak*
Bytes	Retokenizing bytes data	*Example - Retokenizing Bytes Data*
	Retokenizing bytes data with external IV	*Example - Retokenizing Bytes Data with External IV*
	Re-encrypting bytes data	*Example - Re-encrypting Bytes Data*

2.5.7.1 Example - Retokenizing String Data

This section describes how to use the *reprotect* API for retokenizing string data.

Warning:
If you are retokenizing the data using the *reprotect* API, then the old data element and the new data element must have the same tokenization type.

For example, if you have used the Alpha-Numeric data element to protect the data, then you must use only the Alpha-Numeric data element to reprotect the data.

Input

In this example, the following input data are used, which are tokenized using the *AlphaNum* Alpha-Numeric data element. The tokenized input data is retokenized using a new *AlphaNum_1* Alpha-Numeric data element.

Single input data	input = 'Protegrity1'
Bulk input data	input = ['Protegrity1', 'Protegrity2', 'Protegrity3']

Operation

```
const protectedData = protector.protect(input, 'user1', 'AlphaNum')

const reprotectedData = protector.reprotect(protectedData.output, 'user1', 'AlphaNum',
'AlphaNum_1')
```

Result

Example output for single input data:

```
{ returncode: 50, output: 'OgDuaYeAja9' }
```

Example output for bulk input data:

```
{
  returncode: [ 50, 50, 50 ],
  output: [ 'OgDuaYeAja9', 'YrKxfmdTGu6', 'QrKxfmdTGi4' ],
  isSuccess: true
}
```

2.5.7.2 Example - Retokenizing String Data with External IV

This section describes how to use the *reprotect* API for retokenizing the string data, using external IV.

Warning:
If you are retokenizing the data using the *reprotect* API, then the old data element and the new data element must have the same tokenization type.

For example, if you have used the Alpha-Numeric data element to protect the data, then you must use only the Alpha-Numeric data element to reprotect the data.

Note: If you want to pass external IV as an argument to the *reprotect* API, then you must convert it to UTF-8 bytes before passing it to the API.

Input

In this example, the following input data are used, which are tokenized using the *AlphaNum* Alpha-Numeric data element, with the help of the external IV *1234* that is passed as bytes. The tokenized input data is retokenized using a new *AlphaNum* Alpha-Numeric data element, with the help of the external IV *123456* that is passed as bytes.

Single input data	input = 'Protegrity1'
Bulk input data	input = ['Protegrity1', 'Protegrity2', 'Protegrity3']

Operation

```
const protectedData = protector.protect(input, 'user1', 'AlphaNum', Buffer.from('1234'))

const reprotectedData = protector.reprotect(protectedData.output, 'user1', 'AlphaNum',
'AlphaNum', Buffer.from('1234'), Buffer.from('123456'))
```

Result

Example output for single input data:

```
{ returncode: 50, output: 'OgDuaYeAja9' }
```

Example output for bulk input data:

```
{
  returncode: [ 50, 50, 50 ],
  output: [ 'OgDuaYeAja9', 'YrKxfmdTGu6', 'QrKxfmdTGi4' ],
  isSuccess: true
}
```

2.5.7.3 Example - Reprotecting String Data Using Format Preserving Encryption (FPE)

This section describes how to use the *reprotect* API to reprotect the string data using Format Preserving Encryption (FPE) (FF1).

Warning:

If you are reprotecting the data using the *reprotect* API, then the old data element and the new data element must have the same FPE type.

For example, if you have used the FPE data element to protect the data, then you must use only the FPE data element to reprotect the data.

Input

In this example, the following input data are used, which are protected using the *FPE_Alpha* FPE data element. The protected input data is reprotected using a new *FPE_Alpha_1* FPE data element.

Single input data	input = 'protegrity1234ÀÁÂÃÄÅÆÇÈÉ'
Bulk input data	input = ['protegrity1234ÀÁÂÃÄÅÆÇÈÉ', 'protegrity1234ÀÁÂÃÄ', 'protegrity1234ÀÁÂÃÄÅ']

Operation

```
const protectedData = protector.protect(input, 'user1', 'FPE_Alpha')
```

```
const reprotectedData = protector.reprotect(protectedData.output, 'user1', 'FPE_Alpha',
'FPE_Alpha_1')
```

Result

Example output for single input data:

```
{ returncode: 50, output: 'jMLzkFabyp1234ÀÁÂÃÄÅÆÇÈÉ' }
```

Example output for bulk input data:

```
{
  returncode: [ 50, 50, 50 ],
  output: [
    'jMLzkFabyp1234ÀÁÂÃÄÅÆÇÈÉ',
    'hYFUEwnxeB1234ÀÁÂÃÄ',
    'iKnERbuJEH1234ÀÁÂÃÄÅ'
  ],
  isSuccess: true
}
```

2.5.7.4 Example - Reprotecting String Data Using FPE with External IV

This section describes how to use the *reprotect* API to reprotect the string data using FPE (FF1) with external IV.

Warning:
If you are reprotecting the data using the *reprotect* API, then the old data element and the new data element must have the same FPE type.

For example, if you have used the FPE data element to protect the data, then you must use only the FPE data element to reprotect the data.

Note: If you want to pass external IV as an argument to the *reprotect* API, then you must convert them to UTF-8 bytes before passing them to the API.

Input

In this example, the following input data are used, which are protected using the *FPE_Num* FPE data element, with the help of external IV *1234* that are passed as bytes. The protected input data is reprotected using a new *FPE_Num* FPE data element, with the help of external IV *123456* that are passed as bytes.

Single input data	input = '372875647747447'
Bulk input data	input = ['372875647747447', '562875647747412', '702875647747434']

Operation

```
const protectedData = protector.protect(input, 'user1', 'FPE_Num', Buffer.from('1234'))

const reprotectedData = protector.reprotect(protectedData.output, 'user1', 'FPE_Num',
'FPE_Num', Buffer.from('1234'), Buffer.from('123456'))
```

Result

Example output for single input data:

```
{ returncode: 50, output: '386445170344927' }
```

Example output for bulk input data:

```
{
  {
```

```
  returncode: [ 50, 50, 50 ],
  output: [ '386445170344927', '539755103178822', '762007695745544' ],
  isSuccess: true
}
```

2.5.7.5 Example - Reprotecting String Data Using FPE with External Tweak

This section describes how to use the *reprotect* API to reprotect the string data using FPE (FF1) with external tweak.

Warning:
If you are reprotecting the data using the *reprotect* API, then the old data element and the new data element must have the same FPE type.

For example, if you have used the FPE data element to protect the data, then you must use only the FPE data element to reprotect the data.

Note: If you want to pass external tweak as an argument to the *reprotect* API, then you must convert them to UTF-8 bytes before passing them to the API.

Input

In this example, the following input data are used, which are protected using the *FPE_Num* FPE data element, with the help of external tweak *abcd* that are passed as bytes. The protected input data is reprotected using a new *FPE_Num* FPE data element, with the help of external tweak *abcdef* that are passed as bytes.

Single input data	input = '372875647747447'
Bulk input data	input = ['372875647747447', '562875647747412', '702875647747434']

Operation

```
const protectedData = protector.protect(input, 'user1', 'FPE_Num', null,
Buffer.from('abcd'))

const reprotectedData = protector.reprotect(protectedData.output, 'user1', 'FPE_Num',
'FPE_Num', null, null, Buffer.from('abcd'), Buffer.from('abcdef'))
```

Result

Example output for single input data:

```
{ returncode: 50, output: '386445170344927' }
```

Example output for bulk input data:

```
{
  {
  returncode: [ 50, 50, 50 ],
  output: [ '386445170344927', '539755103178822', '762007695745544' ],
  isSuccess: true
}
```

2.5.7.6 Example - Retokenizing Bytes Data

This section describes how to use the *reprotect* API for retokenizing the bytes data.

Warning:
If you are retokenizing the data using the *reprotect* API, then the old data element and the new data element must have the same tokenization type.

For example, if you have used the Alpha-Numeric data element to protect the data, then you must use only the Alpha-Numeric data element to reprotect the data.

Note: The default value for the charset argument is *UTF8*. If the data elements have default encoding or Plaintext Encoding set as either *UTF16LE* or *UTF16BE*, then the respective charset argument must be passed to the API.

Note: Ensure that the input data is of the same byte format as selected in the Plaintext Encoding or default encoding drop-down for the required FPE or Unicode Gen2 data element.

Input

In this example, the following input data are used, which are tokenized using the *AlphaNum* Alpha-Numeric data element and *UnicodeG2_UTF16LE* Unicode Gen2 data element. The protected input data is reprotected using a new *AlphaNum_1* Alpha-Numeric data element and a new *UnicodeG2_UTF16LE_1* Unicode Gen2 data element.

Single input data for Alpha-Numeric data element	input = Buffer.from('Protegrity1')
Single input data for Unicode Gen2 data element	byteInputUTF16LE = Buffer.from('Protegrity1', 'utf16le')
Bulk input data for Alpha-Numeric data element	input = [Buffer.from('Protegrity1'), Buffer.from('Protegrity2'), Buffer.from('Protegrity3')]
Bulk input data for Unicode Gen2 data element	byteInputUTF16LE = [Buffer.from('Protegrity1', 'utf16le'), Buffer.from('Protegrity2', 'utf16le'), Buffer.from('Protegrity3', 'utf16le')]

Operation

The following example shows how to perform the reprotect operation using the new *AlphaNum_1* Alpha-Numeric token data element.

```
const protectedData = protector.protect( input, 'user1', 'AlphaNum')

const reprotectData = protector.reprotect( protectedData.output, 'user1', 'AlphaNum',
'AlphaNum_1' )
```

The following example shows how to perform the reprotect operation using the new *UnicodeG2_UTF16LE_1* Unicode Gen2 token data element.

```
const protectedData = protector.protect( byteInputUTF16LE, 'user1',
'UnicodeG2_UTF16LE', null, null, protector.charset.utf16le)

const reprotectedData = protector.reprotect( protectedData.output, 'user1',
'UnicodeG2_UTF16LE', 'UnicodeG2_UTF16LE_1', null, null, null, null,
protector.charset.utf16le)
```

Result

Example output for single input data for Alpha-Numeric data element:

```
{ returncode: 50, output: <Buffer 4f 67 44 75 61 59 65 41 6a 61 39> }
```

Example output for single input data for Unicode Gen2 data element:

```
{ returncode: 50, output: <Buffer 4f00 6700 4400 7500 6100 5900 6500 4100 6a00 6100
3900> }
```

Example output for bulk input data for Alpha-Numeric data element:

```
{
  returncode: [ 50, 50, 50 ],
  output: [
    <Buffer 4f 67 44 75 61 59 65 41 6a 61 39>,
```

```
    <Buffer 59 72 4b 78 66 6d 64 54 47 75 36>,
    <Buffer 51 72 4b 78 66 6d 64 54 47 69 34>
  ],
  isSuccess: true
}
```

Example output for bulk input data for Unicode Gen2 data element:

```
{
  returncode: [ 50, 50, 50 ],
  output: [
    <Buffer 4f00 6700 4400 7500 6100 5900 6500 4100 6a00 6100 3900>,
    <Buffer 5900 7200 4b00 7800 6600 6d00 6400 5400 4700 7500 3600>,
    <Buffer 5100 7200 4b00 7800 6600 6d00 6400 5400 4700 6900 3400>
  ],
  isSuccess: true
}
```

2.5.7.7 Example - Retokenizing Bytes Data with External IV

This section describes how to use the *reprotect* API for retokenizing the bytes data using external IV.

Warning:

If you are retokenizing the data using the *reprotect* API, then the old data element and the new data element must have the same tokenization type.

For example, if you have used the Alpha-Numeric data element to protect the data, then you must use only the Alpha-Numeric data element to reprotect the data.

Input

In this example, the following input data are used, which are tokenized using the *AlphaNum* Alpha-Numeric data element and *UnicodeG2_UTF16LE* Unicode Gen2 token data element, with the help of the external IV *1234* that is passed as bytes. The protected input data is reprotected using a new *AlphaNum* Alpha-Numeric data element and a new *UnicodeG2_UTF16LE* Unicode Gen2 token data element, with the help of the external IV *123456* that is passed as bytes.

Single input data for Alpha-Numeric data element	input = Buffer.from('Protegrity1')
Single input data for Unicode Gen2 data element	byteInputUTF16LE = Buffer.from('Protegrity1', 'utf16le')
Bulk input data for Alpha-Numeric data element	input = [Buffer.from('Protegrity1'), Buffer.from('Protegrity2'), Buffer.from('Protegrity3')]
Bulk input data for Unicode Gen2 data element	byteInputUTF16LE = [Buffer.from('Protegrity1', 'utf16le'), Buffer.from('Protegrity2', 'utf16le'), Buffer.from('Protegrity3', 'utf16le')]

Operation

The following example shows how to perform the reprotect operation using the *AlphaNum* Alpha-Numeric token data element, with the help of the external IV *123456* that is passed as bytes.

```
const protectedData = protector.protect( input, 'user1', 'AlphaNum',
Buffer.from('1234'))

const reprotectedData = protector.reprotect( protectedData.output, 'user1', 'AlphaNum',
'AlphaNum', Buffer.from('1234'), Buffer.from('123456') )
```

The following example shows how to perform the reprotect operation using the *UnicodeG2_UTF16LE* Unicode Gen2 token data element, with the help of the external IV *123456* that is passed as bytes.

```
const protectedData = protector.protect( byteInputUTF16LE, 'user1',
'UnicodeG2_UTF16LE', Buffer.from('1234'), null, protector.charset.utf16le)
```

```
const reprotectedData = protector.reprotect( protectedData.output, 'user1',
'UnicodeG2_UTF16LE', 'UnicodeG2_UTF16LE', Buffer.from('1234'), Buffer.from('123456'),
protector.charset.utf16le)
```

Result

Example output for single input data for Alpha-Numeric data element:

```
{ returncode: 50, output: <Buffer 50 38 50 43 6d 43 38 67 74 79 31> }
```

Example output for single input data for Unicode Gen2 data element:

```
{ returncode: 50, output: <Buffer 5000 3800 5000 4300 6d00 4300 3800 6700 7400 7900
3100> }
```

Example output for bulk input data for Alpha-Numeric data element:

```
{
  returncode: [ 50, 50, 50 ],
  output: [
    <Buffer 50 38 50 43 6d 43 38 67 74 79 31>,
    <Buffer 50 55 56 72 6a 46 62 37 74 79 32>,
    <Buffer 50 47 57 47 4f 72 68 57 74 79 33>
  ],
  isSuccess: true
}
```

Example output for bulk input data for Unicode Gen2 data element:

```
{
  returncode: [ 50, 50, 50 ],
  output: [
    <Buffer 5000 3800 5000 4300 6d00 4300 3800 6700 7400 7900 3100>,
    <Buffer 5000 5500 5600 7200 6a00 4600 6200 3700 7400 7900 3200>,
    <Buffer 5000 4700 5700 4700 4f00 7200 6800 5700 7400 7900 3300>
  ],
  isSuccess: true
}
```

2.5.7.8 Example - Re-encrypting Bytes Data

This section describes how to use the *reprotect* API for re-encrypting the bytes data.

Warning:

If you are re-encrypting the data using the *reprotect* API, then the old data element and the new data element must have the same encryption type.

For example, if you have used the AES256 data element to protect the data, then you must use only the AES256 data element to reprotect the data.

Warning: To avoid data corruption, do not convert the encrypted bytes data into string format. It is recommended to convert the encrypted bytes data to a Hexadecimal, Base 64, or any other appropriate format.

Input

In this example, the following input data are used, which are first converted to bytes using the Buffer class in NodeJS. The bytes data is then encrypted using the *AES256* data element.

Single input data	input = Buffer.from('Protegrity1')

Bulk input data	input = [Buffer.from('Protegrity1'), Buffer.from('Protegrity2'), Buffer.from('Protegrity3')]

Operation

The following example shows how to perform the reprotect operation using the *AES256* encryption data element. The protected input data is re-encrypted using a new *AES256_1* encryption data element.

```
const protectedData = protector.protect( input, 'user1', 'AES256')

const reprotectData = protector.reprotect( protectedData.output, 'user1', 'AES256',
'AES256_1' )
```

Result

Example output for single input data:

```
{
  returncode: 50,
  output: <Buffer 81 65 k4 5g 3d 8j 7c 78 6f 45 9b 97 ap 9t 93 61>
}
```

Example output for bulk input data:

```
{
  returncode: [ 50, 50, 50 ],
  output: [
    <Buffer 81 65 k4 5g 3d 8j 7c 78 6f 45 9b 97 ap 9t 93 61>,
    <Buffer 8h 70 34 g4 4t c8 s9 2k 63 10 2j 39 25 hw f8 t5>,
    <Buffer r7 4j 7g ck 23 s4 8f 17 89 27 in 5n 60 hh s0 2m>
  ],
  isSuccess: true
}
```

2.6 Application Protector (AP) .Net APIs

A Trusted Application must be added in the datastore for running AP .Net. The AP .Net accesses the information on the Trusted Application from the policy stored in the memory. If the application is trusted, then the user can invoke the protect, unprotect, or reprotect APIs, depending on the requirements. You can flush the audits at the point where the application exits.

> **Note:** The AP .Net APIs can be invoked by a valid *Policy User* or a *Trusted Application* user.

> **Note:**
> When a short running application has completed its execution (in less than a second), the audit logs will not be seen. In such cases, the *FlushAudits()* API needs to be invoked.
>
> For more information about the FlushAudits API, refer to the section *FlushAudits API*.

The following diagram represents the basic flow of the AP .Net.

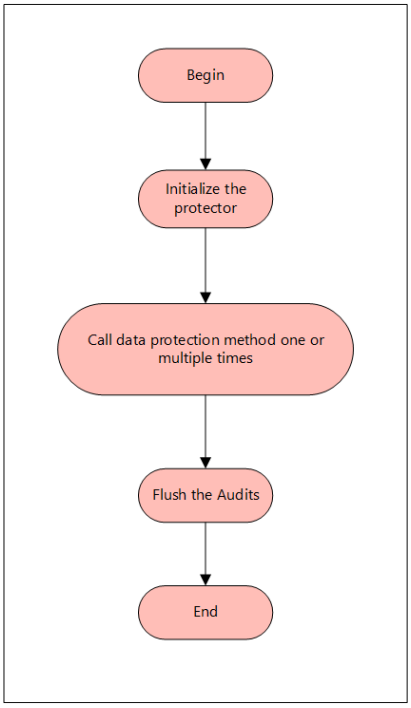

Figure 2-6: Flowchart for AP .Net

> **Note:**
> The AP .Net supports only bytes and string data type.

The following sections provide detailed information for the various functions used by the Protegrity Application Protector .Net.

2.6.1 GetProtector API

The *GetProtector* API returns the Protector object associated with the AP .Net API.

public static Protector GetProtector(int communicationID = 0)

Parameters

communicationID: The Communication ID in integer format that is used by the PEP server. This value must match the value specified by the Communication ID parameter in the *pepserver.cfg* file. This parameter is optional. The default value of the *communicationID* parameter is *0*.

> **Note:** Ensure that the *communicationID* parameter value is equal to the *Communication ID* parameter value in the *pepserver.cfg* file.

Returns

Protector Instance: Object associated with the AP .Net API.

Exception

ProtectorException: If the configuration is invalid, then an exception is returned.

Example

```
Protector protector = Protector.GetProtector();
```

2.6.2 GetVersion API

The *GetVersion* API returns the extended version of the AP .Net in use. The extended version consists of the AP .Net version number and the PEP server version.

public string GetVersion()

Parameters

None

Returns

string: Returns an object with product version of the installed AP .Net and the PEP server version.

Exception

None

Example

```
protector.GetVersion();
```

2.6.3 CheckAccess API

The *CheckAccess* API returns the access permission status of the user for a specified data element.

public bool CheckAccess(string userName, string dataElement, int accessType)

Parameters

userName: String containing the username defined in the policy.

dataElement: String containing the name of the data element defined in the policy.

checkAccess: Type of the access permission of the user for the specified data element. You can specify a value for this parameter from the accessType constants, such as, PROTECT, UNPROTECT, or REPROTECT.

Returns

boolean: Returns *true* if the user has the requested access on the data element and *false* if the user does not have access to the data element.

Exception

ProtectorException: If the CheckAccess operation is unsuccessful, then an exception is thrown.

Example

```
bool access = protector.CheckAccess("user1", "Alphanum", CheckAccessType.PROTECT);
bool access = protector.CheckAccess("user1", "Alphanum", CheckAccessType.UNPROTECT);
bool access = protector.CheckAccess("user1", "Alphanum", CheckAccessType.REPROTECT);
```

2.6.4 FlushAudits API

The *FlushAudits* API is used for flushing the audit logs at any point in the application. This API is required for a short running process that lasts less than a second, to get the audit logs.

> **Note:**
> It is recommended to invoke *FlushAudits* API at the point where the application exits.
>
> If the *FlushAudits* API is invoked after every operation, then the performance of the protector might get impacted.

public void FlushAudits()

Parameters

None

Returns

None

Exception

None

Example

```
protector.FlushAudits();
```

2.6.5 Protect - String API

This API protects the data provided as a string using a tokenization or Format Preserving Encryption data elements.

public string Protect(string input, string userName, string dataElementName, byte[] externalIv = null, byte[] externalTweak = null)

Parameters

input: Input data to be protected in string format.

userName: String containing the user name defined in the policy.

dataElementName: String containing the data element name defined in the policy.

externalIv: External IV is an optional argument. It is a buffer containing data that is used as an initialization vector and accepts input in byte format. When the external IV is null, its value is ignored.

externalTweak: External Tweak is an optional argument that is used only for the FPE data elements. It is a buffer containing data that is used as an external tweak and accepts input in byte format. When the external tweak is empty, its value is ignored.

Returns

string: Protected data in string format.

Exception

ProtectorException: If the protect operation is unsuccessful, then an exception is thrown.

Example

```
string singleProt = protector.Protect("Protegrity1234", "user1", "AlphaNum",
Encoding.UTF8.GetBytes("abcd123"), null);
```

2.6.6 Protect - Bulk String API

This API protects the data provided as a string array using a tokenization or Format Preserving Encryption data elements.

It supports bulk protection. There is no maximum data limit. However, you are recommended to pass not more than 1 MB of input data for each protection call.

public Tuple<string[], int[]> Protect(string[] input, string userName, string dataElementName, byte[] externalIv = null, byte[] externalTweak = null)

Parameters

input: Input array to be protected in string format.

userName: String containing the user name defined in the policy.

dataElementName: String containing the data element name defined in the policy.

externalIv: External IV is an optional argument. It is a buffer containing data that is used as an initialization vector and accepts input in byte format. When the external IV is null, its value is ignored.

externalTweak: External Tweak is an optional argument that is used only for the FPE data elements. It is a buffer containing data that is used as an external tweak and accepts input in byte format. When the external tweak is empty, its value is ignored.

Returns

Tuple<string[], int[]>: Returns a tuple of the following data:

- String array of the protected data

- Int array of the return codes

Exception

ProtectorException: If the protect operation is unsuccessful, then an exception is thrown. For string array, an exception is not thrown for error codes 22, 23, and 44. Instead, an error list is returned for the individual items in the bulk data.

Example

```
Tuple<string[], int[]> prot = protector.Protect({"Protegrity1", "Protegrity2",
"Protegrity3"}, "user1", "AlphaNum", Encoding.UTF8.GetBytes("abcd123"), null);
```

2.6.7 Protect - Byte API

This API protects the data provided as bytes using an encryption or a tokenization data element.

public Protect(byte[] input, string userName, string dataElementName, byte[] externalIv = null, byte[] externalTweak = null, int charset = Charset.UTF8)

Parameters

input: Input data to be protected in byte format.

userName: String containing the user name defined in the policy.

dataElementName: String containing the data element name defined in the policy.

externalIv: External IV is an optional argument. It is a buffer containing data that is used as an initialization vector and accepts input in byte format. When the external IV is null, its value is ignored.

externalTweak: External Tweak is an optional argument that is used only for the FPE data elements. It is a buffer containing data that is used as an external tweak and accepts input in byte format. When the external tweak is empty, its value is ignored.

charset: Charset is an optional argument. It indicates the encoding associated with the bytes of the input data. You can specify a value for this argument from the charset constants, such as, UTF8, UTF16LE, or UTF16BE.

Important:

The charset parameter is mandatory for the data elements created with Unicode Gen2 tokenization method and the FPE encryption method.

The encoding set for the charset parameter must match the encoding of the input data passed.

Note: The default value for the charset argument is UTF-8.

Note: The charset argument is only applicable for the input data of byte type.

Returns

byte[]: Protected data in byte format.

Exception

ProtectorException: If the protect operation is unsuccessful, then an exception is thrown.

Example

```
byte[] singleByteProt = protector.Protect(Encoding.Unicode.GetBytes("Protegrity123"),
"user1", "UnicodeGen2_UTF16LE", Encoding.UTF8.GetBytes("abcd123"), null,
charset:Charset.UTF16LE);
```

2.6.8 Protect - Bulk Byte API

This API protects the data provided as a byte array using an encryption or a tokenization data element.

It supports bulk protection. There is no maximum data limit. However, you are recommended to pass not more than 1 MB of input data for each protection call.

public Tuple<List<byte[]>, int[]> Protect(List<byte[]> input, string userName, string dataElementName, byte[] externalIv = null, byte[] externalTweak = null, int charset = Charset.UTF8)

Parameters

input: Input array to be protected in byte format.

userName: String containing the user name defined in the policy.

dataElementName: String containing the data element name defined in the policy.

externalIv: External IV is an optional argument. It is a buffer containing data that is used as an initialization vector and accepts input in byte format. When the external IV is null, its value is ignored.

externalTweak: External Tweak is an optional argument that is used only for the FPE data elements. It is a buffer containing data that is used as an external tweak and accepts input in byte format. When the external tweak is empty, its value is ignored.

charset: Charset is an optional argument. It indicates the encoding associated with the bytes of the input data. You can specify a value for this argument from the charset constants, such as, UTF8, UTF16LE, or UTF16BE.

Important:

The charset parameter is mandatory for the data elements created with Unicode Gen2 tokenization method and the FPE encryption method.

The encoding set for the charset parameter must match the encoding of the input data passed.

Note: The default value for the charset argument is UTF-8.

Note: The charset argument is only applicable for the input data of byte type.

Returns

Tuple<List<byte[]>, int[]>: Returns a tuple of the following data:

- List of byte arrays of the protected data
- Int array of the return codes

Exception

ProtectorException: If the protect operation is unsuccessful, then an exception is thrown. For byte array, an exception is not thrown for error codes 22, 23, and 44. Instead, an error list is returned for the individual items in the bulk data.

Example

```
Tuple<List<byte[]>, int[]>  bProt  =
protector.Protect({Encoding.BigEndianUnicode.GetBytes("Protegrity123"),
Encoding.BigEndianUnicode.GetBytes("Protegrity12345")}, "user1", "UnicodeGen2_UTF16BE",
Encoding.UTF8.GetBytes("abcd123"), null, charset:Charset.UTF16BE);
```

2.6.9 Unprotect - String API

This API unprotects the data provided as a string using a tokenization or Format Preserving Encryption data elements.

public string Unprotect(string input, string userName, string dataElementName, byte[] externalIv = null, byte[] externalTweak = null)

Parameters

input: Input data to be unprotected in string format.

userName: String containing the user name defined in the policy.

dataElementName: String containing the data element name defined in the policy.

externalIv: External IV is an optional argument. It is a buffer containing data that is used as an initialization vector and accepts input in byte format. When the external IV is null, its value is ignored.

externalTweak: External Tweak is an optional argument that is used only for the FPE data elements. It is a buffer containing data that is used as an external tweak and accepts input in byte format. When the external tweak is empty, its value is ignored.

Returns

string: Unprotected data in string format.

Exception

ProtectorException: If the unprotect operation is unsuccessful, then an exception is thrown.

Example

```
string singleProt = protector.Protect("Protegrity1234", "user1", "AlphaNum",
Encoding.UTF8.GetBytes("abcd123"), null);

string singleUnprot = protector.Unprotect(singleProt, "user1", "AlphaNum",
Encoding.UTF8.GetBytes("abcd123"), null);
```

2.6.10 Unprotect - Bulk String API

This API unprotects the data provided as a string array using a tokenization or Format Preserving Encryption data elements.

It supports bulk unprotection. There is no maximum data limit. However, you are recommended to pass not more than 1 MB of input data for each unprotection call.

public Tuple<string[],int[]> Unprotect(string[] input, string userName, string dataElementName, byte[] externalIv = null, byte[] externalTweak = null)

Parameters

input: Input array to be unprotected in string format.

userName: String containing the user name defined in the policy.

dataElementName: String containing the data element name defined in the policy.

externalIv: External IV is an optional argument. It is a buffer containing data that is used as an initialization vector and accepts input in byte format. When the external IV is null, its value is ignored.

externalTweak: External Tweak is an optional argument that is used only for the FPE data elements. It is a buffer containing data that is used as an external tweak and accepts input in byte format. When the external tweak is empty, its value is ignored.

Returns

Tuple<string[], int[]>: Returns a tuple of the following data:

- String array of the unprotected data
- Int array of the return codes

Exception

ProtectorException: If the unprotect operation is unsuccessful, then an exception is thrown. For string array, an exception is not thrown for error codes 22, 23, and 44. Instead, an error list is returned for the individual items in the bulk data.

Example

```
Tuple<string[], int[]> prot = protector.Protect({"Protegrity1", "Protegrity2",
"Protegrity3"}, "user1", "AlphaNum", Encoding.UTF8.GetBytes("abcd123"), null);

Tuple<string[], int[]> unprot = protector.Unprotect(prot.Item1, "user1", "AlphaNum",
Encoding.UTF8.GetBytes("abcd123"), null);
```

2.6.11 Unprotect - Byte API

This API unprotects the data provided as bytes using an encryption or a tokenization data element.

public Unprotect(byte[] input, string userName, string dataElementName, byte[] externalIv = null, byte[] externalTweak = null, int charset = Charset.UTF8)

Parameters

input: Input data to be unprotected in byte format.

userName: String containing the user name defined in the policy.

dataElementName: String containing the data element name defined in the policy.

externalIv: External IV is an optional argument. It is a buffer containing data that is used as an initialization vector and accepts input in byte format. When the external IV is null, its value is ignored.

externalTweak: External Tweak is an optional argument that is used only for the FPE data elements. It is a buffer containing data that is used as an external tweak and accepts input in byte format. When the external tweak is empty, its value is ignored.

charset: Charset is an optional argument. It indicates the encoding associated with the bytes of the input data. You can specify a value for this argument from the charset constants, such as, UTF8, UTF16LE, or UTF16BE.

Important:

The charset parameter is mandatory for the data elements created with Unicode Gen2 tokenization method and the FPE encryption method.

The encoding set for the charset parameter must match the encoding of the input data passed.

Note: The default value for the charset argument is UTF-8.

Note: The charset argument is only applicable for the input data of byte type.

Returns

byte[]: Unprotected data in byte format.

Exception

ProtectorException: If the unprotect operation is unsuccessful, then an exception is thrown.

Example

```
byte[] singleByteProt = protector.Protect(Encoding.Unicode.GetBytes("Protegrity123"),
"user1", "UnicodeGen2_UTF16LE", Encoding.UTF8.GetBytes("abcd123"), null, charset:
Charset.UTF16LE);

byte[] singleByteUnprot = protector.Unprotect(singleByteProt,
"user1", "UnicodeGen2_UTF16LE", Encoding.UTF8.GetBytes("abcd123"), null,
charset:Charset.UTF16LE);
```

2.6.12 Unprotect - Bulk Byte API

This API unprotects the data provided as a byte array using an encryption or a tokenization data element.

It supports bulk unprotection. There is no maximum data limit. However, you are recommended to pass not more than 1 MB of input data for each unprotection call.

public Tuple<List<byte[]>, int[]> Unprotect(List<byte[]> input, string userName, string dataElementName, byte[] externalIv = null, byte[] externalTweak = null, int charset = Charset.UTF8)

Parameters

input: Input array to be unprotected in byte format.

userName: String containing the user name defined in the policy.

dataElementName: String containing the data element name defined in the policy.

externalIv: External IV is an optional argument. It is a buffer containing data that is used as an initialization vector and accepts input in byte format. When the external IV is null, its value is ignored.

externalTweak: External Tweak is an optional argument that is used only for the FPE data elements. It is a buffer containing data that is used as an external tweak and accepts input in byte format. When the external tweak is empty, its value is ignored.

charset: Charset is an optional argument. It indicates the encoding associated with the bytes of the input data. You can specify a value for this argument from the charset constants, such as, UTF8, UTF16LE, or UTF16BE.

Important:

The charset parameter is mandatory for the data elements created with Unicode Gen2 tokenization method and the FPE encryption method.

The encoding set for the charset parameter must match the encoding of the input data passed.

Note: The default value for the charset argument is UTF-8.

Note: The charset argument is only applicable for the input data of byte type.

Returns

Tuple<List<byte[]>, int[]>: Returns a tuple of the following data:

- List of byte arrays of the protected data
- Int array of the return codes

Exception

ProtectorException: If the unprotect operation is unsuccessful, then an exception is thrown. For byte array, an exception is not thrown for error codes 22, 23, and 44. Instead, an error list is returned for the individual items in the bulk data.

Example

```
Tuple<List<byte[]>, int[]>  bProt  =
protector.Protect({Encoding.BigEndianUnicode.GetBytes("Protegrity123"),
Encoding.BigEndianUnicode.GetBytes("Protegrity12345")}, "user1", "UnicodeGen2_UTF16BE",
Encoding.UTF8.GetBytes("abcd123"), null, charset: Charset.UTF16BE);

Tuple<List<byte[]>, int[]> bUnprot = protector.Unprotect(bProt.Item1,
"user1", "UnicodeGen2_UTF16BE", Encoding.UTF8.GetBytes("abcd123"), null,
charset:Charset.UTF16BE);
```

2.6.13 Reprotect - String API

This API reprotects the data provided as a string using a tokenization or Format Preserving Encryption data elements.

Warning:
If you are using the reprotect API, then the old data element and the new data element must have the same data type.

For example, if you have used an Alpha-Numeric data element to protect the data, then you must use only Alpha-Numeric data element to reprotect the data.

public string Reprotect(string input, string userName, string oldDataElementName, string newDataElementName, byte[] oldExternalIv = null, byte[] newExternalIv = null, byte[] oldExternalTweak = null, byte[] newExternalTweak = null)

Parameters

input: Input data to be reprotected in string format.

userName: String containing the user name defined in the policy.

oldDataElementName: String containing the old data element name defined in the policy.

newDataElementName: String containing the new data element name defined in the policy.

oldExternalIv: Old external IV is an optional argument. It is a buffer containing data that is used as an initialization vector and accepts input in byte format. When the old external IV is null, its value is ignored.

newExternalIv: New external IV is an optional argument. It is a buffer containing data that is used as an initialization vector and accepts input in byte format. When the new external IV is null, its value is ignored.

oldExternalTweak: Old external Tweak is an optional argument that is used only for the FPE data elements. It is a buffer containing data that is used as an external tweak and accepts input in byte format. When the old external tweak is empty, its value is ignored.

newExternalTweak: New external Tweak is an optional argument that is used only for the FPE data elements. It is a buffer containing data that is used as an external tweak and accepts input in byte format. When the new external tweak is empty, its value is ignored.

Returns

string: Reprotected data in string format.

Exception

ProtectorException: If the reprotect operation is unsuccessful, then an exception is thrown.

Example

```
string singleProt = protector.Protect("Protegrity1234", "user1", "AlphaNum",
Encoding.UTF8.GetBytes("abcd123"), null);

string singleReprot = protector.Reprotect(singleProt, "user1", "AlphaNum",
"AlphaNum_1", Encoding.UTF8.GetBytes("abcd123"), Encoding.UTF8.GetBytes("abcd123456"),
null, null);
```

2.6.14 Reprotect - Bulk String API

This API reprotects the data provided as a string array using a tokenization or Format Preserving Encryption data elements.

Warning:
If you are using the reprotect API, then the old data element and the new data element must have the same data type.

For example, if you have used an Alpha-Numeric data element to protect the data, then you must use only Alpha-Numeric data element to reprotect the data.

It supports bulk reprotection. There is no maximum data limit. However, you are recommended to pass not more than 1 MB of input data for each reprotection call.

public Tuple<string[], int[]> Reprotect(string[] input, string userName, string oldDataElementName, string newDataElementName, byte[] oldExternalIv = null, byte[] newExternalIv = null, byte[] oldExternalTweak = null, byte[] newExternalTweak = null)

Parameters

input: Input array to be reprotected in string format.

userName: String containing the user name defined in the policy.

oldDataElementName: String containing the old data element name defined in the policy.

newDataElementName: String containing the new data element name defined in the policy.

oldExternalIv: Old external IV is an optional argument. It is a buffer containing data that is used as an initialization vector and accepts input in byte format. When the old external IV is null, its value is ignored.

newExternalIv: New external IV is an optional argument. It is a buffer containing data that is used as an initialization vector and accepts input in byte format. When the new external IV is null, its value is ignored.

oldExternalTweak: Old external Tweak is an optional argument that is used only for the FPE data elements. It is a buffer containing data that is used as an external tweak and accepts input in byte format. When the old external tweak is empty, its value is ignored.

newExternalTweak: New external Tweak is an optional argument that is used only for the FPE data elements. It is a buffer containing data that is used as an external tweak and accepts input in byte format. When the new external tweak is empty, its value is ignored.

Returns

Tuple<string[], int[]>: Returns a tuple of the following data:

- String array of the protected data
- Int array of the return codes

Exception

ProtectorException: If the reprotect operation is unsuccessful, then an exception is thrown. For string array, an exception is not thrown for error codes 22, 23, and 44. Instead, an error list is returned for the individual items in the bulk data.

Example

```
Tuple<string[], int[]> bulkProt = protector.Protect({"Protegrity1", "Protegrity2",
"Protegrity3"}, "user1", "AlphaNum", Encoding.UTF8.GetBytes("abcd123"), null);

Tuple<string[], int[]> bulkReprot = protector.Reprotect(bulkProt.Item1,
"user1", "AlphaNum", "AlphaNum_1", Encoding.UTF8.GetBytes("abcd123"),
Encoding.UTF8.GetBytes("abcd123456"), null, null);
```

2.6.15 Reprotect - Byte API

This API reprotects the data provided as bytes using an encryption or a tokenization data element.

Warning:
If you are using the reprotect API, then the old data element and the new data element must have the same data type.

For example, if you have used an Alpha-Numeric data element to protect the data, then you must use only Alpha-Numeric data element to reprotect the data.

public Reprotect(byte[] input, string userName, string oldDataElementName, string newDataElementName, byte[] oldExternalIv = null, byte[] newExternalIv = null, byte[] oldExternalTweak = null, byte[] newExternalTweak = null, int charset = Charset.UTF8)

Parameters

input: Input data to be reprotected in byte format.

userName: String containing the user name defined in the policy.

oldDataElementName: String containing the old data element name defined in the policy.

newDataElementName: String containing the new data element name defined in the policy.

oldExternalIv: Old external IV is an optional argument. It is a buffer containing data that is used as an initialization vector and accepts input in byte format. When the old external IV is null, its value is ignored.

newExternalIv: New external IV is an optional argument. It is a buffer containing data that is used as an initialization vector and accepts input in byte format. When the new external IV is null, its value is ignored.

oldExternalTweak: Old external Tweak is an optional argument that is used only for the FPE data elements. It is a buffer containing data that is used as an external tweak and accepts input in byte format. When the old external tweak is empty, its value is ignored.

newExternalTweak: New external Tweak is an optional argument that is used only for the FPE data elements. It is a buffer containing data that is used as an external tweak and accepts input in byte format. When the new external tweak is empty, its value is ignored.

charset: Charset is an optional argument. It indicates the encoding associated with the bytes of the input data. You can specify a value for this argument from the charset constants, such as, UTF8, UTF16LE, or UTF16BE.

> **Important:**
> The charset parameter is mandatory for the data elements created with Unicode Gen2 tokenization method and the FPE encryption method.
>
> The encoding set for the charset parameter must match the encoding of the input data passed.

> **Note:** The default value for the charset argument is UTF-8.

> **Note:** The charset argument is only applicable for the input data of byte type.

Returns

byte[]: Reprotected data in byte format.

Exception

ProtectorException: If the reprotect operation is unsuccessful, then an exception is thrown.

Example

```
byte[] singleByteProt = protector.Protect(Encoding.Unicode.GetBytes("Protegrity123"),
"user1", "UnicodeGen2_UTF16LE", Encoding.UTF8.GetBytes("abcd123"), null, charset:
Charset.UTF16LE);

byte[] singleByteReprot = protector.Reprotect(singleByteProt, "user1",
"UnicodeGen2_UTF16LE", "UnicodeGen2_UTF16LE_1", Encoding.UTF8.GetBytes("abcd123"),
Encoding.UTF8.GetBytes("abcd123456"), null, null, charset:Charset.UTF16LE);
```

2.6.16 Reprotect - Bulk Byte API

This API reprotects the data provided as a byte array using an encryption or a tokenization data element.

> **Warning:**
> If you are using the reprotect API, then the old data element and the new data element must have the same data type.

For example, if you have used an Alpha-Numeric data element to protect the data, then you must use only Alpha-Numeric data element to reprotect the data.

It supports bulk reprotection. There is no maximum data limit. However, you are recommended to pass not more than 1 MB of input data for each reprotection call.

public Tuple<List<byte[]>, int[]> Reprotect(List<byte[]> input, string userName, string oldDataElementName, string newDataElementName, byte[] oldExternalIv = null, byte[] newExternalIv = null, byte[] oldExternalTweak = null, byte[] newExternalTweak = null, int charset = Charset.UTF8)

Parameters

input: Input array to be reprotected in byte format.

userName: String containing the user name defined in the policy.

oldDataElementName: String containing the old data element name defined in the policy.

newDataElementName: String containing the new data element name defined in the policy.

oldExternalIv: Old external IV is an optional argument. It is a buffer containing data that is used as an initialization vector and accepts input in byte format. When the old external IV is null, its value is ignored.

newExternalIv: New external IV is an optional argument. It is a buffer containing data that is used as an initialization vector and accepts input in byte format. When the new external IV is null, its value is ignored.

oldExternalTweak: Old external Tweak is an optional argument that is used only for the FPE data elements. It is a buffer containing data that is used as an external tweak and accepts input in byte format. When the old external tweak is empty, its value is ignored.

newExternalTweak: New external Tweak is an optional argument that is used only for the FPE data elements. It is a buffer containing data that is used as an external tweak and accepts input in byte format. When the new external tweak is empty, its value is ignored.

charset: Charset is an optional argument. It indicates the encoding associated with the bytes of the input data. You can specify a value for this argument from the charset constants, such as, UTF8, UTF16LE, or UTF16BE.

Important:

The charset parameter is mandatory for the data elements created with Unicode Gen2 tokenization method and the FPE encryption method.

The encoding set for the charset parameter must match the encoding of the input data passed.

Note: The default value for the charset argument is UTF-8.

Note: The charset argument is only applicable for the input data of byte type.

Returns

Tuple<List<byte[]>, int[]>: Returns a tuple of the following data:

- List of byte arrays of the protected data
- Int array of the return codes

Exception

ProtectorException: If the reprotect operation is unsuccessful, then an exception is thrown. For byte array, an exception is not thrown for error codes 22, 23, and 44. Instead, an error list is returned for the individual items in the bulk data.

Example

```
Tuple<List<byte[]>, int[]>  bProt  =
protector.Protect({Encoding.BigEndianUnicode.GetBytes("Protegrity123"),
```

```
Encoding.BigEndianUnicode.GetBytes("Protegrity12345")}, "user1", "UnicodeGen2_UTF16BE",
Encoding.UTF8.GetBytes("abcd123"), null, charset: Charset.UTF16BE);

Tuple<List<byte[]>, int[]> bReprot = protector.Reprotect(bProt.Item1, "user1",
"UnicodeGen2_UTF16BE", "UnicodeGen2_UTF16BE_1", Encoding.UTF8.GetBytes("abcd123"),
Encoding.UTF8.GetBytes("abcd123456"), null, null, charset:Charset.UTF16BE);
```

2.7 Application Protectors API Return Codes

When you develop an application using the API of the Protegrity AP C, AP Go, AP Java, AP NodeJS, AP .Net, and AP Python, you may encounter the errors described in this section.

2.7.1 AP C Error Return Codes

This section includes the list of error return codes for Application Protector C.

Error Number	Error Code	Code Definition
0	XC_FAILED	General fail with no detailed description.
1	XC_SUCCESS	Function call is successfully executed and return values are created.
100	XC_INVALID_PARAMETER	A parameter specified in a function call was invalid, or not within valid limits. An example would be when a null parameter is used when not null is expected. All parameters have to be initialized before a call. For instance, output parameters need to be initialized to a value, null, or zero if nothing else is specified.
101	XC_TIMEOUT	The operation timed out before a result was returned. A timeout can occur when you try to connect to a server and the server does not exist.
102	XC_ACCESS_DENIED	If you do not have the permissions to access an object or a file then you will receive a return code that the access is denied.
103	XC_NOT_SUPPORTED	The requested operation is not supported.
104	XC_SESSION_REFUSED	The remote peer did not accept the session request. Check if the server that you are trying to connect to is running.
105	XC_DISCONNECTED	The session was terminated. If you do not have a session and you try to use the session, then you will receive a return code that you are disconnected.
106	XC_UNREACHABLE	The host could not be reached or is not able to be contacted. If you cannot connect to the server, then try to start the server or change the parameters for the connection.
107	XC_SESSION_IN_USE	The session is already in use. You are trying to use a session that is already used.
108	XC_EOF	If you get an end of file that is unexpected, such as an empty key file, then you will receive a return code that an unexpected end of file is reached.
109	XC_NOT_FOUND	Error returned when a file required to complete an operation is not found.
110	XC_BUFFER_TOO_SMALL	If you try to encrypt, decrypt, or re-encrypt and the output buffer size is too small, then you will receive this return code.
111	XC_NOT_DEFINED	A property or setting has not been set or defined.
112	XC_POLICY_LOCKED	The policy is locked so no operations are allowed.
113	XC_PROTOCOL_ERROR	This can be caused by an invalid frame or similar, or formatting errors in the protocol message structure.
114	XC_COMMUNICATION_ERROR	This can happen when sending or receiving data over an SSL or TCP socket.
115	XC_THROW_EXCEPTION	Used when an operation should throw an exception.
116	XC_INVALID_FORMAT	Either the length or the contents of the provided input data are not in a valid format.

2.7.2 AP Java, AP Python, AP NodeJS, AP .Net, and AP Go API Return Codes

When you develop an application using the Application Protector (AP) Java APIs, AP Python APIs, AP NodeJS APIs, AP .Net APIs, or AP Go APIs, you may encounter the errors described in this section. You can avoid most of the errors if you use the API correctly.

For more information, refer to the *Protegrity Application Protector Guide 9.1.0.0*.

2.7.2.1 AP Java, AP Python, AP NodeJS, AP .Net, and AP Go API Log Return Error Codes

This section lists the log return error codes returned by the AP Java, AP Python, AP NodeJS, AP .Net, and AP Go API as a result of policy exceptions. Audits are generated in the ESA for these errors.

Table 2-3: AP Java, AP Python, AP NodeJS, AP .Net, and AP Go API Log Return Codes

Code	Error Message	Error Description
1	1, The username could not be found in the policy in shared memory.	The user name could not be found in the policy residing in the shared memory.
2	2, The data element could not be found in the policy in shared memory.	The data element could not be found in the policy residing in the shared memory.
3	3, The user does not have the appropriate permissions to perform the requested operation.	The user does not have the required permissions to perform the requested operation.
5	5, Integrity check failed.	The data integrity check failed when decrypting using a Data Element with CRC enabled.
6	6, Data protection was successful.	The operation to protect the data was successful.
7	7, Data protection failed.	The operation to protect the data failed.
8	8, Data unprotect operation was successful.	The operation to unprotect the data was successful.
9	9, Data unprotect operation failed.	The operation to unprotect the data failed.
10	10, The user has the appropriate permissions to perform the requested operation. This is just a policy check and no data has been protected or unprotected.	The user has the required permissions to perform the requested operation. This return code ensures a verification and no data is protected or unprotected.
11	11, Data unprotect operation was successful with use of an inactive keyid.	The operation to unprotect the data was successful using an inactive Key ID.
12	12, Input is null or not within allowed limits.	Input parameters are either NULL or not within allowed limits.
13	13, An internal error occured in a function call after the PEP provider is started.	Internal error occurring in a function call after the PEP provider has been opened. For example: - ***`failed to get mutex / semaphore, - unexpected null param`***.
14	14, Failed to load the data encryption key	A key for a data element could not be loaded from shared memory into the Crypto engine.
15	15, Tweak input is too long.	Tweak input is too long.
16	16, External IV is not supported in this version	External IV is not supported in this version.
17	17, Failed to initialize the PEP - This is a fatal error	The PEP server failed to initialize, which is a fatal error.
19	19, Unsupported tweak action for the specified fpe dataelement	The external tweak is not supported for the specified FPE data element.
20	20, Failed to allocate memory.	Failed to allocate memory.
21	21, Input or output buffer is too small.	The input or output buffer is very small.
22	22, Data is too short to be protected/unprotected.	The data is too short to be protected or unprotected.
23	23, Data is too long to be protected/unprotected.	The data is too long to be protected or unprotected.

Code	Error Message	Error Description
25	25, Username too long.	The user name is longer than the maximum supported length of the user name that can be used for protect or unprotect operations.
26	26, Unsupported algorithm or unsupported action for the specific data element.	The algorithm or action for the specific data element is unsupported.
27	27, Application has been authorized.	The application is authorized.
28	28, Application has not been authorized.	The application is not authorized.
31	31, The policy in shared memory is empty.	The policy residing in the shared memory is empty.
39	39, The policy in shared memory is locked. This can be caused by a disk full alert.	The policy residing in the shared memory is locked. This error can be caused by a *Disk Full* alert.
40	40, No valid license or current date is beyond the license expiration date.	The license is invalid or the current date is beyond the license expiry date.
41	41, The use of the protection method is restricted by license.	The use of the Protection method is restricted by the license.
42	42, Invalid license or time is before licensestart.	The license, or the time, is invalid prior to the start of the license tenure.
44	44, The content of the input data is not valid.	The content of the input data is invalid.
49	49, Unsupported input encoding for the specific data element.	The input encoding for the specific data element is not supported.
50	50, Data reprotect operation was successful.	The operation to reprotect the data was successful.

2.7.2.2 AP Java, AP Python, AP NodeJS, AP .Net, and AP Go API PEP Result Codes

This section lists the PEP result codes returned by AP Java, AP Python, AP NodeJS, AP .Net, and AP Go APIs as a result of system exceptions. Audits are not generated in the ESA for these errors.

Table 2-4: AP Java, AP Python, AP NodeJS, AP .Net, and AP Go API PEP Result Codes

Code	Error Message	Error Description
-1	-1, Invalid parameter	The parameter is invalid.
-7	-7, Error when parsing contents, e.g.	The error occurred when the contents were parsed.
-8	-8, Not found!	The search operation was not successful.
-16	-16, Buffer is too small	The buffer size is very small.
-26	-26, Policy not available	The policy is not available.
-43	-43, Invalid format	The format is invalid.
-46	-46, Requesting service/function on an object that is not initialized	The service requested or function is performed on an object that is not initialized.
-47	-47, Policy locked for some reason	The Policy is locked.

2.8 Environment Path Variables

This section lists the variables required for the environment paths for different operating systems.

When an Application Protector API is used and an application needs to load the PLM file, the environment variables are applied. These variables specify where the operating system will look for the PLM file to load and they are used to dynamically update the system behaviour.

OS	32-bit	64-bit	Delimiter
AIX	LIBPATH	LIBPATH	: (Colon)
HP-UX	SHLIB_PATH	LD_LIBRARY_PATH	: (Colon)
Solaris	LD_LIBRARY_PATH	LD_LIBRARY_PATH_64	: (Colon)

OS	32-bit	64-bit	Delimiter
Linux	LD_LIBRARY_PATH	LD_LIBRARY_PATH	: (Colon)
Windows	PATH	PATH	; (Semicolon)

Chapter 3

Big Data Protector

This section describes the APIs, UDFs, and Commands that are supported by the Protegrity Big Data Protector.

Note:
To reduce performance issues that occur due to protection of data or casting of data, a general best practice is to protect the data and present the unprotect APIs, UDFs, or Commands, as applicable, to authorized users only. This eliminates access of the unauthorized users to the unprotection APIs, UDFs, or Commands as the data is in protected form only.

The unprotection of protected data is therefore limited to authorized users and does not cause a significant performance impact as the APIs, UDFs, or Commands are executed restrictively.

Note: Starting from the Big Data Protector 7.2.0 release, the HDFS File Protector (HDFSFP) is deprecated. The HDFSFP-related sections are retained to ensure coverage for using an older version of Big Data Protector with the ESA 7.2.0.

3.1 MapReduce APIs

This section describes the MapReduce APIs available for protection and unprotection in the Big Data Protector to build secure Big Data applications.

Warning: The Protegrity MapReduce protector only supports *bytes* converted from the *string* data type.
If any other data type is directly converted to *bytes* and passed as input to the API that supports *byte* as input and provides *byte* as output, then data corruption might occur.

Caution: If you are using the Protect, or Unprotect, or Reprotect API which accepts byte as input and provides byte as output, then ensure that you encode the string input data type with the corresponding default encoding that is used in the data element.

For example, for Gen2 data element with the default encoding UTF16LE, ensure that you encode the string with the UTF16LE before passing it to the ByteIn or ByteOut APIs.

Note: If you perform a security operation on a single data item, then an exception appears in case of any error. Similarly, if you perform a security operation on bulk data, then an exception appears in case of any error except for the error codes 22, 23, and 44. Instead of an error message, the UDFs return an error list for the individual items in the bulk data. For more information about the API error return codes, refer to Table 8-2: PEP Log Return Codes in section 8 Appendix: Return Codes in the Protegrity Big Data Protector Guide 9.1.0.0.

If you are using the Bulk APIs for the MapReduce protector, then the following two modes for error handling and return codes are available:

- Default mode: Starting with the Big Data Protector, version 6.6.4, the Bulk APIs in the MapReduce protector will return the detailed error and return codes instead of *0* for *failure* and *1* for *success*. In addition, the MapReduce jobs involving Bulk APIs will provide error codes instead of throwing exceptions.

  For more information about the return codes for Big Data Protector, version 9.1.0.0, refer to *Big Data Protector Guide 9.1.0.0*.

- Backward compatibility mode: If you need to continue using the error handling capabilities provided with Big Data Protector, version 6.6.3 or lower, that is *0* for *failure* and *1* for *success*, then you can set this mode.

3.1.1 openSession()

This method opens a new user session for protect and unprotect operations. It is a good practice to create one session per user thread.

public synchronized int openSession(String parameter)

Parameters

parameter:An internal API requirement that should be set to 0.

Result

1: If session is successfully created

Example

```
ptyMapReduceProtector mapReduceProtector = new ptyMapReduceProtector();
int openSessionStatus = mapReduceProtector.openSession("0");
```

Exception (and Error Codes)

ptyMapRedProtectorException: if session creation fails

3.1.2 closeSession()

This function closes the current open user session. Every instance of *ptyMapReduceProtector* opens only one session, and a session ID is not required to close it.

public synchronized int closeSession()

Parameters

None

Result

1: If session is successfully closed

0: If session closure is a failure

Example

```
ptyMapReduceProtector mapReduceProtector = new ptyMapReduceProtector();
int openSessionStatus  = mapReduceProtector.openSession("0");
int closeSessionStatus = mapReduceProtector.closeSession();
```

Exception (and Error Codes)

None

3.1.3 flushAudits()

This method flushes any audit logs that the Mapreduce Protector generates. It must be called in the *Mapper.cleanup()* method to ensure that the logs are flushed at the end of all Mapper Tasks.

public void flushAudits()

Parameters

None

Result

None

Example

```
class CustomMapper extends Mapper {
    .
    .
    ptyMapReduceProtector mapReduceProtector = new ptyMapReduceProtector();
    .
    .
    @Override
    public void cleanup(Mapper.Context context){
        mapReduceProtector.flushAudits();
    }
}
```

Exception (and Error Codes)

No Exception thrown. If any exception is caught during flushing logs, it is logged using the default logger.

3.1.4 getVersion()

This function returns the current version of the MapReduce protector.

public java.lang.String getVersion()

Parameters

None

Result

This function returns the current version of MapReduce protector.

Example

```
ptyMapReduceProtector mapReduceProtector = new ptyMapReduceProtector();
String version = mapReduceProtector.getVersion();
```

3.1.5 getCurrentKeyId()

This method returns the current Key ID for the data element which contains the *KEY ID* attribute, while creating the data element, such as AES-256, AES-128, and so on.

public int getCurrentKeyId(java.lang.String dataElement)

Parameters

dataElement: Name of the data element

Result

This method returns the current Key ID for the data element containing the *KEY ID* attribute.

Example

```
ptyMapReduceProtector mapReduceProtector = new ptyMapReduceProtector();
int currentKeyId = mapReduceProtector.getCurrentKeyId("ENCRYPTION_DE");
```

3.1.6 checkAccess()

This method checks the access of the user for the specified data element.

public boolean checkAccess(java.lang.String dataElement, byte bAccessType)

Parameters

dataElement: Name of the data element

bAccessType: Type of the access of the user for the data element.

The following are the different values for the bAccessType variable:

DELETE	0x01
PROTECT	0x02
REPROTECT	0x04
UNPROTECT	0x08
CREATE	0x10
MANAGE	0x20

Result

1: If the user has access to the data element

Example

```
ptyMapReduceProtector mapReduceProtector = new ptyMapReduceProtector();
byte bAccessType = 0x02;
boolean isAccess = mapReduceProtector.checkAccess("DE_PROTECT" , bAccessType );
```

3.1.7 getDefaultDataElement()

This method returns default data element configured in security policy.

public String getDefaultDataElement(String policyName)

Parameters

policyName: Name of policy configured using Policy management in ESA.

Result

Default data element name configured in a given policy.

Example

```
ptyMapReduceProtector mapReduceProtector = new ptyMapReduceProtector();
String defaultDataElement = mapReduceProtector.getDefaultDataElement("my_policy");
```

Exception

ptyMapRedProtectorException: If unable to return default data element name

3.1.8 protect() - Byte array data

Protects the data provided as a byte *array*. The type of protection applied is defined by *dataElement*.

> **Note:** For Date and Datetime type of data elements, the protect API returns an invalid input data error if the input value falls between the non-existent date range from 05-OCT-1582 to 14-OCT-1582 of the Gregorian Calendar.
>
> For more information about the tokenization and de-tokenization of the cutover dates of the Proleptic Gregorian Calendar, refer to the section *Date* and *Datetime* tokenization in the *Protection Method Reference Guide 9.1.0.0*.

public byte[] protect(String dataElement, byte[] data)

Parameters

dataElement: Name of the data element to be protected. The *Protect* API also supports the HMAC data element for hashing the byte array data.

data: Byte *array* of data to be protected

> **Warning:**
> The Protegrity MapReduce protector only supports *bytes* converted from the *string* data type.
>
> If any other data type is directly converted to *bytes* and passed as input to the API that supports *byte* as input and provides *byte* as output, then data corruption might occur.

> **Note:** If you are using the *Protect* API which accepts *byte* as input and provides *byte* as output, then ensure that when unprotecting the data, the *Unprotect* API, with *byte* as input and *byte* as output is utilized. In addition, ensure that the *byte* data being provided as input to the Protect API has been converted from a *string* data type only.

Result

Byte array of protected data

Example

```
ptyMapReduceProtector mapReduceProtector = new ptyMapReduceProtector();
byte[] bResult        = mapReduceProtector.protect(
                          "DE_PROTECT","protegrity".getBytes());
```

Exception

ptyMapRedProtectorException: If unable to protect data

Table 3-1: Supported Protection Methods

MapReduce APIs	Supported Protection Methods					
	Tokenization	**Encryption**	**FPE**	**No Encryption**	**Masking**	**Monitoring**
protect() - Byte array data	• Numeric (0-9) • Credit Card • Alpha (A-Z) • Upper-case Alpha (A-Z) • Alpha-Numeric (0-9, a-z, A-Z) • Upper Alpha-Numeric (0-9, A-Z) • Lower ASCII • Printable • Date (YYYY-MM-DD, DD/MM/YYYY, MM.DD.YYYY) • Datetime (YYYY-MM-DD HH:MM:SS) • Decimal • Unicode (Gen2) • Unicode (Legacy)	• AES-128 • AES-256 • 3DES • CUSP	FPE (All - Encoded Byte's Charset should match Dataelement's Encoding Type)	Yes	Yes	Yes

MapReduce APIs	Supported Protection Methods					
	Tokenization	Encryption	FPE	No Encryption	Masking	Monitoring
	• Unicode (Base64 - Encoded Byte's Charset should match Dataelement's Encoding Type) • Binary • Email					

Note: The protection methods, that are not mentioned in the *Supported Protection Methods* table, are not supported.

3.1.9 protect() - Int data

Protects the data provided as *int*. The type of protection applied is defined by *dataElement*.

public int protect(String dataElement, int data)

Parameters

dataElement: Name of the data element to be protected

data:*int* to be protected

Result

Protected *int* data

Example

```
ptyMapReduceProtector mapReduceProtector = new ptyMapReduceProtector();
int bResult           = mapReduceProtector.protect(
                          "DE_PROTECT",1234);
```

Exception

ptyMapRedProtectorException: If unable to protect data

Table 3-2: Supported Protection Methods

MapReduce APIs	Supported Protection Methods					
	Tokenization	Encryption	FPE	No Encryption	Masking	Monitoring
protect() - Int data	Integer (4 Bytes)	No	No	Yes	No	Yes

Note: The protection methods, that are not mentioned in the *Supported Protection Methods* table, are not supported.

3.1.10 protect() - Long data

Protects the data provided as *long*. The type of protection applied is defined by *dataElement*.

public long protect(String dataElement, long data)

Parameters

dataElement: Name of the data element to be protected.

data : *long* data to be protected

Result

Protected *long* data

Example

```
ptyMapReduceProtector mapReduceProtector = new ptyMapReduceProtector();
long  bResult           = mapReduceProtector.protect(
                          "DE_PROTECT",123412341234);
```

Exception

ptyMapRedProtectorException: If unable to protect data

Table 3-3: Supported Protection Methods

MapReduce APIs	Supported Protection Methods					
	Tokenization	**Encryption**	**FPE**	**No Encryption**	**Masking**	**Monitoring**
protect() - Long data	Integer (8 Bytes)	No	No	Yes	No	Yes

Note: The protection methods, that are not mentioned in the *Supported Protection Methods* table, are not supported.

3.1.11 unprotect() - Byte array data

This function returns the data in its original form.

Note: For Date and Datetime type of data elements, the protect API returns an invalid input data error if the input value falls between the non-existent date range from 05-OCT-1582 to 14-OCT-1582 of the Gregorian Calendar.

For more information about the tokenization and de-tokenization of the cutover dates of the Proleptic Gregorian Calendar, refer to the section *Date* and *Datetime* tokenization in the *Protection Method Reference Guide 9.1.0.0*.

public byte[] unprotect(String dataElement, byte[] data)

Parameters

dataElement: Name of data element to be unprotected

data: *array* of data to be unprotected

Note:

The Protegrity MapReduce protector only supports *bytes* converted from the *string* data type.

If any other data type is directly converted to *bytes* and passed as input to the API that supports *byte* as input and provides *byte* as output, then data corruption might occur.

Result

Byte *array* of unprotected data

Example

```
ptyMapReduceProtector mapReduceProtector = new ptyMapReduceProtector();
byte[] protectedResult   = mapReduceProtector.protect( "DE_PROTECT_UNPROTECT",
                           "protegrity".getBytes() );
byte[] unprotectedResult = mapReduceProtector.unprotect(
                           "DE_PROTECT_UNPROTECT", protectedResult );
```

Exception

ptyMapRedProtectorException: If unable to unprotect data

Table 3-4: Supported Protection Methods

MapReduce APIs	Supported Protection Methods					
	Tokenization	**Encryption**	**FPE**	**No Encryption**	**Masking**	**Monitoring**
unprotect() - Byte array data	• Numeric (0-9) • Credit Card • Alpha (A-Z) • Upper-case Alpha (A-Z) • Alpha-Numeric (0-9, a-z, A-Z) • Upper Alpha-Numeric (0-9, A-Z) • Lower ASCII • Printable • Date (YYYY-MM-DD, DD/MM/YYYY, MM.DD.YYYY) • Datetime (YYYY-MM-DD HH:MM:SS) • Decimal • Unicode (Gen2) • Unicode (Legacy) • Unicode (Base64 - Encoded Byte's Charset should match Dataelement's Encoding Type) • Binary • Email	• AES-128 • AES-256 • 3DES • CUSP	FPE (All - Encoded Byte's Charset should match Dataelement's Encoding Type)	Yes	Yes	Yes

Note: The protection methods, that are not mentioned in the *Supported Protection Methods* table, are not supported.

3.1.12 unprotect() - Int data

This function returns the data in its original form.

public int unprotect(String dataElement, int data)

Parameters

dataElement: Name of data element to be unprotected

data: *int* to be unprotected

Result

Unprotected *int* data

Example

```
ptyMapReduceProtector mapReduceProtector = new ptyMapReduceProtector();
int protectedResult   = mapReduceProtector.protect( "DE_PROTECT_UNPROTECT",
                                1234 );
```

```
int unprotectedResult = mapReduceProtector.unprotect(
                            "DE_PROTECT_UNPROTECT", protectedResult );
```

Exception

ptyMapRedProtectorException: If unable to unprotect data

Table 3-5: Supported Protection Methods

MapReduce APIs	Supported Protection Methods					
	Tokenization	**Encryption**	**FPE**	**No Encryption**	**Masking**	**Monitoring**
unprotect() - Int data	Integer (4 Bytes)	No	No	Yes	No	Yes

Note: The protection methods, that are not mentioned in the *Supported Protection Methods* table, are not supported.

3.1.13 unprotect() - Long data

This function returns the data in its original form.

public long unprotect(String dataElement, long data)

Parameters

dataElement: Name of data element to be unprotected

data: *long* data to be unprotected

Result

Unprotected *long* data

Example

```
ptyMapReduceProtector mapReduceProtector = new ptyMapReduceProtector();
long protectedResult   = mapReduceProtector.protect( "DE_PROTECT_UNPROTECT",
                             123412341234 );
long unprotectedResult = mapReduceProtector.unprotect(
                             "DE_PROTECT_UNPROTECT", protectedResult );
```

Exception

ptyMapRedProtectorException: If unable to unprotect data

Table 3-6: Supported Protection Methods

MapReduce APIs	Supported Protection Methods					
	Tokenization	**Encryption**	**FPE**	**No Encryption**	**Masking**	**Monitoring**
unprotect() - Long data	Integer (8 Bytes)	No	No	Yes	No	Yes

Note: The protection methods, that are not mentioned in the *Supported Protection Methods* table, are not supported.

3.1.14 bulkProtect() - Byte array data

This is used when a set of data needs to be protected in a bulk operation. It helps to improve performance.

Note: For Date and Datetime type of data elements, the protect API returns an invalid input data error if the input value falls between the non-existent date range from 05-OCT-1582 to 14-OCT-1582 of the Gregorian Calendar.

For more information about the tokenization and de-tokenization of the cutover dates of the Proleptic Gregorian Calendar, refer to the section *Date* and *Datetime* tokenization in the *Protection Method Reference Guide 9.1.0.0*.

public byte[][] bulkProtect(String dataElement, List <Integer> errorIndex, byte[][] inputDataItems)

Parameters

dataElement: Name of data element to be protected. The *bulkProtect* API also supports the HMAC data element for hashing the byte array data.

errorIndex: *array* used to store all error indices encountered while protecting each data entry in *inputDataItems*

inputDataItems: Two-dimensional *array* to store bulk data for protection

> **Note:** If you are using Format Preserving Encryption (FPE) and Byte APIs, then ensure that the encoding, which is used to convert the *string* input data to *bytes*, matches the encoding that is selected in the **Plaintext Encoding** drop-down for the required FPE data element.

Result

Two-dimensional byte array of protected data.

If the Backward Compatibility mode is not set, then the appropriate error code appears. For more information about the return codes, refer to *Table 10-2 PEP Log Return Codes* and *Table 10-3 PEP Result Codes* in the *Big Data Protector Guide 9.1.0.0*.

If the Backward Compatibility mode is set, then the Error Index includes one of the following values, per entry in the bulk protect operation:

- 1: The protect operation for the entry is successful.
- 0: The protect operation for the entry is unsuccessful.

  For more information about the failed entry, view the logs available in ESA Forensics.
- Any other value or garbage return value: The protect operation for the entry is unsuccessful. For more information about the failed entry, view the logs available in ESA Forensics.

Example

```
ptyMapReduceProtector mapReduceProtector = new ptyMapReduceProtector();
List<Integer> errorIndex = new ArrayList<Integer>();

byte[][] protectData     = {"protegrity".getBytes{}, "protegrity".getBytes(),
                            "protegrity".getBytes(), "protegrity".getBytes()};

byte[][] protectedData = mapReduceProtector.bulkProtect( "DE_PROTECT",
            errorIndex, protectData );

System.out.print("Protected Data: ");
for(int i = 0; i < protectedData.length; i++)
{
      //THIS WILL PRINT THE PROTECTED DATA
    System.out.print(protectedData[i] == null ? null : new String(protectedData[i]));
    if(i < protectedData.length - 1)
    {
       System.out.print(",");
    }
}

System.out.println("");
System.out.print("Error Index: ");
for(int i = 0; i < errorIndex.size(); i++)
{
     System.out.print(errorIndex.get( i ));
     if(i < errorIndex.size() - 1)
     {
       System.out.print(",");
     }
}
//ABOVE CODE WILL PRINT THE ERROR INDEXES
```

Exception

ptyMapRedProtectorException: If an error is encountered during bulk protection of data

Table 3-7: Supported Protection Methods

MapReduce APIs	Supported Protection Methods					
	Tokenization	**Encryption**	**FPE**	**No Encryption**	**Masking**	**Monitoring**
bulkProtect() - Byte array data	• Numeric (0-9) • Credit Card • Alpha (A-Z) • Upper-case Alpha (A-Z) • Alpha-Numeric (0-9, a-z, A-Z) • Upper Alpha-Numeric (0-9, A-Z) • Lower ASCII • Printable • Date (YYYY-MM-DD, DD/MM/YYYY, MM.DD.YYYY) • Datetime (YYYY-MM-DD HH:MM:SS) • Decimal • Unicode (Gen2) • Unicode (Legacy) • Unicode (Base64 - Encoded Byte's Charset should match Dataelement's Encoding Type) • Binary • Email	• AES-128 • AES-256 • 3DES • CUSP	FPE (All - Encoded Byte's Charset should match Dataelement's Encoding Type)	Yes	Yes	Yes

Note: The protection methods, that are not mentioned in the *Supported Protection Methods* table, are not supported.

3.1.15 bulkProtect() - Int data

This is used when a set of data needs to be protected in a bulk operation. It helps to improve performance.

public int[] bulkProtect(String dataElement, List <Integer> errorIndex, int[] inputDataItems)

Parameters

dataElement: Name of data element to be protected

errorIndex : *array* used to store all error indices encountered while protecting each data entry in input Data Items

inputDataItems: *array* to store bulk *int* data for protection

Result

int array of protected data

If the Backward Compatibility mode is not set, then the appropriate error code appears. For more information about the return codes, refer to *Table 10-2 PEP Log Return Codes* and *Table 10-3 PEP Result Codes* in the *Big Data Protector Guide 9.1.0.0*.

If the Backward Compatibility mode is set, then the Error Index includes one of the following values, per entry in the bulk protect operation:

- 1: The protect operation for the entry is successful.
- 0: The protect operation for the entry is unsuccessful.
  - For more information about the failed entry, view the logs available in ESA Forensics.
- Any other value or garbage return value: The protect operation for the entry is unsuccessful. For more information about the failed entry, view the logs available in ESA Forensics.

Example

```
ptyMapReduceProtector mapReduceProtector = new ptyMapReduceProtector();
List<Integer> errorIndex = new ArrayList<Integer>();

int[] protectData     = {1234, 5678, 9012, 3456};

int[] protectedData = mapReduceProtector.bulkProtect( "DE_PROTECT",
            errorIndex, protectData );

//CHECK THE ERROR INDEXES FOR ERRORS
System.out.print("Error Index: ");
for(int i = 0; i < errorIndex.size(); i++)
{
     System.out.print(errorIndex.get( i ));
     if(i < errorIndex.size() - 1)
     {
       System.out.print(",");
     }
}
//ABOVE CODE WILL ONLY PRINT THE ERROR INDEXES
```

Exception

ptyMapRedProtectorException: If an error is encountered during bulk protection of data

Table 3-8: Supported Protection Methods

MapReduce APIs	Supported Protection Methods					
	Tokenization	**Encryption**	**FPE**	**No Encryption**	**Masking**	**Monitoring**
bulkProtect() - Int data	Integer (4 Bytes)	No	No	Yes	No	Yes

Note: The protection methods, that are not mentioned in the *Supported Protection Methods* table, are not supported.

3.1.16 bulkProtect() - Long data

This is used when a set of data needs to be protected in a bulk operation. It helps to improve performance.

public long[] bulkProtect(String dataElement, List <Integer> errorIndex, long[] inputDataItems)

Parameters

dataElement: Name of data element to be protected

errorIndex : *array* used to store all error indices encountered while protecting each data entry in input Data Items

inputDataItems: *array* to store bulk *long* data for protection

Result

Long array of protected data

If the Backward Compatibility mode is not set, then the appropriate error code appears. For more information about the return codes, refer to *Table 10-2 PEP Log Return Codes* and *Table 10-3 PEP Result Codes* in the *Big Data Protector Guide 9.1.0.0*.

If the Backward Compatibility mode is set, then the Error Index includes one of the following values, per entry in the bulk protect operation:

- 1: The protect operation for the entry is successful.
- 0: The protect operation for the entry is unsuccessful.
  - For more information about the failed entry, view the logs available in ESA Forensics.
- Any other value or garbage return value: The protect operation for the entry is unsuccessful. For more information about the failed entry, view the logs available in ESA Forensics.

Example

```
ptyMapReduceProtector mapReduceProtector = new ptyMapReduceProtector();
List<Integer> errorIndex = new ArrayList<Integer>();

long[] protectData     = {123412341234, 567856785678, 901290129012,
                          345634563456};

long[] protectedData = mapReduceProtector.bulkProtect( "DE_PROTECT",
            errorIndex, protectData );

//CHECK THE ERROR INDEXES FOR ERRORS

System.out.print("Error Index: ");
for(int i = 0; i < errorIndex.size(); i++)
{
     System.out.print(errorIndex.get( i ));
     if(i < errorIndex.size() - 1)
     {
       System.out.print(",");
     }
}
//ABOVE CODE WILL ONLY PRINT THE ERROR INDEXES
```

Exception

ptyMapRedProtectorException: If an error is encountered during bulk protection of data

Table 3-9: Supported Protection Methods

MapReduce APIs	Supported Protection Methods					
	Tokenization	**Encryption**	**FPE**	**No Encryption**	**Masking**	**Monitoring**
bulkProtect() - Long data	Integer (8 Bytes)	No	No	Yes	No	Yes

Note: The protection methods, that are not mentioned in the *Supported Protection Methods* table, are not supported.

3.1.17 bulkUnprotect() - Byte array data

This method unprotects in bulk the *inputDataItems* with the required data element.

Note: For Date and Datetime type of data elements, the protect API returns an invalid input data error if the input value falls between the non-existent date range from 05-OCT-1582 to 14-OCT-1582 of the Gregorian Calendar.

For more information about the tokenization and de-tokenization of the cutover dates of the Proleptic Gregorian Calendar, refer to the section *Date* and *Datetime* tokenization in the *Protection Method Reference Guide 9.1.0.0*.

public byte[][] bulkUnprotect(String dataElement, List<Integer> errorIndex, byte[][] inputDataItems)

Parameters

String dataElement : Name of data element to be unprotected

int[] error index: *array* of the error indices encountered while unprotecting each data entry in *inputDataItems*

byte[][] inputDataItems: two-dimensional *array* to help store bulk data to be unprotected

> **Note:** If you are using Format Preserving Encryption (FPE) and Byte APIs, then ensure that the encoding selected in the **Plaintext Encoding** drop-down for the required FPE data element matches the encoding that is used to convert the protected *byte* input data to *string*.

Result

Two-dimensional byte *array* of unprotected data

If the Backward Compatibility mode is not set, then the appropriate error code appears. For more information about the return codes, refer to *Table 10-2 PEP Log Return Codes* and *Table 10-3 PEP Result Codes* in section *10 Appendix: Return Codes* in the *Big Data Protector Guide 9.1.0.0*.

If the Backward Compatibility mode is set, then the Error Index includes one of the following values, per entry in the bulk unprotect operation:

- 1: The unprotect operation for the entry is successful.
- 0: The unprotect operation for the entry is unsuccessful.

  For more information about the failed entry, view the logs available in ESA Forensics.
- Any other value or garbage return value: The unprotect operation for the entry is unsuccessful. For more information about the failed entry, view the logs available in ESA Forensics.

Example

```
ptyMapReduceProtector mapReduceProtector = new ptyMapReduceProtector();
List<Integer> errorIndex = new ArrayList<Integer>();

byte[][] protectData     = {"protegrity".getBytes{}, "protegrity".getBytes(),
                            "protegrity".getBytes(), "protegrity".getBytes()};

byte[][] protectedData = mapReduceProtector.bulkProtect( "DE_PROTECT",
            errorIndex, protectData );

//THIS WILL PRINT THE UNPROTECTED DATA
System.out.print("Protected Data: ");
for(int i = 0; i < protectedData.length; i++)
{
    System.out.print(protectedData[i] == null ? null : new String(protectedData[i]));
    if(i < protectedData.length - 1)
    {
       System.out.print(",");
    }
}

//THIS WILL PRINT THE ERROR INDEX FOR PROTECT OPERATION
System.out.println("");
System.out.print("Error Index: ");
for(int i = 0; i < errorIndex.size(); i++)
{
     System.out.print(errorIndex.get( i ));
     if(i < errorIndex.size() - 1)
     {
       System.out.print(",");
     }
}

byte[][] unprotectedData = mapReduceProtector.bulkUnprotect( "DE_PROTECT",
            errorIndex, protectedData );


//THIS WILL PRINT THE PROTECTED DATA
```

```
System.out.print("UnProtected Data: ");
for(int i = 0; i < unprotectedData.length; i++)
{
    System.out.print(unprotectedData[i] == null ? null : new
String(unprotectedData[i]));
    if(i < unprotectedData.length - 1)
    {
       System.out.print(",");
    }
}

//THIS WILL PRINT THE ERROR INDEX FOR UNPROTECT OPERATION
System.out.println("");
System.out.print("Error Index: ");
for(int i = 0; i < errorIndex.size(); i++)
{
     System.out.print(errorIndex.get( i ));
     if(i < errorIndex.size() - 1)
     {
       System.out.print(",");
     }
}
```

Exception

ptyMapRedProtectorException: For errors when unprotecting data

Table 3-10: Supported Protection Methods

MapReduce APIs	Supported Protection Methods					
	Tokenization	**Encryption**	**FPE**	**No Encryption**	**Masking**	**Monitoring**
bulkUnprotect() - Byte array data	• Numeric (0-9) • Credit Card • Alpha (A-Z) • Upper-case Alpha (A-Z) • Alpha-Numeric (0-9, a-z, A-Z) • Upper Alpha-Numeric (0-9, A-Z) • Lower ASCII • Printable • Date (YYYY-MM-DD, DD/MM/YYYY, MM.DD.YYYY) • Datetime (YYYY-MM-DD HH:MM:SS) • Decimal • Unicode (Gen2) • Unicode (Legacy) • Unicode (Base64 - Encoded Byte's Charset should match Dataelement's Encoding Type) • Binary	• AES-128 • AES-256 • 3DES • CUSP	FPE (All - Encoded Byte's Charset should match Dataelement's Encoding Type)	Yes	Yes	Yes

MapReduce APIs	Supported Protection Methods					
	Tokenization	Encryption	FPE	No Encryption	Masking	Monitoring
	• Email					

Note: The protection methods, that are not mentioned in the *Supported Protection Methods* table, are not supported.

3.1.18 bulkUnprotect() - Int data

This method unprotects in bulk the *inputDataItems* with the required data element.

public int[] bulkUnprotect(String dataElement, List<Integer> errorIndex, int[] inputDataItems)

Parameters

String dataElement : Name of data element to be unprotected

int[] error index: *array* of the error indices encountered while unprotecting each data entry in *inputDataItems*

int[] inputDataItems: *int* array to be unprotected

Result

unprotected *int* array data

If the Backward Compatibility mode is not set, then the appropriate error code appears. For more information about the return codes, refer to *Table 10-2 PEP Log Return Codes* and *Table 10-3 PEP Result Codes* in section *10 Appendix: Return Codes* in the *Big Data Protector Guide 9.1.0.0*.

If the Backward Compatibility mode is set, then the Error Index includes one of the following values, per entry in the bulk unprotect operation:

- 1: The unprotect operation for the entry is successful.
- 0: The unprotect operation for the entry is unsuccessful.

  For more information about the failed entry, view the logs available in ESA Forensics.
- Any other value or garbage return value: The unprotect operation for the entry is unsuccessful. For more information about the failed entry, view the logs available in ESA Forensics.

Example

```
ptyMapReduceProtector mapReduceProtector = new ptyMapReduceProtector();
List<Integer> errorIndex = new ArrayList<Integer>();

int[] protectData     = {1234, 5678,9012,3456 };

int[] protectedData = mapReduceProtector.bulkProtect( "DE_PROTECT",
            errorIndex, protectData );

//THIS WILL PRINT THE ERROR INDEX FOR PROTECT OPERATION
System.out.println("");
System.out.print("Error Index: ");
for(int i = 0; i < errorIndex.size(); i++)
{
     System.out.print(errorIndex.get( i ));
     if(i < errorIndex.size() - 1)
     {
       System.out.print(",");
     }
}

int[] unprotectedData = mapReduceProtector.bulkUnprotect( "DE_PROTECT",
            errorIndex, protectedData );

//THIS WILL PRINT THE ERROR INDEX FOR UNPROTECT OPERATION
System.out.println("");
System.out.print("Error Index: ");
```

```
for(int i = 0; i < errorIndex.size(); i++)
{
     System.out.print(errorIndex.get( i ));
     if(i < errorIndex.size() - 1)
     {
       System.out.print(",");
     }
}
```

Exception

ptyMapRedProtectorException: For errors when unprotecting data

Table 3-11: Supported Protection Methods

MapReduce APIs	Supported Protection Methods					
	Tokenization	**Encryption**	**FPE**	**No Encryption**	**Masking**	**Monitoring**
bulkUnprotect() - Int data	Integer (4 Bytes)	No	No	Yes	No	Yes

Note: The protection methods, that are not mentioned in the *Supported Protection Methods* table, are not supported.

3.1.19 bulkUnprotect() - Long data

This method unprotects in bulk the *inputDataItems* with the required data element.

public long[] bulkUnprotect(String dataElement, List<Integer> errorIndex, long[] inputDataItems)

Parameters

String dataElement: Name of data element to be unprotected

int[] error index: *array* of the error indices encountered while unprotecting each data entry in *inputDataItems*

long[] inputDataItems: long *array* to be unprotected

Result

unprotected *long* array data

If the Backward Compatibility mode is not set, then the appropriate error code appears. For more information about the return codes, refer to *Table 10-2 PEP Log Return Codes* and *Table 10-3 PEP Result Codes* in section *10 Appendix: Return Codes* in the *Big Data Protector Guide 9.1.0.0*.

If the Backward Compatibility mode is set, then the Error Index includes one of the following values, per entry in the bulk unprotect operation:

- 1: The unprotect operation for the entry is successful.
- 0: The unprotect operation for the entry is unsuccessful.

  For more information about the failed entry, view the logs available in ESA Forensics.
- Any other value or garbage return value: The unprotect operation for the entry is unsuccessful. For more information about the failed entry, view the logs available in ESA Forensics.

Example

```
ptyMapReduceProtector mapReduceProtector = new ptyMapReduceProtector();
List<Integer> errorIndex = new ArrayList<Integer>();

long[] protectData     = { 123412341234, 567856785678,
                           901290129012, 345634563456 };

long[] protectedData = mapReduceProtector.bulkProtect( "DE_PROTECT",
            errorIndex, protectData );

//THIS WILL PRINT THE ERROR INDEX FOR PROTECT OPERATION
```

```
System.out.println("");
System.out.print("Error Index: ");
for(int i = 0; i < errorIndex.size(); i++)
{
     System.out.print(errorIndex.get( i ));
     if(i < errorIndex.size() - 1)
     {
       System.out.print(",");
     }
}

long[] unprotectedData = mapReduceProtector.bulkUnprotect( "DE_PROTECT",
            errorIndex, protectedData );

//THIS WILL PRINT THE ERROR INDEX FOR UNPROTECT OPERATION
System.out.println("");
System.out.print("Error Index: ");
for(int i = 0; i < errorIndex.size(); i++)
{
     System.out.print(errorIndex.get( i ));
     if(i < errorIndex.size() - 1)
     {
       System.out.print(",");
     }
}
```

Exception

ptyMapRedProtectorException: For errors when unprotecting data

Table 3-12: Supported Protection Methods

MapReduce APIs	Supported Protection Methods					
	Tokenization	**Encryption**	**FPE**	**No Encryption**	**Masking**	**Monitoring**
bulkUnprotect() - Long data	Integer (8 Bytes)	No	No	Yes	No	Yes

> **Note:** The protection methods, that are not mentioned in the *Supported Protection Methods* table, are not supported.

3.1.20 reprotect() - Byte array data

Data that has been protected earlier is protected again with a separate data element.

public byte[] reprotect(String oldDataElement, String newDataElement, byte[] data)

Parameters

String oldDataElement :Name of data element with which data was protected earlier

String newDataElement: Name of new data element with which data is reprotected

byte[] data : *array* of data to be protected

> **Note:** If you are using Format Preserving Encryption (FPE) and Byte APIs, then ensure that the encoding, which is used to convert the *string* input data to *bytes*, matches the encoding that is selected in the **Plaintext Encoding** drop-down for the required FPE data element.

Result

Byte array of reprotected data

Example

```
ptyMapReduceProtector mapReduceProtector = new ptyMapReduceProtector();
byte[] protectedResult   = mapReduceProtector.protect( "DE_PROTECT_1",
                           "protegrity".getBytes() );
```

```
byte[] reprotectedResult = mapReduceProtector.reprotect( "DE_PROTECT_1",
                           "DE_PROTECT_2", protectedResult );
```

Exception

ptyMapRedProtectorException: For errors while reprotecting data

Table 3-13: Supported Protection Methods

MapReduce APIs	Supported Protection Methods					
	Tokenization	**Encryption**	**FPE**	**No Encryption**	**Masking**	**Monitoring**
reprotect() - Byte array data	• Numeric (0-9) • Credit Card • Alpha (A-Z) • Upper-case Alpha (A-Z) • Alpha-Numeric (0-9, a-z, A-Z) • Upper Alpha-Numeric (0-9, A-Z) • Lower ASCII • Printable • Date (YYYY-MM-DD, DD/MM/YYYY, MM.DD.YYYY) • Datetime (YYYY-MM-DD HH:MM:SS) • Decimal • Unicode (Gen2) • Unicode (Legacy) • Unicode (Base64 - Encoded Byte's Charset should match Dataelement's Encoding Type) • Binary • Email	• AES-128 • AES-256 • 3DES • CUSP	FPE (All - Encoded Byte's Charset should match Dataelement's Encoding Type)	Yes	Yes	Yes

Note: The protection methods, that are not mentioned in the *Supported Protection Methods* table, are not supported.

3.1.21 reprotect() - Int data

Data that has been protected earlier is protected again with a separate data element.

public int reprotect(String oldDataElement, String newDataElement, int data)

Parameters

String oldDataElement :Name of data element with which data was protected earlier

String newDataElement: Name of new data element with which data is reprotected

int data: *array* of data to be protected

Result

Reprotected *int* data

Example

```
ptyMapReduceProtector mapReduceProtector = new ptyMapReduceProtector();
int protectedResult   = mapReduceProtector.protect( "DE_PROTECT_1",
                            1234 );
int reprotectedResult = mapReduceProtector.reprotect( "DE_PROTECT_1",
                            "DE_PROTECT_2", protectedResult );
```

Exception

ptyMapRedProtectorException: For errors while reprotecting data

Table 3-14: Supported Protection Methods

MapReduce APIs	Supported Protection Methods					
	Tokenization	**Encryption**	**FPE**	**No Encryption**	**Masking**	**Monitoring**
reprotect() - Int data	Integer (4 Bytes)	No	No	Yes	No	Yes

Note: The protection methods, that are not mentioned in the *Supported Protection Methods* table, are not supported.

3.1.22 reprotect() - Long data

Data that has been protected earlier is protected again with a separate data element.

public long reprotect(String oldDataElement, String newDataElement, long data)

Parameters

String oldDataElement :Name of data element with which data was protected earlier

String newDataElement: Name of new data element with which data is reprotected

long data: *array* of data to be protected

Result

Reprotected *long* data

Example

```
ptyMapReduceProtector mapReduceProtector = new ptyMapReduceProtector();
long protectedResult   = mapReduceProtector.protect( "DE_PROTECT_1",
                            123412341234 );
int reprotectedResult = mapReduceProtector.reprotect( "DE_PROTECT_1",
                            "DE_PROTECT_2", protectedResult );
```

Exception

ptyMapRedProtectorException: For errors while reprotecting data

Table 3-15: Supported Protection Methods

MapReduce APIs	Supported Protection Methods					
	Tokenization	**Encryption**	**FPE**	**No Encryption**	**Masking**	**Monitoring**
reprotect() - Long data	Integer (8 Bytes)	No	No	Yes	No	Yes

Note: The protection methods, that are not mentioned in the *Supported Protection Methods* table, are not supported.

3.1.23 hmac()

> **Note:** It is recommended to use the HMAC data element with the protect() and bulkProtect() Byte APIs for hashing byte array data, instead of using the hmac() API.

This method performs data hashing using the HMAC operation on a single data item with a data element, which is associated with *hmac*. It returns *hmac* value of the given data with the given data element.

public byte[] hmac(String dataElement, byte[] data)

Parameters

String dataElement:Name of data element for HMAC

byte[] data: *array* of data for HMAC

Result

Byte *array* of HMAC data

Example

```
ptyMapReduceProtector mapReduceProtector = new ptyMapReduceProtector();
long protectedResult   = mapReduceProtector.protect( "DE_PROTECT_1",
                         123412341234 );
int reprotectedResult = mapReduceProtector.reprotect( "DE_PROTECT_1",
                         "DE_PROTECT_2", protectedResult );
```

Exception

ptyMapRedProtectorException: If an error occurs while doing HMAC

Table 3-16: Supported Protection Methods

MapReduce APIs	Supported Protection Methods					
	Tokenization	Encryption	FPE	No Encryption	Masking	Monitoring
hmac()	HMAC	No	No	Yes	No	Yes

> **Note:** The protection methods, that are not mentioned in the *Supported Protection Methods* table, are not supported.

3.2 Hive UDFs

This section describes all Hive User Defined Functions (UDFs) that are available for protection and unprotection in Big Data Protector to build secure Big Data applications.

> **Warning:** If you are using Ranger or Sentry, then ensure that your policy provides *create* access permissions to the required UDFs.

3.2.1 ptyGetVersion()

This UDF returns the current version of PEP.

ptyGetVersion()

Parameters

None

Result

This UDF returns the current version of PEP.

Example

```
create temporary function ptyGetVersion AS 'com.protegrity.hive.udf.ptyGetVersion';
```

```
drop table if exists test_data_table;

create table test_data_table(val string) row format delimited fields terminated by ','
stored as textfile;

load data local inpath 'test_data.csv' OVERWRITE INTO TABLE test_data_table;

select ptyGetVersion() from test_data_table;
```

3.2.2 ptyWhoAmI()

This UDF returns the current logged in user.

ptyWhoAmI()

Parameters

None

Result

This UDF returns the current logged in user.

Example

```
create temporary function ptyWhoAmI AS 'com.protegrity.hive.udf.ptyWhoAmI';
select ptyWhoAmI();
```

3.2.3 ptyProtectStr()

This UDF protects *string* values.

> **Note:** For Date and Datetime type of data elements, the protect API returns an invalid input data error if the input value falls between the non-existent date range from 05-OCT-1582 to 14-OCT-1582 of the Gregorian Calendar.
>
> For more information about the tokenization and de-tokenization of the cutover dates of the Proleptic Gregorian Calendar, refer to the section *Date* and *Datetime* tokenization in the *Protection Method Reference Guide 8.1.0.0*.

ptyProtectStr(String input, String dataElement)

Parameters

String input: *String* value to protect

String dataElement: Name of data element to protect *string* value

Result

This UDF returns protected *string* value.

Example

```
create temporary function ptyProtectStr AS 'com.protegrity.hive.udf.ptyProtectStr';

drop table if exists test_data_table;
drop table if exists temp_table;

create table temp_table(val string) row format delimited fields terminated by ','
stored as textfile;

create table test_data_table(val string) row format delimited fields terminated by ','
stored as textfile;

LOAD DATA LOCAL INPATH 'test_data.csv' OVERWRITE INTO TABLE temp_table;

insert overwrite table test_data_table select trim(val) from temp_table;
```

```
select ptyProtectStr(val, 'Token_alpha') from test_data_table;
```

Table 3-17: Supported Protection Methods

HIVE UDFs	Supported Protection Methods					
	Tokenization	**Encryption**	**FPE**	**No Encryption**	**Masking**	**Monitoring**
ptyProtectStr()	• Numeric (0-9) • Credit Card • Alpha • Upper Case Alpha • Alpha Numeric • Upper Alpha Numeric • Lower ASCII • Datetime (YYYY-MM-DD HH:MM:SS) • Date (YYYY-MM-DD, DD/MM/YYYY, MM.DD.YYYY) • Decimal • Email • Unicode (Legacy) • Unicode (Base64 - Encoded Byte's Charset should match Dataelement's Encoding Type) • Unicode (Gen2)	No	Yes	Yes	Yes	Yes

Note: The protection methods, that are not mentioned in the *Supported Protection Methods* table, are not supported.

3.2.4 ptyUnprotectStr()

This UDF unprotects the existing protected string value.

Note: For Date and Datetime type of data elements, the protect API returns an invalid input data error if the input value falls between the non-existent date range from 05-OCT-1582 to 14-OCT-1582 of the Gregorian Calendar.

For more information about the tokenization and de-tokenization of the cutover dates of the Proleptic Gregorian Calendar, refer to the section *Date* and *Datetime* tokenization in the *Protection Method Reference Guide 8.1.0.0*.

ptyUnprotectStr(String input, String dataElement)

Parameters

`String input :` Protected *string* value to unprotect

`String dataElement:` Name of data element to unprotect *string* value

Result

This UDF returns unprotected *string* value.

Example

```
create temporary function ptyProtectStr AS 'com.protegrity.hive.udf.ptyProtectStr';
create temporary function ptyUnprotectStr AS 'com.protegrity.hive.udf.ptyUnprotectStr';

drop table if exists test_data_table;
drop table if exists temp_table;
drop table if exists protected_data_table;

create table temp_table(val string) row format delimited fields terminated by ','
stored as textfile;

create table test_data_table(val string) row format delimited fields terminated by ','
stored as textfile;

create table protected_data_table(protectedValue string) row format delimited fields
terminated by ',' stored as textfile;

LOAD DATA LOCAL INPATH 'test_data.csv' OVERWRITE INTO TABLE temp_table;

insert overwrite table test_data_table select trim(val) from temp_table;

insert overwrite table protected_data_table select ptyProtectStr(val, 'Token_alpha')
from test_data_table;

select ptyUnprotectStr(protectedValue, 'Token_alpha') from protected_data_table;
```

Table 3-18: Supported Protection Methods

HIVE UDFs	Supported Protection Methods					
	Tokenization	**Encryption**	**FPE**	**No Encryption**	**Masking**	**Monitoring**
ptyUnprotectStr()	• Numeric (0-9) • Credit Card • Alpha • Upper Case Alpha • Alpha Numeric • Upper Alpha Numeric • Lower ASCII • Datetime (YYYY-MM-DD HH:MM:SS) • Date (YYYY-MM-DD, DD/MM/YYYY, MM.DD.YYYY) • Decimal • Email • Unicode (Legacy) • Unicode (Base64 - Encoded Byte's Charset should match Dataelement's Encoding Type)	No	Yes	Yes	Yes	Yes

HIVE UDFs	Supported Protection Methods					
	Tokenization	Encryption	FPE	No Encryption	Masking	Monitoring
	• Unicode (Gen2)					

Note: The protection methods, that are not mentioned in the *Supported Protection Methods* table, are not supported.

3.2.5 ptyReprotect()

This UDF reprotects *string* format protected data, which was earlier protected using the *ptyProtectStr* UDF, with a different data element.

ptyReprotect(String input, String oldDataElement, String newDataElement)

Parameters

`String input:` *String* value to reprotect.

`String oldDataElement:` Name of data element used to protect the data earlier.

`String newDataElement:` Name of new data element to reprotect the data.

Result

This UDF returns protected *string* value.

Example

```
create temporary function ptyProtectStr AS 'com.protegrity.hive.udf.ptyProtectStr';

create temporary function ptyReprotect AS 'com.protegrity.hive.udf.ptyReprotect';

drop table if exists test_data_table;
drop table if exists temp_table;

create table temp_table(val string) row format delimited fields terminated by ','
stored as textfile;

create table test_data_table(val string) row format delimited fields terminated by ','
stored as textfile;

create table test_protected_data_table(val string) row format delimited fields
terminated by ',' stored as textfile;

LOAD DATA LOCAL INPATH 'test_data.csv' OVERWRITE INTO TABLE temp_table;

insert overwrite table test_data_table select trim(val) from temp_table;

insert overwrite table test_protected_data_table select ptyProtectStr(val,
'Token_alpha') from test_data_table;

create table test_reprotected_data_table(val string) row format delimited fields
terminated by ',' stored as textfile;

insert overwrite table test_reprotected_data_table select ptyReprotect(val,
'Token_alpha', 'new_Token_alpha') from test_protected_data_table;
```

Table 3-19: Supported Protection Methods

HIVE UDFs	Supported Protection Methods					
	Tokenization	Encryption	FPE	No Encryption	Masking	Monitoring
ptyReprotect()	• Numeric (0-9) • Credit Card • Alpha • Upper Case Alpha	No	Yes	Yes	Yes	Yes

HIVE UDFs	Supported Protection Methods					
	Tokenization	Encryption	FPE	No Encryption	Masking	Monitoring
	• Alpha Numeric • Upper Alpha Numeric • Lower ASCII • Datetime (YYYY-MM-DD HH:MM:SS) • Date (YYYY-MM-DD, DD/MM/YYYY, MM.DD.YYYY) • Decimal • Email • Unicode (Legacy) • Unicode (Base64 - Encoded Byte's Charset should match Dataelement's Encoding Type) • Unicode (Gen2)					

Note: The protection methods, that are not mentioned in the *Supported Protection Methods* table, are not supported.

3.2.6 ptyProtectUnicode()

This UDF protects *string* (Unicode) values.

ptyProtectUnicode(String input, String dataElement)

Parameters

`String input:` *String* (Unicode) value to protect.

`String dataElement:` Name of data element to protect *string* (Unicode) value

Warning:

This UDF should be used only if you need to tokenize Unicode data in Hive, and migrate the tokenized data from Hive to a Teradata database and detokenize the data using the Protegrity Database Protector.

Ensure that you use this UDF with a Unicode tokenization data element only.

For more information about migrating tokenized Unicode data to a Teradata database, refer to the *Big Data Protector Guide 9.1.0.0*.

Result

This UDF returns protected *string* value.

Example

```
create temporary function ptyProtectUnicode AS
'com.protegrity.hive.udf.ptyProtectUnicode';
```

```
drop table if exists temp_table;

create table temp_table(val string) row format delimited fields terminated by ','
stored as textfile;

LOAD DATA LOCAL INPATH 'test_data.csv' OVERWRITE INTO TABLE temp_table;

select ptyProtectUnicode(val, 'Token_unicode') from temp_table;
```

Table 3-20: Supported Protection Methods

HIVE UDFs	Supported Protection Methods					
	Tokenization	**Encryption**	**FPE**	**No Encryption**	**Masking**	**Monitoring**
ptyProtectUnicode()	• Unicode (Legacy) • Unicode Base64	No	No	Yes	No	Yes

Note: The protection methods, that are not mentioned in the *Supported Protection Methods* table, are not supported.

3.2.7 ptyUnprotectUnicode()

This UDF unprotects the existing protected string value.

ptyUnprotectUnicode(String input, String dataElement)

Parameters

`String input:` Protected *string* value to unprotect.

`String dataElement:` Name of data element to unprotect *string* value.

Warning:

This UDF should be used only if you need to tokenize Unicode data in Teradata using the Protegrity Database Protector, and migrate the tokenized data from a Teradata database to Hive and detokenize the data using the Protegrity Big Data Protector for Hive.

Ensure that you use this UDF with a Unicode tokenization data element only.

For more information about migrating tokenized Unicode data from a Teradata database, refer to *Big Data Protector Guide 9.1.0.0*.

Result

This UDF returns unprotected *string* (Unicode) value.

Example

```
create temporary function ptyProtectUnicode AS
'com.protegrity.hive.udf.ptyProtectUnicode';

create temporary function ptyUnprotectUnicode AS
'com.protegrity.hive.udf.ptyUnprotectUnicode';

drop table if exists temp_table;
drop table if exists protected_data_table;

create table temp_table(val string) row format delimited fields terminated by ','
stored as textfile;

create table protected_data_table(protectedValue string) row format delimited fields
terminated by ',' stored as textfile;

LOAD DATA LOCAL INPATH 'test_data.csv' OVERWRITE INTO TABLE temp_table;
```

```
insert overwrite table protected_data_table select ptyProtectUnicode(val,
'Token_unicode') from temp_table;
```

Table 3-21: Supported Protection Methods

HIVE UDFs	Supported Protection Methods					
	Tokenization	**Encryption**	**FPE**	**No Encryption**	**Masking**	**Monitoring**
ptyUnprotectUnicode()	• Unicode (Legacy) • Unicode Base64	No	No	Yes	No	Yes

Note: The protection methods, that are not mentioned in the *Supported Protection Methods* table, are not supported.

3.2.8 ptyReprotectUnicode()

This UDF reprotects *string* format protected data, which was protected earlier using the *ptyProtectUnicode* UDF, with a different data element.

ptyReprotectUnicode(String input, String oldDataElement, String newDataElement)

Parameters

`String input:` *String*(Unicode) value to reprotect.

`String oldDataElement:` Name of data element used to protect the data earlier.

`String newDataElement:` Name of new data element to reprotect the data.

Warning:

This UDF should be used only if you need to tokenize Unicode data in Hive, and migrate the tokenized data from Hive to a Teradata database and detokenize the data using the Protegrity Database Protector.

Ensure that you use this UDF with a Unicode tokenization data element only.

For more information about migrating tokenized Unicode data to a Teradata database, refer to the *Big Data Protector Guide 9.1.0.0*.

Result

This UDF returns protected *string* value.

Example

```
create temporary function ptyProtectUnicode AS
'com.protegrity.hive.udf.ptyProtectUnicode';

create temporary function ptyReprotectUnicode AS
'com.protegrity.hive.udf.ptyReprotectUnicode';

drop table if exists test_data_table;
drop table if exists temp_table;

create table temp_table(val string) row format delimited fields terminated by ','
stored as textfile;

create table test_data_table(val string) row format delimited fields terminated by ','
stored as textfile;

create table test_protected_data_table(val string) row format delimited fields
terminated by ',' stored as textfile;

LOAD DATA LOCAL INPATH 'test_data.csv' OVERWRITE INTO TABLE temp_table;
```

```
insert overwrite table test_data_table select cast(trim(val)) from temp_table;

insert overwrite table test_protected_data_table select ptyProtectUnicode(val,
'Unicode_Token') from test_data_table;

create table test_reprotected_data_table(val string) row format delimited fields
terminated by ',' stored as textfile;

insert overwrite table test_reprotected_data_table select ptyReprotectUnicode(val,
'Unicode_Token',’new_Unicode_Token’) from test_data_table;
```

Table 3-22: Supported Protection Methods

HIVE UDFs	Supported Protection Methods					
	Tokenization	**Encryption**	**FPE**	**No Encryption**	**Masking**	**Monitoring**
ptyReprotectUnicode()	• Unicode (Legacy) • Unicode Base64	No	No	Yes	No	Yes

Note: The protection methods, that are not mentioned in the *Supported Protection Methods* table, are not supported.

3.2.9 ptyProtectShort()

This UDF protects *SmallInt (Short)* values.

Signature

ptyProtectShort(SmallInt input, String dataElement)

Parameters

`SmallInt input:` Specifies the *SmallInt* value to protect.

`String dataElement:` Name of the data element to protect the *SmallInt* value.

Result

This UDF returns the protected *SmallInt* value.

Example

```
create temporary function ptyProtectShort AS 'com.protegrity.hive.udf.ptyProtectShort';
drop table if exists test_data_table;
drop table if exists temp_table;
create table temp_table(val string) row format delimited fields terminated by ','
stored as textfile;
create table test_data_table(val smallint) row format delimited fields terminated by
',' stored as textfile;
LOAD DATA LOCAL INPATH 'test_data.csv' OVERWRITE INTO TABLE temp_table;
insert overwrite table test_data_table select cast(trim(val) as smallint) from
temp_table;
select ptyProtectShort(val, 'Token_Integer_2') from test_data_table;
```

Table 3-23: Supported Protection Methods

HIVE UDFs	Supported Protection Methods					
	Tokenization	**Encryption**	**FPE**	**No Encryption**	**Masking**	**Monitoring**
ptyProtectShort()	Integer 2 Bytes	No	No	Yes	No	Yes

Note: The protection methods that are not mentioned in the *Supported Protection Methods* table are not supported.

3.2.10 ptyUnprotectShort()

This UDF unprotects the existing protected *SmallInt (Short)* values.

Signature

ptyUnprotectShort(SmallInt input, String dataElement)

Parameters

`SmallInt input:` Specifies the protected *SmallInt* value to unprotect.

`String dataElement:` Name of the data element to unprotect the *SmallInt* value.

Result

This UDF returns the unprotected *SmallInt* value.

Example

```
create temporary function ptyProtectShort AS 'com.protegrity.hive.udf.ptyProtectShort';
create temporary function ptyUnprotectShort AS
'com.protegrity.hive.udf.ptyUnprotectShort';
drop table if exists test_data_table;
drop table if exists temp_table;
drop table if exists protected_data_table;
create table temp_table(val string) row format delimited fields terminated by ','
stored as textfile;
create table test_data_table(val smallint) row format delimited fields terminated by
',' stored as textfile;
create table protected_data_table(protectedValue smallint) row format delimited fields
terminated by ',' stored as textfile;
LOAD DATA LOCAL INPATH 'test_data.csv' OVERWRITE INTO TABLE temp_table;
insert overwrite table test_data_table select cast(trim(val) as smallint) from
temp_table;
insert overwrite table protected_data_table select ptyProtectShort(val,
'Token_Integer_2') from test_data_table;
select ptyUnprotectShort(protectedValue, 'Token_Integer_2') from protected_data_table;
```

Table 3-24: Supported Protection Methods

HIVE UDFs	Supported Protection Methods					
	Tokenization	**Encryption**	**FPE**	**No Encryption**	**Masking**	**Monitoring**
ptyUnprotectShort()	Integer 2 Bytes	No	No	Yes	No	Yes

Note: The protection methods that are not mentioned in the *Supported Protection Methods* table are not supported.

3.2.11 ptyReprotect()

This UDF reprotects the protected *SmallInt (Short)* data with a different data element.

Signature

ptyReprotect(SmallInt input, String oldDataElement, String newDataElement)

Parameters

`SmallInt input:` Specifies the *SmallInt* value to reprotect.

`String oldDataElement:` Name of the data element used to protect the data earlier.

`String newDataElement:` Name of the new data element used to reprotect the data.

Result

This UDF returns the reprotected *SmallInt* value.

Example

```
create temporary function ptyProtectShort AS 'com.protegrity.hive.udf.ptyProtectShort';
create temporary function ptyReprotect AS 'com.protegrity.hive.udf.ptyReprotect';
drop table if exists test_data_table;
drop table if exists temp_table;
create table temp_table(val string) row format delimited fields terminated by ','
stored as textfile;
create table test_data_table(val smallint) row format delimited fields terminated by
',' stored as textfile;
```

```
create table test_protected_data_table(val smallint) row format delimited fields
terminated by ',' stored as textfile;
LOAD DATA LOCAL INPATH 'test_data.csv' OVERWRITE INTO TABLE temp_table;
insert overwrite table test_data_table select cast(trim(val) as smallint) from
temp_table;
insert overwrite table test_protected_data_table select ptyProtectShort(val,
' Token_Integer_2') from test_data_table;
create table test_reprotected_data_table(val smallint) row format delimited fields
terminated by ',' stored as textfile;
insert overwrite table test_reprotected_data_table select ptyReprotect(val,
'Token_Integer_2', 'new_Token_Integer_2') from test_protected_data_table;
```

Table 3-25: Supported Protection Methods

HIVE UDFs	Supported Protection Methods					
	Tokenization	**Encryption**	**FPE**	**No Encryption**	**Masking**	**Monitoring**
ptyReprotect()	Integer 2 Bytes	No	No	Yes	No	Yes

Note: The protection methods that are not mentioned in the *Supported Protection Methods* table are not supported.

3.2.12 ptyProtectInt()

This UDF protects *integer* values.

ptyProtectInt(int input, String dataElement)

Parameters

`int input:` *Integer* value to protect.

`String dataElement:` Name of data element to protect *integer* value.

Result

This UDF returns protected *integer* value.

Example

```
create temporary function ptyProtectInt AS 'com.protegrity.hive.udf.ptyProtectInt';

drop table if exists test_data_table;
drop table if exists temp_table;

create table temp_table(val string) row format delimited fields terminated by ','
stored as textfile;

create table test_data_table(val int) row format delimited fields terminated by ','
stored as textfile;

LOAD DATA LOCAL INPATH 'test_data.csv' OVERWRITE INTO TABLE temp_table;

insert overwrite table test_data_table select cast(trim(val) as int) from temp_table;

select ptyProtectInt(val, 'Token_numeric') from test_data_table;
```

Table 3-26: Supported Protection Methods

HIVE UDFs	Supported Protection Methods					
	Tokenization	**Encryption**	**FPE**	**No Encryption**	**Masking**	**Monitoring**
ptyProtectInt()	Integer 4 Bytes	No	No	Yes	No	Yes

Note: The protection methods, that are not mentioned in the *Supported Protection Methods* table, are not supported.

3.2.13 ptyUnprotectInt()

This UDF unprotects the existing protected integer value.

ptyUnprotectInt(int input, String dataElement)

Parameters

`int input:` Protected *integer* value to unprotect.

`String dataElement:` Name of data element to unprotect *integer* value.

Result

This UDF returns unprotected *integer* value.

Example

```
create temporary function ptyProtectInt AS 'com.protegrity.hive.udf.ptyProtectInt';

create temporary function ptyUnprotectInt AS 'com.protegrity.hive.udf.ptyUnprotectInt';

drop table if exists test_data_table;
drop table if exists temp_table;
drop table if exists protected_data_table;

create table temp_table(val string) row format delimited fields terminated by ','
stored as textfile;

create table test_data_table(val int) row format delimited fields terminated by ','
stored as textfile;

create table protected_data_table(protectedValue int) row format delimited fields
terminated by ',' stored as textfile;

LOAD DATA LOCAL INPATH 'test_data.csv' OVERWRITE INTO TABLE temp_table;

insert overwrite table test_data_table select cast(trim(val) as int) from temp_table;

insert overwrite table protected_data_table select ptyProtectInt(val, 'Token_numeric')
from test_data_table;

select ptyUnprotectInt(protectedValue, 'Token_numeric') from protected_data_table;
```

Table 3-27: Supported Protection Methods

HIVE UDFs	Supported Protection Methods					
	Tokenization	Encryption	FPE	No Encryption	Masking	Monitoring
ptyUnprotectInt()	Integer 4 Bytes	No	No	Yes	No	Yes

Note: The protection methods, that are not mentioned in the *Supported Protection Methods* table, are not supported.

3.2.14 ptyReprotect()

This UDF reprotects *integer* format protected data with a different data element.

ptyReprotect(int input, String oldDataElement, String newDataElement)

Parameters

`int input:` *Integer* value to reprotect.

`String oldDataElement:` Name of data element used to protect the data earlier.

`String newDataElement:` Name of new data element to reprotect the data.

Result

This UDF returns protected *integer* value.

Example

```
create temporary function ptyProtectInt AS 'com.protegrity.hive.udf.ptyProtectInt';

create temporary function ptyReprotect AS 'com.protegrity.hive.udf.ptyReprotect';

drop table if exists test_data_table;
drop table if exists temp_table;

create table temp_table(val int) row format delimited fields terminated by ',' stored
as textfile;

create table test_data_table(val int) row format delimited fields terminated by ','
stored as textfile;

create table test_protected_data_table(val int) row format delimited fields terminated
by ',' stored as textfile;

LOAD DATA LOCAL INPATH 'test_data.csv' OVERWRITE INTO TABLE temp_table;

insert overwrite table test_data_table select cast(trim(val) as int) from temp_table;

insert overwrite table test_protected_data_table select ptyProtectInt(val,
'Token_Integer') from test_data_table;

create table test_reprotected_data_table(val int) row format delimited fields
terminated by ',' stored as textfile;

insert overwrite table test_reprotected_data_table select ptyReprotect(val,
'Token_Integer', 'new_Token_Integer') from test_protected_data_table;
```

Table 3-28: Supported Protection Methods

HIVE UDFs	Supported Protection Methods					
	Tokenization	**Encryption**	**FPE**	**No Encryption**	**Masking**	**Monitoring**
ptyReprotect()	Integer 4 Bytes	No	No	Yes	No	Yes

> **Note:** The protection methods, that are not mentioned in the *Supported Protection Methods* table, are not supported.

3.2.15 ptyProtectBigInt()

This UDF protects *BigInt* value.

ptyProtectBigInt(BigInt input, String dataElement)

Parameters

`BigInt input:` Value to protect

`String dataElement:` Name of data element to protect value

Result

This UDF returns protected *BigInteger* value.

Example

```
create temporary function ptyProtectBigInt as
'com.protegrity.hive.udf.ptyProtectBigInt';

drop table if exists test_data_table;
drop table if exists temp_table;

create table temp_table(val bigint) row format delimited fields terminated by ','
stored as textfile;

create table test_data_table(val bigint) row format delimited fields terminated by ','
stored as textfile;
```

```
load data local inpath 'test_data.csv' overwrite into table temp_table;

insert overwrite table test_data_table select cast(trim(val) as bigint) from temp_table;

select ptyProtectBigInt(val, 'BIGINT_DE') from test_data_table;
```

Table 3-29: Supported Protection Methods

HIVE UDFs	Supported Protection Methods					
	Tokenization	**Encryption**	**FPE**	**No Encryption**	**Masking**	**Monitoring**
ptyProtectBigInt()	Integer 8 Bytes	No	No	Yes	No	Yes

Note: The protection methods, that are not mentioned in the *Supported Protection Methods* table, are not supported.

3.2.16 ptyUnprotectBigInt()

This UDF unprotects protected *BigInt* value.

ptyUnprotectBigInt(BigInt input, String dataElement)

Parameters

`BigInt input:` Protected value to unprotect

`String dataElement:` Name of data element to unprotect value

Result

This UDF returns unprotected *BigInteger* value.

Example

```
create temporary function ptyProtectBigInt as
'com.protegrity.hive.udf.ptyProtectBigInt';

create temporary function ptyUnprotectBigInt as
'com.protegrity.hive.udf.ptyUnprotectBigInt';

drop table if exists test_data_table;
drop table if exists temp_table;
drop table if exists protected_data_table;

create table temp_table(val bigint) row format delimited fields terminated by ','
stored as textfile;

create table test_data_table(val bigint) row format delimited fields terminated by ','
stored as textfile;

create table protected_data_table(protectedValue bigint) row format delimited fields
terminated by ',' stored as textfile;

load data local inpath 'test_data.csv' overwrite into table temp_table;

insert overwrite table test_data_table select cast(trim(val) as bigint) from temp_table;

insert overwrite table protected_data_table select ptyProtectBigInt(val, 'BIGINT_DE')
from test_data_table;

select ptyUnprotectBigInt(protectedValue, 'BIGINT_DE') from protected_data_table;
```

Table 3-30: Supported Protection Methods

HIVE UDFs	Supported Protection Methods					
	Tokenization	Encryption	FPE	No Encryption	Masking	Monitoring
ptyUnprotectBigInt()	Integer 8 Bytes	No	No	Yes	No	Yes

Note: The protection methods, that are not mentioned in the *Supported Protection Methods* table, are not supported.

3.2.17 ptyReprotect()

This UDF reprotects *BigInt* format protected data with a different data element.

ptyReprotect(Bigint input, String oldDataElement, String newDataElement)

Parameters

`Bigint input:` *Bigint* value to reprotect

`String oldDataElement:` Name of data element used to protect the data earlier

`String newDataElement:` Name of new data element to reprotect the data

Result

This UDF returns protected *bigint* value.

Example

```
create temporary function ptyProtectBigInt AS
'com.protegrity.hive.udf.ptyProtectBigInt';

create temporary function ptyReprotect AS 'com.protegrity.hive.udf.ptyReprotect';

drop table if exists test_data_table;
drop table if exists temp_table;

create table temp_table(val bigint) row format delimited fields terminated by ','
stored as textfile;

create table test_data_table(val bigint) row format delimited fields terminated by ','
stored as textfile;

create table test_protected_data_table(val bigint) row format delimited fields
terminated by ',' stored as textfile;

LOAD DATA LOCAL INPATH 'test_data.csv' OVERWRITE INTO TABLE temp_table;

insert overwrite table test_data_table select cast(trim(val) as bigint) from temp_table;

insert overwrite table test_protected_data_table select ptyProtectBigInt(val,
'Token_BigInteger') from test_data_table;

create table test_reprotected_data_table(val bigint) row format delimited fields
terminated by ',' stored as textfile;

insert overwrite table test_reprotected_data_table select ptyReprotect(val,
' 'BIGINT_DE', 'new_BIGINT_DE') from test_protected_data_table;
```

Table 3-31: Supported Protection Methods

HIVE UDFs	Supported Protection Methods					
	Tokenization	Encryption	FPE	No Encryption	Masking	Monitoring
ptyReprotect()	Integer 8 Bytes	No	No	Yes	No	Yes

Note: The protection methods, that are not mentioned in the *Supported Protection Methods* table, are not supported.

3.2.18 ptyProtectFloat()

This UDF protects *float* value.

ptyProtectFloat(Float input, String dataElement)

Parameters

`Float input:` *Float* value to protect

`String dataElement:` Name of data element to unprotect *float* value

> **Warning:** Ensure that you use the data element with the *No Encryption* method only. Using any other data element might cause corruption of data.

Result

This UDF returns protected *float* value.

Example

```
create temporary function ptyProtectFloat as 'com.protegrity.hive.udf.ptyProtectFloat';

drop table if exists test_data_table;
drop table if exists temp_table;

create table temp_table(val string) row format delimited fields terminated by ','
stored as textfile;

create table test_data_table(val float) row format delimited fields terminated by ','
stored as textfile;

load data local inpath 'test_data.csv' overwrite into table temp_table;

insert overwrite table test_data_table select cast(trim(val) as float) from temp_table;

select ptyProtectFloat(val, 'FLOAT_DE') from test_data_table;
```

Table 3-32: Supported Protection Methods

HIVE UDFs	Supported Protection Methods					
	Tokenization	Encryption	FPE	No Encryption	Masking	Monitoring
ptyProtectFloat()	No	No	No	Yes	No	Yes

> **Note:** The protection methods, that are not mentioned in the *Supported Protection Methods* table, are not supported.

3.2.19 ptyUnprotectFloat()

This UDF unprotects protected *float* value.

ptyUnprotectFloat(Float input, String dataElement)

Parameters

`Float input:` Protected *float* value to unprotect

`String dataElement:` Name of data element to unprotect *float* value.

> **Warning:** Ensure that you use the data element with the *No Encryption* method only. Using any other data element might cause corruption of data.

Result

This UDF returns unprotected *float* value.

Example

```
create temporary function ptyProtectFloat as 'com.protegrity.hive.udf.ptyProtectFloat';

create temporary function ptyUnprotectFloat as
'com.protegrity.hive.udf.ptyUnprotectFloat';

drop table if exists test_data_table;
drop table if exists temp_table;
drop table if exists protected_data_table;

create table temp_table(val string) row format delimited fields terminated by ','
stored as textfile;

create table test_data_table(val float) row format delimited fields terminated by ','
stored as textfile;

create table protected_data_table(protectedValue float) row format delimited fields
terminated by ',' stored as textfile;

load data local inpath 'test_data.csv' overwrite into table temp_table;

insert overwrite table test_data_table select cast(trim(val) as float) from temp_table;

insert overwrite table protected_data_table select ptyProtectFloat(val, 'FLOAT_DE')
from test_data_table;

select ptyUnprotectFloat(protectedValue, 'FLOAT_DE') from protected_data_table;
```

Table 3-33: Supported Protection Methods

HIVE UDFs	Supported Protection Methods					
	Tokenization	**Encryption**	**FPE**	**No Encryption**	**Masking**	**Monitoring**
ptyUnprotectFloat()	No	No	No	Yes	No	Yes

> **Note:** The protection methods, that are not mentioned in the *Supported Protection Methods* table, are not supported.

3.2.20 ptyReprotect()

This UDF reprotects *float* format protected data with a different data element.

ptyReprotect(Float input, String oldDataElement, String newDataElement)

Parameters

`Float input:` *Float* value to reprotect

`String oldDataElement:` Name of data element used to protect the data earlier

`String newDataElement:` Name of new data element to reprotect the data

> **Warning:** Ensure that you use the data element with the *No Encryption* method only. Using any other data element might cause corruption of data.

Result

This UDF returns protected *float* value.

Example

```
create temporary function ptyProtectFloat AS 'com.protegrity.hive.udf.ptyProtectFloat';

create temporary function ptyReprotect AS 'com.protegrity.hive.udf.ptyReprotect';

drop table if exists test_data_table;
drop table if exists temp_table;
```

```
create table temp_table(val float) row format delimited fields terminated by ',' stored
as textfile;

create table test_data_table(val float) row format delimited fields terminated by ','
stored as textfile;

create table test_protected_data_table(val float) row format delimited fields
terminated by ',' stored as textfile;

LOAD DATA LOCAL INPATH 'test_data.csv' OVERWRITE INTO TABLE temp_table;

insert overwrite table test_data_table select cast(trim(val) as float) from temp_table;

insert overwrite table test_protected_data_table select ptyProtectFloat(val,
'NoEncryption') from test_data_table;

create table test_reprotected_data_table(val float) row format delimited fields
terminated by ',' stored as textfile;

insert overwrite table test_reprotected_data_table select ptyReprotect(val, '
NoEncryption','NoEncryption') from test_protected_data_table;
```

Table 3-34: Supported Protection Methods

HIVE UDFs	Supported Protection Methods					
	Tokenization	Encryption	FPE	No Encryption	Masking	Monitoring
ptyReprotect()	No	No	No	Yes	No	Yes

Note: The protection methods, that are not mentioned in the *Supported Protection Methods* table, are not supported.

3.2.21 ptyProtectDouble()

This UDF protects *double* value.

ptyProtectDouble(Double input, String dataElement)

Parameters

`Double input:` *Double* value to unprotect

`String dataElement:` Name of data element to unprotect *double* value

Warning: Ensure that you use the data element with the *No Encryption* method only. Using any other data element might cause corruption of data.

Result

This UDF returns protected *double* value.

Example

```
create temporary function ptyProtectDouble as
'com.protegrity.hive.udf.ptyProtectDouble';

drop table if exists test_data_table;
drop table if exists temp_table;

create table temp_table(val string) row format delimited fields terminated by ','
stored as textfile;

create table test_data_table(val double) row format delimited fields terminated by ','
stored as textfile;

load data local inpath 'test_data.csv' overwrite into table temp_table;

insert overwrite table test_data_table select cast(trim(val) as double) from temp_table;
```

```
select ptyProtectDouble(val, 'DOUBLE_DE') from test_data_table;
```

Table 3-35: Supported Protection Methods

HIVE UDFs	Supported Protection Methods					
	Tokenization	Encryption	FPE	No Encryption	Masking	Monitoring
ptyProtectDouble()	No	No	No	Yes	No	Yes

Note: The protection methods, that are not mentioned in the *Supported Protection Methods* table, are not supported.

3.2.22 ptyUnprotectDouble()

This UDF unprotects protected *double* value.

ptyUnprotectDouble(Double input, String dataElement)

Parameters

`Double input:` *Double* value to unprotect

`String dataElement:` Name of data element to unprotect *double* value

Warning: Ensure that you use the data element with the *No Encryption* method only. Using any other data element might cause corruption of data.

Result

This UDF returns unprotected *double* value.

Example

```
create temporary function ptyProtectDouble as
'com.protegrity.hive.udf.ptyProtectDouble';

create temporary function ptyUnprotectDouble as 'com.prot
egrity.hive.udf.ptyUnprotectDouble';

drop table if exists test_data_table;
drop table if exists temp_table;
drop table if exists protected_data_table;

create table temp_table(val double) row format delimited fields terminated by ','
stored as textfile;

create table test_data_table(val double) row format delimited fields terminated by ','
stored as textfile;

create table protected_data_table(protectedValue double) row format delimited fields
terminated by ',' stored as textfile;

load data local inpath 'test_data.csv' overwrite into table temp_table;

insert overwrite table test_data_table select cast(trim(val) as double) from temp_table;

insert overwrite table protected_data_table select ptyProtectDouble(val, 'DOUBLE_DE')
from test_data_table;

select ptyUnprotectDouble(protectedValue, 'DOUBLE_DE') from protected_data_table;
```

Table 3-36: Supported Protection Methods

HIVE UDFs	Supported Protection Methods					
	Tokenization	**Encryption**	**FPE**	**No Encryption**	**Masking**	**Monitoring**
ptyUnprotectDouble( )	No	No	No	Yes	No	Yes

Note: The protection methods, that are not mentioned in the *Supported Protection Methods* table, are not supported.

3.2.23 ptyReprotect()

This UDF reprotects *double* format protected data with a different data element.

ptyReprotect(Double input, String oldDataElement, String newDataElement)

Parameters

`Double input:` *Double* value to reprotect

`String oldDataElement:` Name of data element used to protect the data earlier

`String newDataElement:` Name of new data element to reprotect the data

Warning: Ensure that you use the data element with the *No Encryption* method only. Using any other data element might cause corruption of data.

Result

This UDF returns protected *double* value.

Example

```
create temporary function ptyProtectDouble AS
'com.protegrity.hive.udf.ptyProtectDouble';

create temporary function ptyReprotect AS 'com.protegrity.hive.udf.ptyReprotect';

drop table if exists test_data_table;
drop table if exists temp_table;

create table temp_table(val double) row format delimited fields terminated by ','
stored as textfile;

create table test_data_table(val double) row format delimited fields terminated by ','
stored as textfile;

create table test_protected_data_table(val double) row format delimited fields
terminated by ',' stored as textfile;

LOAD DATA LOCAL INPATH 'test_data.csv' OVERWRITE INTO TABLE temp_table;

insert overwrite table test_data_table select cast(trim(val) as double) from temp_table;

insert overwrite table test_protected_data_table select ptyProtectDouble(val,
'NoEncryption') from test_data_table;

create table test_reprotected_data_table(val double) row format delimited fields
terminated by ',' stored as textfile;

insert overwrite table test_reprotected_data_table select ptyReprotect(val, '
NoEncryption’,’NoEncryption’) from test_protected_data_table;
```

Table 3-37: Supported Protection Methods

HIVE UDFs	Supported Protection Methods					
	Tokenization	**Encryption**	**FPE**	**No Encryption**	**Masking**	**Monitoring**
ptyProtectDouble()	No	No	No	Yes	No	Yes

Note: The protection methods, that are not mentioned in the *Supported Protection Methods* table, are not supported.

3.2.24 ptyProtectDec()

This UDF protects *decimal* value.

Note: This API works only with the CDH 4.3 distribution.

ptyProtectDec(Decimal input, String dataElement)

Parameters

`Decimal input:` *Decimal* value to protect

`String dataElement:` Name of data element to protect *decimal* value

Warning: Ensure that you use the data element with the *No Encryption* method only. Using any other data element might cause corruption of data.

Result

This UDF returns protected *decimal* value.

Example

```
create temporary function ptyProtectDec as 'com.protegrity.hive.udf.ptyProtectDec';

drop table if exists test_data_table;
drop table if exists temp_table;

create table temp_table(val decimal) row format delimited fields terminated by ','
stored as textfile;

create table test_data_table(val decimal) row format delimited fields terminated by ','
stored as textfile;

load data local inpath 'test_data.csv' overwrite into table temp_table;

insert overwrite table test_data_table select cast(trim(val) as decimal) from
temp_table;

select ptyProtectDec(val, 'BIGDECIMAL_DE') from test_data_table;
```

Table 3-38: Supported Protection Methods

HIVE UDFs	Supported Protection Methods					
	Tokenization	**Encryption**	**FPE**	**No Encryption**	**Masking**	**Monitoring**
ptyProtectDec()	No	No	No	Yes	No	Yes

Note: The protection methods, that are not mentioned in the *Supported Protection Methods* table, are not supported.

3.2.25 ptyUnprotectDec()

This UDF unprotects protected *decimal* value.

> **Note:** This API works only with the CDH 4.3 distribution.

ptyUnprotectDec(Decimal input, String dataElement)

Parameters

`Decimal input:` Protected *Decimal* value to unProtect

`String dataElement:` Name of data element to Unprotect *decimal* value

> **Warning:** Ensure that you use the data element with the *No Encryption* method only. Using any other data element might cause corruption of data.

Result

This UDF returns unprotected *decimal* value.

Example

```
create temporary function ptyProtectDec as 'com.protegrity.hive.udf.ptyProtectDec';

create temporary function ptyUnprotectDec as 'com.protegrity.hive.udf.ptyUnprotectDec';

drop table if exists test_data_table;
drop table if exists temp_table;
drop table if exists protected_data_table;

create table temp_table(val string) row format delimited fields terminated by ','
stored as textfile;

create table test_data_table(val decimal) row format delimited fields terminated by ','
stored as textfile;

create table protected_data_table(protectedValue decimal) row format delimited fields
terminated by ',' stored as textfile;

load data local inpath 'test_data.csv' overwrite into table temp_table;

insert overwrite table test_data_table select cast(trim(val) as decimal) from
temp_table;

insert overwrite table protected_data_table select ptyProtectDec(val, 'BIGDECIMAL_DE')
from test_data_table;

select ptyUnprotectDec(protectedValue, 'BIGDECIMAL_DE') from protected_data_table;
```

Table 3-39: Supported Protection Methods

HIVE UDFs	Supported Protection Methods					
	Tokenization	Encryption	FPE	No Encryption	Masking	Monitoring
ptyUnprotectDec()	No	No	No	Yes	No	Yes

> **Note:** The protection methods, that are not mentioned in the *Supported Protection Methods* table, are not supported.

3.2.26 ptyProtectHiveDecimal()

This UDF protects *decimal* value.

> **Note:** This API works only for distributions which include Hive, Version 0.11 and later.

ptyProtectHiveDecimal(Decimal input, String dataElement)

Parameters

`Decimal input:` *Decimal* value to protect

`String dataElement:` Name of data element to protect *decimal* value

> **Warning:** Ensure that you use the data element with the *No Encryption* method only. Using any other data element might cause corruption of data.

> **Caution:** Before the *ptyProtectHiveDecimal()* UDF is called, Hive rounds off the decimal value in the table to 18 digits in scale, irrespective of the length of the data.

Result

This UDF returns protected *decimal* value.

Example

```
create temporary function ptyProtectHiveDecimal as
'com.protegrity.hive.udf.ptyProtectHiveDecimal';

drop table if exists test_data_table;
drop table if exists temp_table;

create table temp_table(val string) row format delimited fields terminated by ','
stored as textfile;

create table test_data_table(val decimal) row format delimited fields terminated by ','
stored as textfile;

load data local inpath 'test_data.csv' overwrite into table temp_table;

insert overwrite table test_data_table select cast(trim(val) as decimal) from
temp_table;

select ptyProtectHiveDecimal(val, 'BIGDECIMAL_DE') from test_data_table;
```

Table 3-40: Supported Protection Methods

HIVE UDFs	Supported Protection Methods					
	Tokenization	**Encryption**	**FPE**	**No Encryption**	**Masking**	**Monitoring**
ptyProtectHiveDecimal()	No	No	No	Yes	No	Yes

> **Note:** The protection methods, that are not mentioned in the *Supported Protection Methods* table, are not supported.

3.2.27 ptyUnprotectHiveDecimal()

This UDF unprotects Decimal value.

> **Note:** This API works only for distributions which include Hive, Version 0.11 and later.

ptyUnprotectHiveDecimal(Decimal input, String dataElement)

Parameters

`Decimal input:` *Decimal* value to protect

`String dataElement:` Name of data element to unprotect *decimal* value

> **Warning:** Ensure that you use the data element with the *No Encryption* method only. Using any other data element might cause corruption of data.

Result

This UDF returns unprotected *decimal* value.

Example

```
create temporary function ptyProtectHiveDecimal as
'com.protegrity.hive.udf.ptyProtectHiveDecimal';

create temporary function ptyUnprotectHiveDecimal as
'com.protegrity.hive.udf.ptyUnprotectHiveDecimal';

drop table if exists test_data_table;
drop table if exists temp_table;
drop table if exists protected_data_table;

create table temp_table(val string) row format delimited fields terminated by ','
stored as textfile;

create table test_data_table(val decimal) row format delimited fields terminated by ','
stored as textfile;

create table protected_data_table(protectedValue decimal) row format delimited fields
terminated by ',' stored as textfile;

load data local inpath 'test_data.csv' overwrite into table temp_table;

insert overwrite table test_data_table select cast(trim(val) as decimal) from
temp_table;

insert overwrite table protected_data_table select ptyProtectHiveDecimal(val,
'BIGDECIMAL_DE') from test_data_table;

select ptyUnprotectHiveDecimal(protectedValue, 'BIGDECIMAL_DE') from
protected_data_table;
```

Table 3-41: Supported Protection Methods

HIVE UDFs	Supported Protection Methods					
	Tokenization	**Encryption**	**FPE**	**No Encryption**	**Masking**	**Monitoring**
ptyUnprotectHiveDecimal()	No	No	No	Yes	No	Yes

> **Note:** The protection methods, that are not mentioned in the *Supported Protection Methods* table, are not supported.

3.2.28 ptyReprotect()

This UDF reprotects *decimal* format protected data with a different data element.

> **Note:** This API works only for distributions which include Hive, Version 0.11 and later.

ptyReprotect(Decimal input, String oldDataElement, String newDataElement)

Parameters

`Decimal input:` *Decimal* value to reprotect

`String oldDataElement:` Name of data element used to protect the data earlier

`String newDataElement:` Name of new data element to reprotect the data

> **Warning:** Ensure that you use the data element with the *No Encryption* method only. Using any other data element might cause corruption of data.

Result

This UDF returns unprotected *decimal* value.

Example

```
create temporary function ptyProtectHiveDecimal AS
'com.protegrity.hive.udf.ptyProtectHiveDecimal';

create temporary function ptyReprotect AS 'com.protegrity.hive.udf.ptyReprotect';

drop table if exists test_data_table;
drop table if exists temp_table;

create table temp_table(val decimal) row format delimited fields terminated by ','
stored as textfile;

create table test_data_table(val decimal) row format delimited fields terminated by ','
stored as textfile;

create table test_protected_data_table(val decimal) row format delimited fields
terminated by ',' stored as textfile;

LOAD DATA LOCAL INPATH 'test_data.csv' OVERWRITE INTO TABLE temp_table;

insert overwrite table test_data_table select cast(trim(val) as decimal) from
temp_table;

insert overwrite table test_protected_data_table select ptyProtectHiveDecimal(val,
'NoEncryption') from test_data_table;

create table test_reprotected_data_table(val decimal) row format delimited fields
terminated by ',' stored as textfile;

insert overwrite table test_reprotected_data_table select ptyReprotect(val, '
NoEncryption’,’NoEncyption’) from test_protected_data_table;
```

Table 3-42: Supported Protection Methods

HIVE UDFs	Supported Protection Methods					
	Tokenization	Encryption	FPE	No Encryption	Masking	Monitoring
ptyReprotect()	No	No	No	Yes	No	Yes

> **Note:** The protection methods, that are not mentioned in the *Supported Protection Methods* table, are not supported.

3.2.29 ptyProtectDate()

This UDF protects *date* format data, which is provided as input.

ptyProtectDate(Date input, String dataElement)

> **Note:** In the Big Data Protector, version 7.1 release, the *date* format supported is *YYYY-MM-DD* only.

Parameters

`Date input:` The *date* format data, which needs to be protected

`String dataElement:` The data element that will be used to protect *date* format data

Result

This UDF returns *date* format data, which is protected.

Example

```
create temporary function ptyProtectDate AS 'com.protegrity.hive.udf.ptyProtectDate';

drop table if exists test_data_table;
drop table if exists temp_table;

create table temp_table(val date) row format delimited fields terminated by ',' stored
as textfile;

create table test_data_table(val date) row format delimited fields terminated by ','
stored as textfile;

LOAD DATA LOCAL INPATH 'test_data.csv' OVERWRITE INTO TABLE temp_table;

insert overwrite table test_data_table select cast(trim(val) as date) from temp_table;

select ptyProtectDate(val, 'Token_Date') from test_data_table;
```

Table 3-43: Supported Protection Methods

HIVE UDFs	Supported Protection Methods					
	Tokenization	Encryption	FPE	No Encryption	Masking	Monitoring
ptyProtectDate()	Date	No	No	Yes	No	Yes

Note: The protection methods, that are not mentioned in the *Supported Protection Methods* table, are not supported.

3.2.30 ptyUnprotectDate()

This UDF unprotects the protected date format data, which is provided as input.

ptyUnprotectDate(Date input, String dataElement)

Note: In the Big Data Protector, version 7.1 release, the *date* format supported is *YYYY-MM-DD* only.

Parameters

`Date input:` The protected *date* format data, which is provided as input

`String dataElement:` The data element that will be used to unprotect *date* format data

Result

This UDF returns *date* format data, which is unprotected.

Example

```
create temporary function ptyProtectDate AS 'com.protegrity.hive.udf.ptyProtectDate';

create temporary function ptyUnprotectDate AS
'com.protegrity.hive.udf.ptyUnprotectDate';

drop table if exists test_data_table;
drop table if exists temp_table;
drop table if exists protected_data_table;

create table temp_table(val date) row format delimited fields terminated by ',' stored
as textfile;

create table test_data_table(val date) row format delimited fields terminated by ','
stored as textfile;
```

```
create table protected_data_table(protectedValue date) row format delimited fields
terminated by ',' stored as textfile;

LOAD DATA LOCAL INPATH 'test_data.csv' OVERWRITE INTO TABLE temp_table;

insert overwrite table test_data_table select cast(trim(val) as date) from temp_table;

insert overwrite table protected_data_table select ptyProtectDate(val, ‘Token_Date’)
from test_data_table;

select ptyUnprotectDate(protectedValue, ‘Token_Date’) from protected_data_table;
```

Table 3-44: Supported Protection Methods

HIVE UDFs	Supported Protection Methods					
	Tokenization	Encryption	FPE	No Encryption	Masking	Monitoring
ptyUnprotectDate()	Date	No	No	Yes	No	Yes

Note: The protection methods, that are not mentioned in the *Supported Protection Methods* table, are not supported.

3.2.31 ptyReprotect()

This UDF reprotects *date* format protected data, which was earlier protected using the *ptyProtectDate* UDF, with a different data element.

ptyReprotect(Date input, String oldDataElement, String newDataElement)

Note: In the Big Data Protector, version 7.1 release, the *date* format supported is *YYYY-MM-DD* only.

Parameters

`Date input:` The date format data, which needs to be reprotected

`String oldDataElement:` The data element that was used to protect the data earlier

`String newDataElement:` The new data that will be used to reprotect the data

Result

This UDF returns *date* format data, which is protected.

Example

```
create temporary function ptyProtectDate AS 'com.protegrity.hive.udf.ptyProtectDate';

create temporary function ptyReprotect AS 'com.protegrity.hive.udf.ptyReprotect';

drop table if exists test_data_table;
drop table if exists temp_table;

create table temp_table(val date) row format delimited fields terminated by ',' stored
as textfile;

create table test_data_table(val date) row format delimited fields terminated by ','
stored as textfile;

create table test_protected_data_table(val date) row format delimited fields terminated
by ',' stored as textfile;

LOAD DATA LOCAL INPATH 'test_data.csv' OVERWRITE INTO TABLE temp_table;

insert overwrite table test_data_table select cast(trim(val) as date) from temp_table;

insert overwrite table test_protected_data_table select ptyProtectDate(val,
‘Token_Date’) from test_data_table;

create table test_reprotected_data_table(val date) row format delimited fields
```

```
terminated by ',' stored as textfile;

insert overwrite table test_reprotected_data_table select ptyReprotect(val,
‘Token_Date’, 'new_Token_Date') from test_protected_data_table;
```

Table 3-45: Supported Protection Methods

HIVE UDFs	Supported Protection Methods					
	Tokenization	**Encryption**	**FPE**	**No Encryption**	**Masking**	**Monitoring**
ptyReprotect()	Date	No	No	Yes	No	Yes

> **Note:** The protection methods, that are not mentioned in the *Supported Protection Methods* table, are not supported.

3.2.32 ptyProtectDateTime()

This UDF protects *timestamp* format data, which is provided as input.

ptyProtectDateTime(Timestamp input, String dataElement)

Parameters

`Timestamp input:` The data in *timestamp* format, which needs to be protected

`String dataElement:` The data element that will be used to protect *timestamp* format data

Result

This UDF returns *timestamp* format data, which is protected.

Example

```
create temporary function ptyProtectDateTime AS
'com.protegrity.hive.udf.ptyProtectDateTime';

drop table if exists test_data_table;
drop table if exists temp_table;

create table temp_table(val timestamp) row format delimited fields terminated by ','
stored as textfile;

create table test_data_table(val timestamp) row format delimited fields terminated by
',' stored as textfile;

LOAD DATA LOCAL INPATH 'test_data.csv' OVERWRITE INTO TABLE temp_table;

insert overwrite table test_data_table select cast(trim(val) as timestamp) from
temp_table;

select ptyProtectDateTime(val, 'Token_Timestamp') from test_data_table;
```

Table 3-46: Supported Protection Methods

HIVE UDFs	Supported Protection Methods					
	Tokenization	**Encryption**	**FPE**	**No Encryption**	**Masking**	**Monitoring**
ptyProtectDateTime( )	Datetime	No	No	Yes	No	Yes

> **Note:** The protection methods, that are not mentioned in the *Supported Protection Methods* table, are not supported.

3.2.33 ptyUnprotectDateTime()

This UDF unprotects the protected timestamp format data, which is provided as input.

ptyUnprotectDateTime(Timestamp input, String dataElement)

Parameters

`Timestamp input:` The protected data in *timestamp* format, which needs to be unprotected

`String dataElement:` The data element that will be used to unprotect *timestamp* format data

Result

This UDF returns *timestamp* format data, which is unprotected.

Example

```
create temporary function ptyProtectDateTime AS
'com.protegrity.hive.udf.ptyProtectDateTime';

create temporary function ptyUnprotectDateTime AS
'com.protegrity.hive.udf.ptyUnprotectDateTime';

drop table if exists test_data_table;
drop table if exists temp_table;
drop table if exists protected_data_table;

create table temp_table(val timestamp) row format delimited fields terminated by ','
stored as textfile;

create table test_data_table(val timestamp) row format delimited fields terminated by
',' stored as textfile;

create table protected_data_table(protectedValue timestamp) row format delimited fields
terminated by ',' stored as textfile;

LOAD DATA LOCAL INPATH 'test_data.csv' OVERWRITE INTO TABLE temp_table;

insert overwrite table test_data_table select cast(trim(val) as timestamp) from
temp_table;

insert overwrite table protected_data_table select ptyProtectDateTime(val,
‘Token_Timestamp’) from test_data_table;

select ptyUnprotectDateTime(protectedValue, ‘Token_Timestamp’) from
protected_data_table;
```

Table 3-47: Supported Protection Methods

HIVE UDFs	Supported Protection Methods					
	Tokenization	**Encryption**	**FPE**	**No Encryption**	**Masking**	**Monitoring**
ptyUnprotectDateTime()	Datetime	No	No	Yes	No	Yes

Note: The protection methods, that are not mentioned in the *Supported Protection Methods* table, are not supported.

3.2.34 ptyReprotect()

This UDF reprotects *timestamp* format protected data, which was earlier protected using the *ptyProtectDateTime* UDF, with a different data element.

ptyReprotect(Timestamp input, varchar oldDataElement, varchar newDataElement)

Parameters

`Timestamp input:` The data in *timestamp* format, which needs to be reprotected

`varchar oldDataElement:` The data element that was used to protect the data earlier

`varchar newDataElement:` The new data element that will be used to reprotect the data

Result

This UDF returns *timestamp* format data, which is protected.

Example

```
create temporary function ptyProtectDateTime AS
'com.protegrity.hive.udf.ptyProtectDateTime';

create temporary function ptyReprotect AS 'com.protegrity.hive.udf.ptyReprotect';

drop table if exists test_data_table;
drop table if exists temp_table;

create table temp_table(val timestamp) row format delimited fields terminated by ','
stored as textfile;

create table test_data_table(val timestamp) row format delimited fields terminated by
',' stored as textfile;

create table test_protected_data_table(val timestamp) row format delimited fields
terminated by ',' stored as textfile;

LOAD DATA LOCAL INPATH 'test_data.csv' OVERWRITE INTO TABLE temp_table;

insert overwrite table test_data_table select cast(trim(val) as timestamp) from
temp_table;

insert overwrite table test_protected_data_table select ptyProtectDateTime(val,
‘Token_Timestamp’) from test_data_table;

create table test_reprotected_data_table(val timestamp) row format delimited fields
terminated by ',' stored as textfile;

insert overwrite table test_reprotected_data_table select ptyReprotect(val,
‘Token_Timestamp’, 'new_Token_Timestamp') from test_protected_data_table;
```

Table 3-48: Supported Protection Methods

HIVE UDFs	Supported Protection Methods					
	Tokenization	**Encryption**	**FPE**	**No Encryption**	**Masking**	**Monitoring**
ptyReprotect()	Datetime	No	No	Yes	No	Yes

Note: The protection methods, that are not mentioned in the *Supported Protection Methods* table, are not supported.

3.2.35 ptyProtectChar()

This UDF protects *character* value.

Note: It is recommended to use the String UDFs, such as *ptyProtectStr()*, *ptyUnprotectStr()*, or *ptyReprotect()* instead of the respective Char UDFs, such as *ptyProtectChar()*, *ptyUnprotectChar()*, or *ptyReprotect()* unless it is required to use the *char* data type only.

Note: For Date and Datetime type of data elements, the protect API returns an invalid input data error if the input value falls between the non-existent date range from 05-OCT-1582 to 14-OCT-1582 of the Gregorian Calendar.

For more information about the tokenization and de-tokenization of the cutover dates of the Proleptic Gregorian Calendar, refer to the section *Date* and *Datetime* tokenization in the *Protection Method Reference Guide 8.1.0.0*.

ptyProtectChar(Char input, String dataElement)

Parameters

`Char input:` *Character* value to protect

`String DataElement:` Name of the data element to protect *character* value

> **Warning:**
> If you have fixed length data fields and the input data is shorter than the length of the field, then ensure that you truncate the trailing white spaces and leading white spaces, if applicable, before passing the input to the respective Protect and Unprotect UDFs.
>
> The truncation of the white spaces ensures that the results of the protection and unprotection operations will result in consistent data output across the Protegrity products.

> **Warning:**
> Ensure that the lengths of the *Char* column in the source and target Hive tables are the same to avoid data corruption, since as per Hive behaviour, characters that exceed the defined *Char* column size, are truncated.
>
> The UDF only supports *Numeric*, *Alpha*, *Alpha Numeric*, *Upper-case Alpha*, *Upper Alpha-Numeric*, and *Email* tokenization data elements, and with length preservation selected.
>
> Using any other data elements with this UDF is not supported.
>
> Using non-length preserving data elements with this UDF is not supported.

Result

This UDF returns protected *character* value.

Example

```
create temporary function ptyProtectChar AS 'com.protegrity.hive.udf.ptyProtectChar';

drop table if exists temp_table;

create table temp_table(val char(10)) row format delimited fields terminated by ','
stored as textfile;

LOAD DATA LOCAL INPATH 'test_data.csv' OVERWRITE INTO TABLE temp_table;

select ptyProtectChar(val, 'TOKEN_ELEMENT') from temp_table;
```

Exception

`ptyHiveProtectorException: INPUT-ERROR: Integer type Data Element is not supported`: An *Integer* data element, which is unsupported, is provided.

`ptyHiveProtectorException: INPUT-ERROR: Non-length or Non-Format Preserving Data Element(s) is not supported`: A non-length preserving data element is provided.

Table 3-49: Supported Protection Methods

HIVE UDFs	Supported Protection Methods					
	Tokenization	Encryption	FPE	No Encryption	Masking	Monitoring
ptyProtectChar()	All length preserving tokens	No	No	Yes	Yes	Yes

Note: The protection methods, that are not mentioned in the *Supported Protection Methods* table, are not supported.

3.2.36 ptyUnprotectChar()

This UDF unprotects *character* value.

> **Note:** It is recommended to use the String UDFs, such as *ptyProtectStr()*, *ptyUnprotectStr()*, or *ptyReprotect()* instead of the respective Char UDFs, such as *ptyProtectChar()*, *ptyUnprotectChar()*, or *ptyReprotect()* unless it is required to use the *char* data type only.

> **Note:** For Date and Datetime type of data elements, the protect API returns an invalid input data error if the input value falls between the non-existent date range from 05-OCT-1582 to 14-OCT-1582 of the Gregorian Calendar.
>
> For more information about the tokenization and de-tokenization of the cutover dates of the Proleptic Gregorian Calendar, refer to the section *Date* and *Datetime* tokenization in the *Protection Method Reference Guide 8.1.0.0*.

ptyUnprotectChar(Char input, String dataElement)

Parameters

`Char input:` Protected *Character* value to unprotect

`String DataElement:` Name of data element that was used to protect the *character* value, to unprotect the *character* value

> **Warning:**
>
> If you have fixed length data fields and the input data is shorter than the length of the field, then ensure that you truncate the trailing white spaces and leading white spaces, if applicable, before passing the input to the respective Protect and Unprotect UDFs.
>
> The truncation of the white spaces ensures that the results of the protection and unprotection operations will result in consistent data output across the Protegrity products.

> **Warning:**
>
> Ensure that the lengths of the *Char* column in the source and target Hive tables are the same to avoid data corruption, since as per Hive behaviour, characters that exceed the defined *Char* column size, are truncated.
>
> The UDF only supports *Numeric*, *Alpha*, *Alpha Numeric*, *Upper-case Alpha*, *Upper Alpha-Numeric*, and *Email* tokenization data elements, and with length preservation selected.
>
> Using any other data elements with this UDF is not supported.
>
> Using non-length preserving data elements with this UDF is not supported.

Result

This UDF returns unprotected *character* value.

Example

```
create temporary function ptyProtectChar AS 'com.protegrity.hive.udf.ptyProtectChar';
create temporary function ptyUnprotectChar AS
'com.protegrity.hive.udf.ptyUnprotectChar';

drop table if exists test_data_table;
drop table if exists protected_data_table;

create table test_data_table(val char(10)) row format delimited fields terminated by
',' stored as textfile;
```

```
LOAD DATA LOCAL INPATH 'test_data.csv' OVERWRITE INTO TABLE test_data_table;

create table protected_data_table(protectedValue char(10)) row format delimited fields
terminated by ',' stored as textfile;
insert overwrite table protected_data_table select ptyProtectChar(val, 'TOKEN_ELEMENT')
from test_data_table;

select ptyUnprotectChar(protectedValue,'TOKEN_ELEMENT') FROM protected_data_table;
```

Exception

`ptyHiveProtectorException: INPUT-ERROR: Integer type Data Element is not supported`: An *Integer* data element, which is unsupported, is provided.

`ptyHiveProtectorException: INPUT-ERROR: Non-length or Non-Format Preserving Data Element(s) is not supported`: A non-length preserving data element is provided.

Table 3-50: Supported Protection Methods

HIVE UDFs	Supported Protection Methods					
	Tokenization	Encryption	FPE	No Encryption	Masking	Monitoring
ptyUnprotectChar()	All length preserving tokens	No	No	Yes	Yes	Yes

Note: The protection methods, that are not mentioned in the *Supported Protection Methods* table, are not supported.

3.2.37 ptyReprotect() - Char data

This UDF reprotects *character* format protected data with a different data element.

Note: It is recommended to use the String UDFs, such as *ptyProtectStr()*, *ptyUnprotectStr()*, or *ptyReprotect()* instead of the respective Char UDFs, such as *ptyProtectChar()*, *ptyUnprotectChar()*, or *ptyReprotect()* unless it is required to use the *char* data type only.

ptyReprotect(Char input, String oldDataElement, String newDataElement)

Parameters

`Char input:` *Character* value to reprotect

`String oldDataElement:` Protected *Character* value to unprotect

`String newDataElement:` Name of new data element to reprotect the datas

Warning:

If you have fixed length data fields and the input data is shorter than the length of the field, then ensure that you truncate the trailing white spaces and leading white spaces, if applicable, before passing the input to the respective Protect and Unprotect UDFs.

The truncation of the white spaces ensures that the results of the protection and unprotection operations will result in consistent data output across the Protegrity products.

Warning:

Ensure that the lengths of the *Char* column in the source and target Hive tables are the same to avoid data corruption, since as per Hive behaviour, characters that exceed the defined *Char* column size, are truncated.

The UDF only supports *Numeric*, *Alpha*, *Alpha Numeric*, *Upper-case Alpha*, *Upper Alpha-Numeric*, and *Email* tokenization data elements, and with length preservation selected.

Using any other data elements with this UDF is not supported.

Using non-length preserving data elements with this UDF is not supported.

Result

This UDF returns protected *character* value.

Example

```
create temporary function ptyProtectChar AS 'com.protegrity.hive.udf.ptyProtectChar';
create temporary function ptyUnprotectChar AS
'com.protegrity.hive.udf.ptyUnprotectChar';
create temporary function ptyReprotect AS 'com.protegrity.hive.udf.ptyReprotect';

drop table if exists test_data_table;
drop table if exists protected_data_table;
drop table if exists unprotected_data_table;
drop table if exists reprotected_data_table;

create table test_data_table(val char(10)) row format delimited fields terminated by
',' stored as textfile;
LOAD DATA LOCAL INPATH 'test_data.csv' OVERWRITE INTO TABLE test_data_table;

create table protected_data_table(val char(10)) row format delimited fields terminated
by ',' stored as textfile;
insert overwrite table protected_data_table select ptyProtectChar(val, 'TOKEN_ELEMENT')
from test_data_table;

create table reprotected_data_table(val char(10)) row format delimited fields
terminated by ',' stored as textfile;
insert overwrite table reprotected_data_table select ptyReprotect(val,
'old_Token_alpha', 'new_Token_alpha') from protected_data_table;

create table unprotected_data_table(val char(10)) row format delimited fields
terminated by ',' stored as textfile;
insert overwrite table unprotected_data_table select ptyUnprotectChar(val,
'TOKEN_ELEMENT') from reprotected_data_table;
```

Exception

`ptyHiveProtectorException: INPUT-ERROR: Integer type Data Element is not supported`: An *Integer* data element, which is unsupported, is provided.

`ptyHiveProtectorException: INPUT-ERROR: Non-length or Non-Format Preserving Data Element(s) is not supported`: A non-length preserving data element is provided.

Table 3-51: Supported Protection Methods

HIVE UDFs	Supported Protection Methods					
	Tokenization	Encryption	FPE	No Encryption	Masking	Monitoring
ptyReprotect() - Char data	All length preserving tokens	No	No	Yes	Yes	Yes

Note: The protection methods, that are not mentioned in the *Supported Protection Methods* table, are not supported.

3.2.38 ptyStringEnc()

This UDF encrypts *string* value.

ptyStringEnc(String input, String DataElement)

Parameters

`String input:` *String* value to encrypt

`String DataElement:` Name of the data element to encrypt *string* value

Warning:
The string encryption UDFs are limited to accept 2 GB data size at maximum as input.

Ensure that the field size for the protected binary data post the required encoding does not exceed the 2 GB input limit.

Warning:
The field size to store the input data is dependent on the encryption algorithm selected, such as AES-128, AES-256, 3DES, and CUSP, and the encoding type selected, such as No Encoding, Base64, and Hex.

Ensure that you set the input data size based on the required encryption algorithm and encoding so that the it does not exceed the 2 GB input limit.

For more information about estimating the field size of the data, refer to section *3.2.35.1 Guidelines for Estimating Field Size of Data*.

Result

This UDF returns encrypted *binary* value.

Example

```
create temporary function ptyStringEnc as 'com.protegrity.hive.udf.ptyStringEnc';

DROP TABLE IF EXISTS stringenc_data;
DROP TABLE IF EXISTS stringenc_data_protect;

CREATE TABLE stringenc_data (stringdata String) row format delimited fields terminated
by ',' stored as textfile;
LOAD DATA INPATH '/tmp/stringdata.csv' OVERWRITE INTO TABLE stringenc_data;

CREATE TABLE stringenc_data_protect (stringdata String) stored as textfile;

INSERT OVERWRITE TABLE stringenc_data_protect SELECT
base64(ptyStringEnc(stringdata,'AES128')) FROM stringenc_data;
```

Exception

`ptyHiveProtectorException: INPUT-ERROR: Tokenization or Format Preserving Data Elements are not supported`: A data element, which is unsupported, is provided.

`java.io.IOException: Too many bytes before newline: 2147483648`: The length of the input needs to be less than the maximum limit of 2 GB.

Table 3-52: Supported Protection Methods

HIVE UDFs	Supported Protection Methods					
	Tokenization	**Encryption**	**FPE**	**No Encryption**	**Masking**	**Monitoring**
ptyStringEnc()	No	• AES-128 • AES-256 • 3DES • CUSP	No	Yes	No	Yes

Note: The protection methods, that are not mentioned in the *Supported Protection Methods* table, are not supported.

3.2.38.1 Guidelines for Estimating Field Size of Data

The encryption algorithm and the field sizes in bytes required by the features, such as, Key ID (KID), Initialization Vector (IV), and Integrity Check (CRC) is listed in the following table.

Table 3-53: Encryption Algorithm and Field Sizes Required

Encryption Algorithm	KID (size in Bytes)	IV (size in Bytes)	CRC (size in Bytes)
AES	16	16	4
3DES	8	8	4
CUSP_TRDES	2	N/A	4
CUSP_AES	2	N/A	4

Note: The number of bytes considered for 1 GB and 2 GB are *1073741824* and *2147483648* respectively.

The byte sizes required by the input file, encoding type selected, and the encryption algorithm with the features selected is listed in the following table.

Table 3-54: Byte Sizes for the Encoding Type and Encryption Algorithm

Encoding Type	Encryption Algorithm			
	AES	3DES	CUSP_TRDES	CUSP_AES
AES	(Input file size in Bytes) + (Bytes needed by Encryption Algorithm and Features) <= 2147483647	(Input file size in Bytes) + (Bytes needed by Encryption Algorithm and Features) <= 2147483648		
3DES	(Input file size in Bytes) + (Bytes needed by Encryption Algorithm and Features) <= 1073741823	(Input file size in Bytes) + (Bytes needed by Encryption Algorithm and Features) <= 1073741824		
CUSP_TRDES	(Input file size in Bytes) + (Bytes needed by Encryption Algorithm and Features) <= 1610612735	(Input file size in Bytes) + (Bytes needed by Encryption Algorithm and Features) <= 1610612736		

3.2.39 ptyStringDec()

This UDF decrypts *binary* value.

ptyStringDec(Binary input, String DataElement)

Parameters

`Binary input:` Protected *Binary* value to unprotect

`String DataElement:` Name of data element that was used to encrypt the *string* value, to decrypt the *binary* value

Result

This UDF returns the decrypted *string* value.

Example

```
create temporary function ptyStringEnc as 'com.protegrity.hive.udf.ptyStringEnc';
create temporary function ptyStringDec as 'com.protegrity.hive.udf.ptyStringDec';

DROP TABLE IF EXISTS stringenc_data;
DROP TABLE IF EXISTS stringenc_data_protect;
DROP TABLE IF EXISTS stringenc_data_unprotect;

CREATE TABLE stringenc_data (stringdata String) row format delimited fields terminated
```

```
by ',' stored as textfile;
LOAD DATA INPATH '/tmp/stringdata.csv' OVERWRITE INTO TABLE stringenc_data;

CREATE TABLE stringenc_data_protect (stringdata String) stored as textfile;
INSERT OVERWRITE TABLE stringenc_data_protect SELECT
base64(ptyStringEnc(stringdata,'AES128')) FROM stringenc_data;

CREATE TABLE stringenc_data_unprotect (stringdata String) stored as textfile;
INSERT OVERWRITE TABLE stringenc_data_unprotect SELECT
ptyStringDec(unbase64(stringdata),'AES128') FROM stringenc_data_protect;
```

Exception

`ptyHiveProtectorException: INPUT-ERROR: First argument (Input Data to be unprotected) is not a valid Binary Datatype`: The input data, which is not in binary format is provided.

`ptyHiveProtectorException: INPUT-ERROR: Tokenization or Format Preserving Data Elements are not supported`: A data element, which is unsupported, is provided.

Table 3-55: Supported Protection Methods

HIVE UDFs	Supported Protection Methods					
	Tokenization	Encryption	FPE	No Encryption	Masking	Monitoring
ptyStringDec()	No	• AES-128 • AES-256 • 3DES • CUSP	No	Yes	No	Yes

Note: The protection methods, that are not mentioned in the *Supported Protection Methods* table, are not supported.

3.2.40 ptyStringReEnc()

This UDF reencrypts *Binary* format encrypted data with a different data element.

ptyStringReEnc(Binary input, String oldDataElement, String newDataElement)

Parameters

`Binary input:` *Binary* value to reencrypt

`String oldDataElement:` Name of data element used to encrypt the data earlier

`String newDataElement:` Name of new data element to reencrypt the data

Result

This UDF returns *binary* format data, which is reencrypted.

Example

```
create temporary function ptyStringEnc as 'com.protegrity.hive.udf.ptyStringEnc';
create temporary function ptyStringDec as 'com.protegrity.hive.udf.ptyStringDec';
create temporary function ptyStringReEnc as 'com.protegrity.hive.udf.ptyStringReEnc';

DROP TABLE IF EXISTS stringenc_data;
DROP TABLE IF EXISTS stringenc_data_protect;
DROP TABLE IF EXISTS stringenc_data_unprotect;
DROP TABLE IF EXISTS stringenc_data_reprotect;
DROP TABLE IF EXISTS stringenc_data_unprotect_after_reprotect;

CREATE TABLE stringenc_data (stringdata String) row format delimited fields terminated
by ',' stored as textfile;
LOAD DATA INPATH '/tmp/stringdata.csv' OVERWRITE INTO TABLE stringenc_data;

CREATE TABLE stringenc_data_protect (stringdata String) stored as textfile;
INSERT OVERWRITE TABLE stringenc_data_protect SELECT
base64(ptyStringEnc(stringdata,'AES128')) FROM stringenc_data;
```

```
CREATE TABLE stringenc_data_unprotect (stringdata String) stored as textfile;
INSERT OVERWRITE TABLE stringenc_data_unprotect SELECT
ptyStringDec(unbase64(stringdata),'AES128') FROM stringenc_data_protect;

CREATE TABLE stringenc_data_reprotect (stringdata String) stored as textfile;
INSERT OVERWRITE TABLE stringenc_data_reprotect SELECT
base64(ptyStringReEnc(unbase64(stringdata),'AES128','AES128_KID')) FROM
stringenc_data_protect;

CREATE TABLE stringenc_data_unprotect_after_reprotect (stringdata String) stored as
textfile;
INSERT OVERWRITE TABLE stringenc_data_unprotect_after_reprotect SELECT
ptyStringDec(unbase64(stringdata),'AES128_KID') FROM stringenc_data_reprotect;
```

Exception

`ptyHiveProtectorException: INPUT-ERROR: First argument (Input Data to be reprotected) is not a valid Binary Datatype`: The input data, which is not in binary format is provided.

`java.io.IOException: Too many bytes before newline: 2147483648`: The length of the input needs to be less than the maximum limit of 2 GB.

`com.protegrity.hive.udf.ptyHiveProtectorException: 26, Unsupported algorithm or unsupported action for the specific data element`: The data element is not supported for this UDF.

Table 3-56: Supported Protection Methods

HIVE UDFs	Supported Protection Methods					
	Tokenization	**Encryption**	**FPE**	**No Encryption**	**Masking**	**Monitoring**
ptyStringReEnc()	No	• AES-128 • AES-256 • 3DES • CUSP	No	Yes	No	Yes

Note: The protection methods, that are not mentioned in the *Supported Protection Methods* table, are not supported.

3.3 Pig UDFs

This section describes all Pig UDFs that are available for protection and unprotection in Big Data Protector to build secure Big Data applications.

3.3.1 ptyGetVersion()

This UDF returns the current version of PEP.

ptyGetVersion()

Parameters

None

Result

`chararray` : Version number

Example

```
REGISTER /opt/protegrity/Hadoop-protector/lib/peppig-0.10.0.jar;
                                                // register pep pig version
DEFINE ptyGetVersion com.protegrity.pig.udf.ptyGetVersion;
                                                //define UDF
employees = LOAD ‘employee.csv’ using PigStorage(‘,’)
```

```
                    AS (eid:chararray,name:chararray, ssn:chararray);
                                                    // load employee.csv from HDFS path
version = FOREACH employees GENERATE ptyGetVersion();
DUMP version;
```

3.3.2 ptyWhoAmI()

This UDF returns the current logged in user name.

ptyWhoAmI()

Parameters

None

Result

`chararray` : User name

Example

```
REGISTER /opt/protegrity/Hadoop_protector/lib/peppig-0.10.0.jar;
DEFINE ptyWhoAmI com.protegrity.pig.udf.ptyWhoAmI;
employees = LOAD 'employee.csv' using PigStorage(',')
            AS (eid:chararray, name:chararray, ssn:chararray);
username = FOREACH employees GENERATE ptyWhoAmI();
DUMP username;
```

3.3.3 ptyProtectInt()

This UDF returns protected value for *integer* data.

ptyProtectInt (int data, chararray dataElement)

Parameters

`int data` : Data to protect

`chararray dataElement`: Name of data element to use for protection

Result

Protected value for given numeric data

Example

```
REGISTER /opt/protegrity/hadoop_protector/lib/peppig-0.10.0.jar;
DEFINE ptyProtectInt com.protegrity.pig.udf.ptyProtectInt;
employees = LOAD 'employee.csv' using PigStorage(',') AS (eid:int, name:chararray, ssn:c
hararray);
data_p = FOREACH employees GENERATE ptyProtectInt(eid, 'token_integer');
DUMP data_p;
```

Table 3-57: Supported Protection Methods

Pig UDFs	Supported Protection Methods					
	Tokenization	Encryption	FPE	No Encryption	Masking	Monitoring
ptyProtectInt()	Integer 4 Bytes	No	No	Yes	No	Yes

Note: The protection methods, that are not mentioned in the *Supported Protection Methods* table, are not supported.

3.3.4 ptyUnprotectInt()

This UDF returns unprotected value for protected *integer* data.

ptyUnprotectInt (int data, chararray dataElement)

Parameters

`int data` : Protected data

`chararray dataElement`: Name of data element to use for unprotection

Result

Unprotected value for given protected *integer* data

Example

```
REGISTER /opt/protegrity/hadoop_protector/lib/peppig-0.10.0.jar;
DEFINE ptyProtectInt com.protegrity.pig.udf.ptyProtectInt;
DEFINE ptyUnprotectInt com.protegrity.pig.udf.ptyUnProtectInt;
employees = LOAD ‘employee.csv’ using PigStorage(‘,’) AS (eid:int, name:chararray,
ssn:chararray);
data_p = FOREACH employees GENERATE ptyProtectInt(eid, ‘token_integer’);
data_u = FOREACH data_p GENERATE ptyUnprotectInt(eid, ‘token_integer’);
DUMP data_u;
```

Table 3-58: Supported Protection Methods

Pig UDFs	Supported Protection Methods					
	Tokenization	**Encryption**	**FPE**	**No Encryption**	**Masking**	**Monitoring**
ptyUnprotectInt()	Integer 4 Bytes	No	No	Yes	No	Yes

Note: The protection methods, that are not mentioned in the *Supported Protection Methods* table, are not supported.

3.3.5 ptyProtectStr()

This UDF protects *string* value.

Note: For Date and Datetime type of data elements, the protect API returns an invalid input data error if the input value falls between the non-existent date range from 05-OCT-1582 to 14-OCT-1582 of the Gregorian Calendar.

For more information about the tokenization and de-tokenization of the cutover dates of the Proleptic Gregorian Calendar, refer to the section *Date* and *Datetime* tokenization in the *Protection Method Reference Guide 8.1.0.0*.

ptyProtectStr(chararray input, chararray dataElement)

Parameters

`chararray data`: *String* value to protect

`chararray dataElement`: Name of data element to unprotect *string* value

Result

chararray

Example

```
REGISTER /opt/protegrity/hadoop_protector/lib/peppig-0.10.0.jar;
DEFINE ptyProtectStr com.protegrity.pig.udf.ptyProtectStr;
employees = LOAD ‘employee.csv’ using PigStorage(‘,’) AS (eid:chararray,
name:chararray, ssn:chararray);
data_p = FOREACH employees GENERATE ptyProtectIntStr(name, ‘token_alphanumeric’);
DUMP data_p
```

Table 3-59: Supported Protection Methods

Pig UDFs	Supported Protection Methods					
	Tokenization	**Encryption**	**FPE**	**No Encryption**	**Masking**	**Monitoring**
ptyProtectStr()	• Numeric (0-9)	No	Yes	Yes	Yes	Yes

Pig UDFs	Supported Protection Methods					
	Tokenization	Encryption	FPE	No Encryption	Masking	Monitoring
	• Credit Card • Alpha (A-Z) • Upper-case Alpha (A-Z) • Alpha-Numeric (0-9, a-z, A-Z) • Upper Alpha-Numeric (0-9, A-Z) • Lower ASCII • Date (YYYY-MM-DD, DD/MM/YYYY, MM.DD.YYYY) • Datetime (YYYY-MM-DD HH:MM:SS) • Decimal • Email					

Note: The protection methods, that are not mentioned in the *Supported Protection Methods* table, are not supported.

3.3.6 ptyUnprotectStr()

This UDF unprotects protected *string* value.

Note: For Date and Datetime type of data elements, the protect API returns an invalid input data error if the input value falls between the non-existent date range from 05-OCT-1582 to 14-OCT-1582 of the Gregorian Calendar.

For more information about the tokenization and de-tokenization of the cutover dates of the Proleptic Gregorian Calendar, refer to the section *Date* and *Datetime* tokenization in the *Protection Method Reference Guide 8.1.0.0*.

ptyUnprotectStr (chararray input, chararray dataElement)

Parameters

`chararray input` : Unprotected *string* value

`chararray dataElement`: Name of data element to unprotect *string* value

Result

`chararray`: Unprotected value

Example

```
REGISTER /opt/protegrity/hadoop_protector/lib/peppig-0.10.0.jar;
DEFINE ptyProtectInt com.protegrity.pig.udf.ptyProtectStr;
DEFINE ptyUnprotectInt com.protegrity.pig.udf.ptyUnProtectStr;
employees = LOAD ‘employee.csv’ using PigStorage(‘,’) AS (eid:chararray,
name:chararray, ssn:chararray);
data_p = FOREACH employees
          GENERATE ptyProtectStr(name, ‘token_alphanumeric’) as name:chararray
DUMP data_p;
data_u = FOREACH data_p GENERATE ptyUnprotectStr(ssn, ‘Token_alphanumeric’);
DUMP data_u;
```

Table 3-60: Supported Protection Methods

Pig UDFs	Supported Protection Methods					
	Tokenization	**Encryption**	**FPE**	**No Encryption**	**Masking**	**Monitoring**
ptyUnprotectStr()	• Numeric (0-9) • Credit Card • Alpha (A-Z) • Upper-case Alpha (A-Z) • Alpha-Numeric (0-9, a-z, A-Z) • Upper Alpha-Numeric (0-9, A-Z) • Lower ASCII • Date (YYYY-MM-DD, DD/MM/YYYY, MM.DD.YYYY) • Datetime (YYYY-MM-DD HH:MM:SS) • Decimal • Email	No	Yes	Yes	Yes	Yes

Note: The protection methods, that are not mentioned in the *Supported Protection Methods* table, are not supported.

3.4 HDFSFP Commands (Deprecated from Big Data Protector 7.2.0)

Hadoop provides shell commands for modifying and administering HDFS. HDFSFP extends the modification commands to control access to files and directories in HDFS.

The section describes the commands supported in HDFSFP.

Note:

Starting from the Big Data Protector 7.2.0 release, the HDFS File Protector (HDFSFP) is deprecated. This section is retained to ensure coverage for using an older version of Big Data Protector with the ESA 7.2.0.

3.4.1 copyFromLocal

This command ingests local data into HDFS.

hadoop ptyfs -copyFromLocal <local path of file to copy> <destination HDFS directory path>

Result

- If the destination directory path is protected and the user executing the command has permissions to create and protect, then the data is ingested in encrypted form.
- If the destination directory path is protected and the user does not have permissions to create and protect, then the copy operation fails.
- If the destination HDFS directory path is not protected, then the data is ingested in clear form.

3.4.2 put

This command ingests local data into HDFS.

hadoop ptyfs -put <local path of file to copy> <destination HDFS directory path>

Result

- If the destination HDFS directory path is protected and the user executing the command has permissions to create and protect, then the data is ingested in encrypted form.
- If the destination HDFS directory path is protected and the user does not have permissions to create and protect, then the copy operation fails.
- If the destination HDFS directory path is not protected, then the data is ingested in clear form.

3.4.3 copyToLocal

This command is used to copy an HDFS file to a local directory.

hadoop ptyfs -copyToLocal <HDFS file path to copy> <destination local directory>

Result

- If the source HDFS file is protected and the user has unprotect permissions, then the file is copied to the destination directory in clear form.
- If the source HDFS file is not protected, then the file is copied to the destination directory.
- If the HDFS file is protected the user does not have unprotect permissions, then the copy operation fails.

3.4.4 get

This command copies an HDFS file to a local directory.

hadoop ptyfs -get <HDFS file path to copy> <destination local directory>

Result

- If the source HDFS file is protected and the user has unprotect permissions, then the file is copied to the destination directory in clear form.
- If the source HDFS file is not protected, then the file is copied to the destination directory.
- If the HDFS file is protected the user does not have unprotect permissions, then the copy operation fails.

3.4.5 cp

This command copies a file from one HDFS directory to another HDFS directory.

hadoop ptyfs -cp <source HDFS file path> <destination HDFS directory path>

Result

- If the source HDFS file is protected and the user has unprotect permissions for the source HDFS file, the destination directory is protected, and the user has permissions to protect and create on the destination HDFS directory path, then the file gets copied in encrypted form.
- If the source HDFS file is protected and the user does not have permissions to unprotect, then the copy operation fails.
- If the destination directory is protected and the user does not have permissions to protect and create, then the copy operation fails.
- If the source HDFS file is unprotected and destination directory is protected and the user has permissions to protect or create on the destination HDFS directory path, then the file is copied in encrypted form.
- If the source HDFS file is protected and the user has permissions to unprotect for the source HDFS file and destination HDFS directory path is not protected, then the file is copied in clear form.

- If the source HDFS file and destination HDFS directory path are unprotected, then the command works similar to the default Hadoop file shell *-cp* command.

3.4.6 mkdir

This command creates a new directory in HDFS.

hadoop ptyfs -mkdir <new HDFS directory path>

Result

- If the new directory is protected and the user has permissions to create, then the new directory is created.
- If the new directory is not protected, then this command runs similar to the default HDFS file shell *-mkdir* command.

3.4.7 mv

This command moves an HDFS file from one HDFS directory to another HDFS directory.

hadoop ptyfs -mv <source HDFS file path> <destination HDFS directory path>

Result

- If the source HDFS file is protected and the user has unprotect and delete permissions and the destination directory is also protected with the user having permissions to protect and create on the destination HDFS directory path, then the file is moved to the destination directory in encrypted form.
- If the HDFS file is protected and the user does not have unprotect and delete permissions or the destination directory is protected and the user does not have permissions to protect and create, then the move operation fails.
- If the source HDFS file is unprotected, the destination directory is protected and the user has permissions to protect and create on the destination HDFS directory path, then the file is copied in encrypted form.
- If the source HDFS file is protected and the user has permissions to unprotect and the destination HDFS directory path is not protected, then the file is copied in clear form.
- If the source HDFS file and destination HDFS directory path are unprotected, then the command works similar to the default Hadoop file shell *-cp* command.

3.4.8 rm

This command deletes HDFS files.

hadoop ptyfs -rm <HDFS file paths to delete>

Result

- If the HDFS file is protected and the user has permissions to delete on the HDFS file path, then the file is deleted.
- If the HDFS file is protected and the user does not have permissions to delete on the HDFS file path, then the delete operation fails.
- If the HDFS file is not protected, then the command works similar to the default Hadoop file shell *-rm* command.

3.4.9 rmr

This command deletes an HDFS directory, its subdirectories and files.

hadoop ptyfs -rmr <HDFS directory path to delete>

Result

- If the HDFS directory path is protected and the user has permissions to delete on the HDFS directory path, then the directory and its contents are deleted.
- If the HDFS directory path is protected and the user does not have permissions to delete on the HDFS directory path, then the delete operation fails.

- If the HDFS directory path is not protected, then the command works as the default Hadoop rm recursive (*hadoop fs -rmr* or *hadoop fs –rm -r*) command.

3.5 HDFSFP Java API (Deprecated from Big Data Protector 7.2.0)

Protegrity provides a Java API for working with files and directories using HDFSFP. The Java API for HDFSFP provides an alternate means of working with HDFSFP besides the HDFSFP shell commands, hadoop ptyfs, and enables you to integrate HDFSFP with Java applications.

The section describes the Java API commands supported in HDFSFP.

> **Note:**
> Starting from the Big Data Protector 7.2.0 release, the HDFS File Protector (HDFSFP) is deprecated. This section is retained to ensure coverage for using an older version of Big Data Protector with the ESA 7.2.0.

3.5.1 copy

This command copies a file from one HDFS directory to another HDFS directory.

copy(java.lang.String srcs, java.lang.String dst)

Parameters

`src`: HDFS file path

`dst`: HDFS file or directory path

Returns

`True`: If the operation is successful

`Exception`: If the operation fails

Exception (and Error Codes)

The API returns an exception (*com.protegrity.hadoop.fileprotector.fs.ProtectorException*) if any of the following conditions are met:

- Input is null.
- The path does not exist.
- The user does not have protect and write permissions on the destination path in case the destination path is protected, or the user does not have unprotect permission on the source path or both.

For more information on exceptions, refer to the Javadoc provided with the HDFSFP Java API. The Javadoc can be found in *<protegrity_base_directory>/protegrity/hdfsfp/doc* on the Data Ingestion Node.

Result

- If the source HDFS file is protected and the user has unprotect permission for the source HDFS file, the destination directory is protected, the ACL entry for the directory is activated, and the user has permissions to protect and create on the destination HDFS directory path, then the file gets copied in encrypted form.
- If the source HDFS file is protected and the user does not have permission to unprotect, then the copy operation fails.
- If the destination directory is protected and the user does not have permissions to protect and create, then the copy operation fails.
- If the source HDFS file is unprotected and destination directory is protected, the ACL entry for the directory is activated, and the user has permissions to protect or create on the destination HDFS directory path, then the file is copied in encrypted form.
- If the source HDFS file is protected and the user has permissions to unprotect for the source HDFS file and destination HDFS directory path is not protected, then the file is copied in clear form.

3.5.2 copyFromLocal

This command ingests local data into HDFS.

copyFromLocal(java.lang.String[] srcs, java.lang.String dst)

Parameters

`srcs` : Array of local file paths

`dst` : HDFS directory path

Returns

`True` : If the operation is successful

`Exception` : If the operation fails

Exception (and Error Codes)

The API returns an exception (*com.protegrity.hadoop.fileprotector.fs.ProtectorException*) if any of the following conditions are met:

- Input is null.
- The path does not exist.
- The user does not have protect and write permissions on the destination path if it is protected.

For more information on exceptions, refer to the Javadoc provided with the HDFSFP Java API. The Javadoc can be found in *<protegrity_base_directory>/protegrity/hdfsfp/doc* on the Data Ingestion Node.

Result

- If the destination directory path is protected, the ACL entry for the directory is activated, and the user executing the command has permissions to create and protect, then the data is ingested in encrypted form.
- If the destination directory path is protected and the user does not have permissions to create and protect, then the copy operation fails.
- If the destination HDFS directory path is not protected, then the data is ingested in clear form.

3.5.3 copyToLocal

This command is used to copy an HDFS file or directory to a local directory.

copyToLocal(java.lang.String srcs, java.lang.String dst)

Parameters

`srcs` : HDFS file or directory path

`dst` : Local directory or file path

Returns

`True` : If the operation is successful

`Exception` : If the operation fails

Exception (and Error Codes)

The API returns an exception (*com.protegrity.hadoop.fileprotector.fs.ProtectorException*) if any of the following conditions are met:

- Input is null.
- The path does not exist.
- The user does not have protect and write permissions on the destination path if it is protected.

For more information on exceptions, refer to the Javadoc provided with the HDFSFP Java API. The Javadoc can be found in *<protegrity_base_directory>/protegrity/hdfsfp/doc* on the Data Ingestion Node.

Result

- If the source HDFS file is protected, the ACL entry for the directory is activated, and the user has unprotect permission, then the file is copied to the destination directory in clear form.

- If the source HDFS file is not protected, then the file is copied to the destination directory.
- If the HDFS file is protected the user does not have unprotect permissions, then the copy operation fails.

3.5.4 deleteFile

This command deletes files from HDFS.

deleteFile(java.lang.String srcf, boolean skipTrash)

Parameters

`srcf` : HDFS file or directory path

`skipTrash` : Boolean value which decides if the file should be moved to trash. If the Boolean value is true, then the file is not moved to trash;

if false, then the file is moved to trash.

Returns

`True` : If the operation is successful

`Exception` : If the operation fails

Exception (and Error Codes)

The API returns an exception (*com.protegrity.hadoop.fileprotector.fs.ProtectorException*) if any of the following conditions are met:

- Input is null.
- The path does not exist.
- The user does not have protect and write permissions on the destination path if it is protected.

For more information on exceptions, refer to the Javadoc provided with the HDFSFP Java API. The Javadoc can be found in *<protegrity_base_directory>/protegrity/hdfsfp/doc* on the Data Ingestion Node.

Result

- If the HDFS file is protected and the user has permission to delete on the HDFS file path, then the file is deleted.
- If the HDFS file is protected and the user does not have permission to delete on the HDFS file path, then the delete operation fails.

3.5.5 deleteDir

This command deletes recursively an HDFS directory, its subdirectories, and files.

deleteDir(java.lang.String srcdir, boolean skipTrash)

Parameters

`srcdir` : HDFS file or directory path

`skipTrash` : Boolean value which decides if the file should be moved to trash. If the Boolean value is true, then the file is not moved to trash;

if false, then the file is moved to trash.

Returns

`True` : If the operation is successful

`Exception` : If the operation fails

Exception (and Error Codes)

The API returns an exception (*com.protegrity.hadoop.fileprotector.fs.ProtectorException*) if any of the following conditions are met:

- Input is null.

- The path does not exist.
- The user does not have protect and write permissions on the destination path if it is protected.

For more information on exceptions, refer to the Javadoc provided with the HDFSFP Java API. The Javadoc can be found in *<protegrity_base_directory>/protegrity/hdfsfp/doc* on the Data Ingestion Node.

Result

- If the HDFS directory path is protected and the user has permission to delete on the HDFS directory path, then the directory and its contents are deleted.
- If the HDFS directory path is protected and the user does not have permission to delete on the HDFS directory path, then the delete operation fails.

3.5.6 mkdir

This command creates a new directory in HDFS.

mkdir(java.lang.String dir)

Parameters

`dir` : HDFS file or directory path

Returns

`True` : If the operation is successful

`Exception` : If the operation fails

Exception (and Error Codes)

The API returns an exception (*com.protegrity.hadoop.fileprotector.fs.ProtectorException*) if any of the following conditions are met:

- Input is null.
- The user does not have write permissions to the path.

For more information on exceptions, refer to the Javadoc provided with the HDFSFP Java API. The Javadoc can be found in *<protegrity_base_directory>/protegrity/hdfsfp/doc* on the Data Ingestion Node.

Result

- If the new directory path exists in ACL or the ACL path for the parent directory path is activated recursively, and the user has permissions to create, then the new directory with an activated ACL path is created.
- If the new directory path or its parent directory path is not present in ACL recursively, then the new directory is created without HDFSFP protection.

3.5.7 move

This command moves an HDFS file from one HDFS directory to another HDFS directory.

move(java.lang.String src, java.lang.String dst)

Parameters

`src` : HDFS file path

`dst` : HDFS file or directory path

Returns

`True` : If the operation is successful

`Exception` : If the operation fails

Exception (and Error Codes)

The API returns an exception (*com.protegrity.hadoop.fileprotector.fs.ProtectorException*) if any of the following conditions are met:

- Input is null.

- The path does not exist.
- The user does not have unprotect and read, or protect and write, or create permissions to the path.
- The user does not have protect and write permissions on the destination path in case the destination path is protected, or the user does not have unprotect permission on the source path or both.

For more information on exceptions, refer to the Javadoc provided with the HDFSFP Java API. The Javadoc can be found in *<protegrity_base_directory>/protegrity/hdfsfp/doc* on the Data Ingestion Node.

Result

- If the source HDFS file is protected, the ACL entry for the directory is activated, and the user has unprotect and delete permissions and the destination directory is also protected with the user having permissions to protect and create on the destination HDFS directory path, then the file is moved to the destination directory in encrypted form.
- If the HDFS file is protected and the user does not have unprotect and delete permissions or the destination directory is protected and the user does not have permissions to protect and create, then the move operation fails.
- If the source HDFS file is unprotected, the destination directory is protected, the ACL entry for the directory is activated, and the user has permissions to protect and create on the destination HDFS directory path, then the file is copied in encrypted form.
- If the source HDFS file is protected and the user has permission to unprotect and the destination HDFS directory path is not protected, then the file is copied in clear form.

3.6 HBase Commands

Hadoop provides shell commands to ingest, extract, and display the data in an HBase table.

The section describes the commands supported by HBase.

Warning:

If you are using HBase shell, it is not recommended to use Format Preserving Encryption (FPE).

If you are using HBase Java API (Byte APIs), then ensure that the encoding, which is used to convert the *string* input data to *bytes*, matches the encoding that is selected in the **Plaintext Encoding** drop-down for the required FPE data element.

3.6.1 put

This command ingests the data provided by the user in protected form, using the configured data elements, into the required row and column of an HBase table. You can use this command to ingest data into all the columns for the required row of the HBase table.

Note: For Date and Datetime type of data elements, the protect API returns an invalid input data error if the input value falls between the non-existent date range from 05-OCT-1582 to 14-OCT-1582 of the Gregorian Calendar.

For more information about the tokenization and de-tokenization of the cutover dates of the Proleptic Gregorian Calendar, refer to the section *Date* and *Datetime* tokenization in the *Protection Method Reference Guide 8.1.0.0*.

```
put '<table_name>','<row_number>', 'column_family:<column_name>', '<data>'
```

Parameters

`table_name` : Name of the table.

`row_number` : Number of the row in the HBase table.

`column_family`: Name of the column family.

3.6.2 get

This command displays the protected data from the required row and column of an HBase table in cleartext form. You can use this command to display the data contained in all the columns of the required row of the HBase table.

```
get '<table_name>','<row_number>', 'column_family:<column_name>'
```

Parameters

`table_name` : Name of the table.

`row_number` : Number of the row in the HBase table.

`column_family`: Name of the column family.

> **Note:** Ensure that the logged in user has the permissions to view the protected data in cleartext form. If the user does not have the permissions to view the protected data, then only the protected data appears.

3.6.3 scan

This command displays the data from the HBase table in protected or unprotected form.

View the protected data using the following command.

```
scan '<table_name>', { ATTRIBUTES => {'BYPASS_COPROCESSOR'=>'1'}}
```

View the unprotected data using the following command.

```
scan '<table_name>'
```

Parameters

`table_name` : Name of the table.

`ATTRIBUTES` : Additional parameters to consider when displaying the protected or unprotected data.

> **Note:** Ensure that the logged in user has the permissions to unprotect the protected data. If the user does not have the permissions to unprotect the protected data, then only the protected data appears.

3.7 Impala UDFs

This section describes all Impala UDFs that are available for protection and unprotection in Big Data Protector to build secure Big Data applications.

3.7.1 pty_GetVersion()

This UDF returns the PEP version number.

```
ptyGetVersion()
```

Parameters

None

Result

This UDF returns the current version of the PEP.

Example

```
select pty_GetVersion ();
```

3.7.2 pty_WhoAmI()

This UDF returns the logged in user name.

ptyWhoAmI()

Parameters

None

Result

`Text` : Returns the logged in user name.

Example

```
select pty_WhoAmI();
```

3.7.3 pty_GetCurrentKeyId()

This UDF returns the current active key identification number of the encryption type data element.

pty_GetCurrentKeyId(dataElement string)

Parameters

`dataElement`: Variable specifies the protection method

Result

`integer` : Returns the current key identification number

Example

```
select pty_GetCurrentKeyId('enc_3des_kid');
```

3.7.4 pty_GetKeyId()

This UDF returns the key ID used for each row in a table.

pty_GetKeyId(dataElement string, col string)

Parameters

`dataElement` : Variable specifies the protection method

`col`: String array of the data in table

Result

`integer` : Returns the key indentification number for the row

Example

```
select pty_GetKeyId('enc_3des_kid',column_name) from table_name;
```

3.7.5 pty_StringEnc()

This UDF returns the encrypted value for a column containing *String* format data.

pty_StringEnc(data string, dataElement string)

Parameters

`data` : Column name of the data to encrypt in the table

`dataElement`: Variable specifying the protection method

Result

`string` : Returns a string value

Example

```
select pty_StringEnc(column_name,'enc_3des') from table_name;
```

Table 3-61: Supported Protection Methods

Impala UDFs	Supported Protection Methods					
	Tokenization	**Encryption**	**FPE**	**No Encryption**	**Masking**	**Monitoring**
pty_StringEnc()	No	• AES-128 • AES-256 • 3DES • CUSP	No	Yes	Yes	Yes

Note: The protection methods, that are not mentioned in the *Supported Protection Methods* table, are not supported.

3.7.6 pty_StringDec()

This UDF returns the decrypted value for a column containing *String* format data.

pty_StringDec(data string, dataElement string)

Parameters

`data` : Column name of the data to decode in the table

`dataElement`: Variable specifying the unprotection method

Result

`string` : Returns a string value

Example

```
select pty_StringDec(column_name,'enc_3des') from table_name;
```

Table 3-62: Supported Protection Methods

Impala UDFs	Supported Protection Methods					
	Tokenization	**Encryption**	**FPE**	**No Encryption**	**Masking**	**Monitoring**
pty_StringDec()	No	• AES-128 • AES-256 • 3DES • CUSP	No	Yes	Yes	Yes

Note: The protection methods, that are not mentioned in the *Supported Protection Methods* table, are not supported.

3.7.7 pty_StringIns()

This UDF returns the tokenized value for a column containing *String* format data.

Note: For Date and Datetime type of data elements, the protect API returns an invalid input data error if the input value falls between the non-existent date range from 05-OCT-1582 to 14-OCT-1582 of the Gregorian Calendar.

For more information about the tokenization and de-tokenization of the cutover dates of the Proleptic Gregorian Calendar, refer to the section *Date* and *Datetime* tokenization in the *Protection Method Reference Guide 8.1.0.0*.

pty_StringIns(data string, dataElement string)

Parameters

`data`: Column name of the data to tokenize in the table

`dataElement`: Variable specifying the protection method

Result

`string`: Returns the tokenized string value

Example

```
select pty_StringIns(column_name, 'TOK_NAME') from table_name;
```

Table 3-63: Supported Protection Methods

Impala UDFs	Supported Protection Methods					
	Tokenization	**Encryption**	**FPE**	**No Encryption**	**Masking**	**Monitoring**
pty_StringIns()	• Numeric (0-9) • Credit Card • Alpha • Upper Case Alpha • Alpha Numeric • Upper Alpha Numeric • Lower ASCII • Printable • Datetime (YYYY-MM-DD HH:MM:SS) • Date (YYYY-MM-DD, DD/MM/YYYY, MM.DD.YYYY) • Decimal • Email • Unicode (Legacy) • Unicode (Base64 - Encoded Byte's Charset should match Dataelement's Encoding Type) • Unicode (Gen2)	No	No	Yes	Yes	Yes

Note: The protection methods, that are not mentioned in the *Supported Protection Methods* table, are not supported.

3.7.8 pty_StringSel()

This UDF returns the detokenized value for a column containing *String* format data.

Note: For Date and Datetime type of data elements, the protect API returns an invalid input data error if the input value falls between the non-existent date range from 05-OCT-1582 to 14-OCT-1582 of the Gregorian Calendar.

For more information about the tokenization and de-tokenization of the cutover dates of the Proleptic Gregorian Calendar, refer to the section *Date* and *Datetime* tokenization in the *Protection Method Reference Guide 8.1.0.0*.

pty_StringSel(data string, dataElement string)

Parameters

`data`: Column name of the data to detokenize in the table

`dataElement`: Variable specifing the unprotection method

Result

`string`: Returns the detokenized string value

Example

```
select pty_StringSel(column_name, 'TOK_NAME') from table_name;
```

Table 3-64: Supported Protection Methods

Impala UDFs	Supported Protection Methods					
	Tokenization	**Encryption**	**FPE**	**No Encryption**	**Masking**	**Monitoring**
pty_StringSel()	• Numeric (0-9) • Credit Card • Alpha • Upper Case Alpha • Alpha Numeric • Upper Alpha Numeric • Lower ASCII • Printable • Datetime (YYYY-MM-DD HH:MM:SS) • Date (YYYY-MM-DD, DD/MM/YYYY, MM.DD.YYYY) • Decimal • Email • Unicode (Legacy) • Unicode (Base64 - Encoded Byte's Charset should match Dataelement's Encoding Type) • Unicode (Gen2)	No	No	Yes	Yes	Yes

Note: The protection methods, that are not mentioned in the *Supported Protection Methods* table, are not supported.

3.7.9 pty_UnicodeStringIns()

This UDF returns the tokenized value for a column containing *String* (Unicode) format data.

pty_UnicodeStringIns(data string, dataElement string)

Parameters

`data`: Column name of the *string* (Unicode) format data to tokenize in the table

`dataElement`: Name of data element to protect *string* (Unicode) value

> **Warning:**
> This UDF should be used only if you need to tokenize Unicode data in Impala, and migrate the tokenized data from Impala to a Teradata database and detokenize the data using the Protegrity Database Protector.
>
> Ensure that you use this UDF with a Unicode tokenization data element only.
>
> For more information about migrating tokenized Unicode data to a Teradata database, refer to the *Big Data Protector Guide 9.1.0.0*.

Result

This UDF returns protected *string* value.

Example

```
select pty_UnicodeStringIns(val, 'Token_unicode') from temp_table;
```

Table 3-65: Supported Protection Methods

Impala UDFs	Supported Protection Methods					
	Tokenization	**Encryption**	**FPE**	**No Encryption**	**Masking**	**Monitoring**
pty_UnicodeStringIns()	• Unicode (Legacy) • Unicode (Base64)	No	No	Yes	No	Yes

> **Note:** The protection methods, that are not mentioned in the *Supported Protection Methods* table, are not supported.

3.7.10 pty_UnicodeStringSel()

This UDF unprotects the existing protected String value.

pty_UnicodeStringSel(data string, dataElement string)

Parameters

`data`: Column name of the *string* format data to detokenize in the table

`varchar dataElement`: Name of data element to unprotect *string* value

> **Warning:**
> This UDF should be used only if you need to tokenize Unicode data in Teradata using the Protegrity Database Protector, and migrate the tokenized data from a Teradata database to Impala and detokenize the data using the Protegrity Big Data Protector for Impala.
>
> Ensure that you use this UDF with a Unicode tokenization data element only.
>
> For more information about migrating tokenized Unicode data to a Teradata database, refer to *Big Data Protector Guide 9.1.0.0*.

Result

This UDF returns detokenized *string* (Unicode) value.

Example

```
select pty_UnicodeStringSel(val, 'Token_unicode') from temp_table;
```

Table 3-66: Supported Protection Methods

Impala UDFs	Supported Protection Methods					
	Tokenization	**Encryption**	**FPE**	**No Encryption**	**Masking**	**Monitoring**
pty_UnicodeStringSel()	• Unicode (Legacy) • Unicode (Base64)	No	No	Yes	No	Yes

Note: The protection methods, that are not mentioned in the *Supported Protection Methods* table, are not supported.

3.7.11 pty_UnicodeStringFPEIns()

This UDF returns the encrypted value for a column containing String (Unicode) format data with Format Preserving Encryption (FPE) as the protection method.

Note: Ensure that you use this UDF with an FPE data element only.

pty_UnicodeStringFPEIns(data string, dataElement string)

Parameters

`data`: Column name of the data to encrypt in the table

`dataElement`: Variable specifying the protection method

Result

`string`: Returns a string value

Example

```
SELECT pty_unicodestringfpeins(column_name,'<DataElement>') from table_name;
```

Table 3-67: Supported Protection Methods

Impala UDFs	Supported Protection Methods					
	Tokenization	**Encryption**	**FPE**	**No Encryption**	**Masking**	**Monitoring**
pty_UnicodeStringFPEIns()	No	No	FPE (All)	Yes	No	Yes

Note: The protection methods, that are not mentioned in the *Supported Protection Methods* table, are not supported.

3.7.12 pty_UnicodeStringFPESel()

This UDF unprotects the existing encrypted String value that was encrypted using the FPE enabled data element.

Note: Ensure that you use this UDF with an FPE data element only.

pty_UnicodeStringFPESel(data string, dataElement string)

Parameters

`data`: Column name of the data to decrypt in the table

`varchar dataElement`: Name of data element to decrypt the encrypted *string* value

> **Note:** Ensure that the FPE data element used to encrypt and decrypt the data is same.

Result

This UDF returns decrypted *string* (Unicode) value.

Example

```
select pty_unicodestringfpesel(NAME,'<DataElement>') from table_name;
```

Table 3-68: Supported Protection Methods

Impala UDFs	Supported Protection Methods					
	Tokenization	Encryption	FPE	No Encryption	Masking	Monitoring
pty_UnicodeStringFPESel()	No	No	FPE (All)	Yes	No	Yes

> **Note:** The protection methods, that are not mentioned in the *Supported Protection Methods* table, are not supported.

3.7.13 pty_IntegerEnc()

This UDF returns the encrypted value for a column containing *Integer* format data.

pty_IntegerEnc(data integer, dataElement string)

Parameters

`data`: Column name of the data to encrypt in the table

`dataElement`: Variable specifying the protection method

Result

Returns a string value.

Example

```
select pty_IntegerEnc(column_name,'enc_3des') from table_name;
```

Table 3-69: Supported Protection Methods

Spark Java APIs	Supported Protection Methods					
	Tokenization	Encryption	FPE	No Encryption	Masking	Monitoring
pty_IntegerEnc()	No	• AES-128 • AES-256 • 3DES • CUSP	No	Yes	No	Yes

> **Note:** The protection methods, that are not mentioned in the *Supported Protection Methods* table, are not supported.

3.7.14 pty_IntegerDec()

This UDF returns the decrypted value for a column containing *Integer* format data.

pty_IntegerDec(data string, dataElement string)

Parameters

`data`: Column name of the data to decode in the table

`dataElement`: Variable specifying the unprotection method

Result

Returns an integer value

Example

```
select pty_IntegerDec(column_name,'enc_3des') from table_name;
```

Table 3-70: Supported Protection Methods

Impala UDFs	Supported Protection Methods					
	Tokenization	**Encryption**	**FPE**	**No Encryption**	**Masking**	**Monitoring**
pty_IntegerDec()	No	• AES-128 • AES-256 • 3DES • CUSP	No	Yes	No	Yes

Note: The protection methods, that are not mentioned in the *Supported Protection Methods* table, are not supported.

3.7.15 pty_IntegerIns()

This UDF returns the tokenized value for a column containing *Integer* format data.

pty_IntegerIns(data integer, dataElement string)

Parameters

`data`: Column name of the data to tokenize in the table

`dataElement`: Variable specifying the protection method

Result

Returns the tokenized integer value

Example

```
select pty_IntegerIns(column_name,'integer_de') from table_name;
```

Table 3-71: Supported Protection Methods

Impala UDFs	Supported Protection Methods					
	Tokenization	**Encryption**	**FPE**	**No Encryption**	**Masking**	**Monitoring**
pty_IntegerIns()	Integer (4 Bytes)	No	No	Yes	No	Yes

Note: The protection methods, that are not mentioned in the *Supported Protection Methods* table, are not supported.

3.7.16 pty_IntegerSel()

This UDF returns the detokenized value for a column containing *Integer* format data.

pty_IntegerSel(data integer, dataElement string)

Parameters

`data`: Column name of the data to detokenize in the table

`dataElement`: Variable specifying the unprotection method

Result

Returns the detokenized integer value

Example

```
select pty_IntegerSel(column_name,'integer_de') from table_name;
```

Table 3-72: Supported Protection Methods

Impala UDFs	Supported Protection Methods					
	Tokenization	Encryption	FPE	No Encryption	Masking	Monitoring
pty_IntegerSel()	Integer (4 Bytes)	No	No	Yes	No	Yes

Note: The protection methods, that are not mentioned in the *Supported Protection Methods* table, are not supported.

3.7.17 pty_FloatEnc()

This UDF returns the encrypted value for a column containing *Float* format data.

pty_FloatEnc(data float, dataElement string)

Parameters

`data`: Column name of the data to encrypt in the table

`dataElement`: Variable specifying the protection method

Result

Returns a string value

Example

```
select pty_FloatEnc(column_name,'enc_3des') from table_name;
```

Table 3-73: Supported Protection Methods

Impala UDFs	Supported Protection Methods					
	Tokenization	Encryption	FPE	No Encryption	Masking	Monitoring
pty_FloatEnc()	No	• AES-128 • AES-256 • 3DES • CUSP	No	Yes	No	Yes

Note: The protection methods, that are not mentioned in the *Supported Protection Methods* table, are not supported.

3.7.18 pty_FloatDec()

This UDF returns the decrypted value for a column containing *Float* format data.

pty_FloatDec(data string, dataElement string)

Parameters

`data`: Column name of the data to decode in the table

`dataElement`: Variable specifying the unprotection method

Result

`float` : Returns a string value

Example

```
select pty_FloatDec(column_name,'enc_3des') from table_name;
```

Table 3-74: Supported Protection Methods

Impala UDFs	Supported Protection Methods					
	Tokenization	**Encryption**	**FPE**	**No Encryption**	**Masking**	**Monitoring**
pty_FloatDec()	No	• AES-128 • AES-256 • 3DES • CUSP	No	Yes	No	Yes

Note: The protection methods, that are not mentioned in the *Supported Protection Methods* table, are not supported.

3.7.19 pty_FloatIns()

This UDF returns the tokenized value for a column containing *Float* format data.

pty_FloatIns(data float, dataElement string)

Parameters

`data`: Column name of the data to tokenize in the table

`dataElement`: Variable specifying the protection method

Result

`float`: Returns the tokenized float value

Example

```
select pty_FloatIns(cast(12.3 as float), 'no_enc');
```

Warning:

Ensure that you use the data element with the *No Encryption* method only. Using any other data element would return an error mentioning that the operation is not supported for that data type.

If you need to tokenize the Float column, then load the Float column into a String column and use the pty_StringIns UDF to tokenize the column.

For more information about the *pty_StringIns* UDF, refer to section *pty_StringIns()*.

Table 3-75: Supported Protection Methods

Impala UDFs	Supported Protection Methods					
	Tokenization	**Encryption**	**FPE**	**No Encryption**	**Masking**	**Monitoring**
pty_FloatIns()	No	No	No	Yes	No	Yes

Note: The protection methods, that are not mentioned in the *Supported Protection Methods* table, are not supported.

3.7.20 pty_FloatSel()

This UDF returns the detokenized value for a column containing *Float* format data.

pty_FloatSel(data float, dataElement string)

Parameters

`data`: Column name of the data to detokenize in the table

`dataElement`: Variable specifying the unprotection method

Result

`float`: Returns the detokenized float value

Example

```
select pty_FloatSel(tokenized_value, 'no_enc');
```

Warning: Ensure that you use the data element with the *No Encryption* method only. Using any other data element would return an error mentioning that the operation is not supported for that data type.

Table 3-76: Supported Protection Methods

Impala UDFs	Supported Protection Methods					
	Tokenization	**Encryption**	**FPE**	**No Encryption**	**Masking**	**Monitoring**
pty_FloatSel()	No	No	No	Yes	No	Yes

Note: The protection methods, that are not mentioned in the *Supported Protection Methods* table, are not supported.

3.7.21 pty_DoubleEnc()

This UDF returns the encrypted value for a column containing *Double* format data.

pty_DoubleEnc(data double, dataElement string)

Parameters

`data`: Integer data column to encrypt in the table

`dataElement`: Variable specifying the protection method

Result

`string`: Returns a string

Example

```
select pty_DoubleEnc(column_name,'enc_3des') from table_name;
```

Table 3-77: Supported Protection Methods

Impala UDFs	Supported Protection Methods					
	Tokenization	**Encryption**	**FPE**	**No Encryption**	**Masking**	**Monitoring**
pty_DoubleEnc()	No	• AES-128 • AES-256 • 3DES • CUSP	No	Yes	No	Yes

Note: The protection methods, that are not mentioned in the *Supported Protection Methods* table, are not supported.

3.7.22 pty_DoubleDec()

This UDF returns the decrypted value for a column containing *Double* format data.

Pty_DoubleDec(data string, dataElement string)

Parameters

`data`: Integer data column to decode in the table

`dataElement`: Variable specifying the unprotection method

Result

`double`: Returns a *double* value

Example

```
select pty_DoubleDec(column_name,'enc_3des') from table_name;
```

Table 3-78: Supported Protection Methods

Impala UDFs	Supported Protection Methods					
	Tokenization	**Encryption**	**FPE**	**No Encryption**	**Masking**	**Monitoring**
pty_DoubleDec()	No	• AES-128 • AES-256 • 3DES • CUSP	No	Yes	No	Yes

Note: The protection methods, that are not mentioned in the *Supported Protection Methods* table, are not supported.

3.7.23 pty_DoubleIns()

This UDF returns the tokenized value for a column containing *Double* format data.

pty_DoubleIns(data double, dataElement string)

Parameters

`data`: Column name of the data to tokenize in the table

`dataElement`: Variable specifying the protection method

Result

`double`: Returns a *double* value

Example

```
select pty_DoubleIns(cast(1.2 as double), 'no_enc');
```

Warning:

Ensure that you use the data element with the *No Encryption* method only. Using any other data element would return an error mentioning that the operation is not supported for that data type.

If you need to tokenize the *Double* column, then load the *Double* column into a String column and use the pty_StringIns UDF to tokenize the column.

For more information about the *pty_StringIns* UDF, refer to section *pty_StringIns()*.

Table 3-79: Supported Protection Methods

Impala UDFs	Supported Protection Methods					
	Tokenization	**Encryption**	**FPE**	**No Encryption**	**Masking**	**Monitoring**
pty_DoubleIns()	No	No	No	Yes	No	Yes

Note: The protection methods, that are not mentioned in the *Supported Protection Methods* table, are not supported.

3.7.24 pty_DoubleSel()

This UDF returns the detokenized value for a column containing *Double* format data.

pty_DoubleSel(data double, dataElement string)

Parameters

`data`: Column name of the data to detokenize in the table

`dataElement`: Variable specifying the unprotection method

Result

`float`: Returns the detokenized double value

Example

```
select pty_DoubleSel(tokenized_value, 'no_enc');
```

Warning: Ensure that you use the data element with the *No Encryption* method only. Using any other data element would return an error mentioning that the operation is not supported for that data type.

Table 3-80: Supported Protection Methods

Impala UDFs	Supported Protection Methods					
	Tokenization	Encryption	FPE	No Encryption	Masking	Monitoring
pty_DoubleSel()	No	No	No	Yes	No	Yes

Note: The protection methods, that are not mentioned in the *Supported Protection Methods* table, are not supported.

3.8 Spark Java APIs

This section describes the Spark APIs (Java) available for protection and unprotection in the Big Data Protector to build secure Big Data applications.

Warning:

The Protegrity Spark protector only supports *bytes* converted from the *string* data type.

If any other data type is directly converted to *bytes* and passed as input to the API that supports *byte* as input and provides *byte* as output, then data corruption might occur.

Caution: If you are using the Protect, or Unprotect, or Reprotect API which accepts byte as input and provides byte as output, then ensure that you encode the string input data type with the corresponding default encoding that is used in the data element.

For example, for Gen2 data element with the default encoding UTF16LE, ensure that you encode the string with the UTF16LE before passing it to the ByteIn or ByteOut APIs.

Note: If you perform a security operation on bulk data, then an exception appears in case of any error except for the error codes 22, 23, and 44. Instead of an error message, the UDFs return an error list for the individual items in the bulk data. For more information about the API error return codes, refer to Table 8-2: PEP Log Return Codes in section 8 Appendix: Return Codes in the Protegrity Big Data Protector Guide 9.1.0.0.

3.8.1 getVersion()

This function returns the current version of the Spark protector.

public string getVersion()

Parameters

None

Result

This function returns the current version of Spark protector.

Example

```
String applicationId = sparkContext.getConf().getAppId();
Protector protector = new PtySparkProtector(applicationId);
String version = protector.getVersion();
```

Exception

`PtySparkProtectorException`: If unable to return the current version of the Spark protector

3.8.2 getCurrentKeyId()

This method returns the current Key ID for the data element, which contains the *KEY ID* attribute, while creating the data element, such as AES-256, AES-128, and so on.

public int getCurrentKeyId(String dataElement)

Parameters

`dataElement` : Name of the data element

Result

This method returns the current Key ID for the data element containing the *KEY ID* attribute.

Example

```
String applicationId = sparkContext.getConf().getAppId();
Protector protector = new PtySparkProtector(applicationId);
int keyId = protector.getCurrentKeyId("AES-256");
```

Exception

`PtySparkProtectorException` : If unable to return the current Key ID for the data element

3.8.3 checkAccess()

This method checks the access of the user for the specified data element.

public boolean checkAccess(String dataElement, Permission permission)

Parameters

`dataElement` : Name of the data element

`Permission` : Type of the access of the user for the data element

Result

`true` : If the user has access to the data element

`false` : If the user does not have access to the data element

Example

```
String applicationId = sparkContext.getConf().getAppId()
Protector protector = new PtySparkProtector(applicationId);
boolean accessType = protector.checkAccess(dataElement, Permission.PROTECT);
```

Exception

`PtySparkProtectorException` : If unable to verify the access of the user for the data element

3.8.4 getDefaultDataElement()

This method returns default data element configured in the security policy.

public String getDefaultDataElement(String policyName)

Parameters

`policyName` : Name of policy configured using Policy management in ESA

Result

Default data element name configured in the security policy.

Example

```
String applicationId = sparkContext.getConf().getAppId();
Protector protector = new PtySparkProtector(applicationId);
String dataElement = protector.getDefaultDataElement("sample_policy");
```

Exception

`PtySparkProtectorException` : If unable to verify the access of the user for the data element

3.8.5 hmac()

> **Note:** It is recommended to use the HMAC data element with the protect() Byte API for hashing byte array data, instead of using the hmac() API.

This method performs hashing of the data using the HMAC operation on a single data item with a data element, which is associated with HMAC. It returns the hmac value of the data with the data element.

public byte[] hmac(String dataElement, byte[] input)

Parameters

`dataElement` : Name of the data element for HMAC

`data` : Byte *array* of data for HMAC

Result

Result Byte array of HMAC data

Example

```
String applicationId = sparkContext.getConf().getAppId()
Protector protector = new PtySparkProtector(applicationId);
byte[] output = protector.hmac("HMAC-SHA1", "test1".getBytes());
```

Exception

`PtySparkProtectorException` : If unable to protect data.

Table 3-81: Supported Protection Methods

Spark Java APIs	Supported Protection Methods					
	Tokenization	Encryption	FPE	No Encryption	Masking	Monitoring
pty_IntegerEnc()	HMAC	No	No	Yes	No	Yes

> **Note:** The protection methods, that are not mentioned in the *Supported Protection Methods* table, are not supported.

3.8.6 protect() - Byte array data

Protects the data provided as a byte *array*. The type of protection applied is defined by *dataElement*.

> **Note:** For Date and Datetime type of data elements, the protect API returns an invalid input data error if the input value falls between the non-existent date range from 05-OCT-1582 to 14-OCT-1582 of the Gregorian Calendar.
>
> For more information about the tokenization and de-tokenization of the cutover dates of the Proleptic Gregorian Calendar, refer to the section *Date* and *Datetime* tokenization in the *Protection Method Reference Guide 8.1.0.0*.

public void protect(String dataElement, List<Integer> errorIndex, byte[][] input, byte[][] output)

Parameters

`dataElement` : Name of the data element used for protection. The *Protect* API also supports the HMAC data element for hashing the byte array data

`errorIndex` : List of the Error Index

`input` : Array of a byte array of data to be protected

`output` : Array of a byte array containing protected data

> **Note:**
> The Protegrity Spark protector only supports *bytes* converted from the *string* data type.
>
> If any other data type is directly converted to *bytes* and passed as input to the API that supports *byte* as input and provides *byte* as output, then data corruption might occur.

> **Warning:** If you are using the *Protect* API which accepts *byte* as input and provides *byte* as output, then ensure that when unprotecting the data, the *Unprotect* API, with *byte* as input and *byte* as output is utilized. In addition, ensure that the *byte* data being provided as input to the Protect API has been converted from a *string* data type only.

Result

The *output* variable in the method signature contains protected data

Example

```
String applicationId = sparkContext.getConf().getAppId();
Protector protector = new PtySparkProtector (applicationId);
String dataElement=”Binary”;
byte[][] input = new byte[][]{“test1”.getbytes(),”test2”.getbytes()};
byte[][] output = new byte[input.length][];
List<Integer> errorIndexList = new ArrayList<Integer>();
protector.protect(dataElement, errorIndexList, input, output);
```

Exception

`PtySparkProtectorException` : If unable to protect data.

Table 3-82: Supported Protection Methods

Spark Java APIs	Supported Protection Methods					
	Tokenization	**Encryption**	**FPE**	**No Encryption**	**Masking**	**Monitoring**
protect() - Byte array data	• Numeric (0-9) • Credit Card • Alpha • Upper Case Alpha	• AES-128 • AES-256 • 3DES • CUSP	FPE (All - Encoded Byte's Charset should match Dataelement's Encoding Type)	Yes	Yes	Yes

Spark Java APIs	Supported Protection Methods					
	Tokenization	Encryption	FPE	No Encryption	Masking	Monitoring
	• Alpha Numeric • Upper Alpha Numeric • Lower ASCII • Printable • Datetime (YYYY-MM-DD HH:MM:SS) • Date (YYYY-MM-DD, DD/MM/YYYY, MM.DD.YYYY) • Decimal • Email • Binary • Unicode (Legacy) • Unicode (Base64 - Encoded Byte's Charset should match Dataelement's Encoding Type) • Unicode (Gen2)					

Note: The protection methods, that are not mentioned in the *Supported Protection Methods* table, are not supported.

3.8.7 protect() - Short array data

Protects the short format data provided as a short *array*. The type of protection applied is defined by *dataElement*.

public void protect(String dataElement, List<Integer> errorIndex, short[] input, short[] output)

Parameters

`dataElement` : Name of the data element used for protection

`errorIndex` : List of the Error Index

`input` : Short *array* of data to be protected

`output` : Short *array* containing protected data

Result

The *output* variable in the method signature contains protected data

Example

```
String applicationId = sparkContext.getConf().getAppId();
Protector protector = new PtySparkProtector (applicationId);
String dataElement=”short”;
short[] input = new short[] {1234, 4545};
short[] output = new short[input.length];
List<Integer> errorIndexList = new ArrayList<Integer>();
protector.protect(dataElement, errorIndexList, input, output);
```

Exception

`PtySparkProtectorException` :If unable to protect data

Table 3-83: Supported Protection Methods

Spark Java APIs	Supported Protection Methods					
	Tokenization	**Encryption**	**FPE**	**No Encryption**	**Masking**	**Monitoring**
protect() - Short array data	Integer (2 Bytes)	No	No	Yes	No	Yes

Note: The protection methods, that are not mentioned in the *Supported Protection Methods* table, are not supported.

3.8.8 protect() - Short array data for encryption

Encrypts the short format data provided as a short *array*. The type of encryption applied is defined by *dataElement*.

public void protect(String dataElement, List<Integer> errorIndex, short[] input, byte[][] output)

Parameters

`dataElement` : Name of the data element used for encryption

`errorIndex` : List of the Error Index

`input` : Short *array* of data to be encrypted

`output` : Array of an encrypted byte *array*

Result

The *output* variable in the method signature contains protected data

Example

```
String applicationId = sparkContext.getConf().getAppId();
Protector protector = new PtySparkProtector (applicationId);
String dataElement= "AES-256";
short[] input = new short[] {1234, 4545};
byte[][] output = new byte[input.length][];
List<Integer> errorIndexList = new ArrayList<Integer>();
protector.protect(dataElement, errorIndexList, input, output);
```

Exception

`PtySparkProtectorException` :If unable to encrypt data

Table 3-84: Supported Protection Methods

Spark Java APIs	Supported Protection Methods					
	Tokenization	**Encryption**	**FPE**	**No Encryption**	**Masking**	**Monitoring**
protect() - Short array data for encryption	No	• AES-128 • AES-256 • 3DES • CUSP	No	Yes	No	Yes

Note: The protection methods, that are not mentioned in the *Supported Protection Methods* table, are not supported.

3.8.9 protect() - Int array

Protects the data provided as *int* array. The type of protection applied is defined by *dataElement*.

public void protect(String dataElement, List<Integer> errorIndex, int[] input, int[] output)

Parameters

`dataElement`: Name of the data element used for protection

`errorIndex`: List of the Error Index

`input`: Int *array* of data to be protected

`output`: Int *array* containing protected data

Result

The *output* variable in the method signature contains protected *int* data

Example

```
String applicationId = sparkContext.getConf().getAppId();
Protector protector = new PtySparkProtector (applicationId);
String dataElement = "int";
int[] input = new int[]{1234, 4545};
int[] output = new int[input.length];
List<Integer> errorIndexList = new ArrayList<Integer>();
protector.protect(dataElement, errorIndexList, input, output);
```

Exception

`PtySparkProtectorException` :If unable to protect data

Table 3-85: Supported Protection Methods

Spark Java APIs	Supported Protection Methods					
	Tokenization	**Encryption**	**FPE**	**No Encryption**	**Masking**	**Monitoring**
protect() - Int array	Integer (4 Bytes)	No	No	Yes	No	Yes

Note: The protection methods, that are not mentioned in the *Supported Protection Methods* table, are not supported.

3.8.10 protect() - Int array data for encryption

Encrypts the data provided as *int* array. The type of encryption applied is defined by *dataElement*.

public void protect(String dataElement, List<Integer> errorIndex, int[] input, byte[][] output)

Parameters

`dataElement`: Name of the data element used for encryption

`errorIndex`: List of the Error Index

`input`: Int *array* of data to be encrypted

`output`: Array of an encrypted byte *array*

Result

The *output* variable in the method signature contains encrypted data

Example

```
String applicationId = sparkContext.getConf().getAppId();
Protector protector = new PtySparkProtector (applicationId);
String dataElement = "AES-256";
int[] input = new int[]{1234, 4545};
byte[][] output = new byte[input.length][];
List<Integer> errorIndexList = new ArrayList<Integer>();
protector.protect(dataElement, errorIndexList, input, output);
```

Exception

`PtySparkProtectorException` : If unable to encrypt data

Table 3-86: Supported Protection Methods

Spark Java APIs	Supported Protection Methods					
	Tokenization	**Encryption**	**FPE**	**No Encryption**	**Masking**	**Monitoring**
protect() - Int array data for encryption	No	• AES-128 • AES-256	No	Yes	No	Yes

Spark Java APIs	Supported Protection Methods					
	Tokenization	Encryption	FPE	No Encryption	Masking	Monitoring
		• 3DES • CUSP				

Note: The protection methods, that are not mentioned in the *Supported Protection Methods* table, are not supported.

3.8.11 protect() - Long array data

Protects the data provided as *long* byte array. The type of protection applied is defined by *dataElement*.

public void protect(String dataElement, List<Integer> errorIndex, long[] input, long[] output)

Parameters

`dataElement`: Name of the data element used for protection

`errorIndex`: List of the Error Index

`input`: Long *array* of data to be protected

`output`: Long *array* containing protected data

Result

The *output* variable in the method signature contains protected data

Example

```
String applicationId = sparkContext.getConf().getAppId();
Protector protector = new PtySparkProtector (applicationId);
String dataElement = "long";
long[] input = new long[] {1234, 4545};
long[] output = new long[input.length];
List<Integer> errorIndexList = new ArrayList<Integer>();
protector.protect(dataElement, errorIndexList, input, output);
```

Exception

`PtySparkProtectorException` : If unable to protect data

Table 3-87: Supported Protection Methods

Spark Java APIs	Supported Protection Methods					
	Tokenization	Encryption	FPE	No Encryption	Masking	Monitoring
protect() - Long array data	Integer (8 Bytes)	No	No	Yes	No	Yes

Note: The protection methods, that are not mentioned in the *Supported Protection Methods* table, are not supported.

3.8.12 protect() - Long array data for encryption

Encrypts the data provided as *long* byte array. The type of encryption applied is defined by *dataElement*.

public void protect(String dataElement, List<Integer> errorIndex, long[] input, byte[][] output)

Parameters

`dataElement` : Name of the data element used for encryption

`errorIndex` : List of the Error Index

`input` : Long *array* of data to be encrypted

`output` : Array of a byte *array* containing encrypted data

Result

The *output* variable in the method signature contains encrypted data

Example

```
String applicationId = sparkContext.getConf().getAppId();
Protector protector = new PtySparkProtector (applicationId);
String dataElement = "AES-256";
short[] input = new short[]{1234, 4545};
byte[][] output = new byte[input.length][];
List<Integer> errorIndexList = new ArrayList<Integer>();
protector.protect(dataElement, errorIndexList, input, output);
```

Exception

`PtySparkProtectorException` :If unable to encrypt data

Table 3-88: Supported Protection Methods

Spark Java APIs	Supported Protection Methods					
	Tokenization	**Encryption**	**FPE**	**No Encryption**	**Masking**	**Monitoring**
protect() - Long array data for encryption	No	• AES-128 • AES-256 • 3DES • CUSP	No	Yes	No	Yes

Note: The protection methods, that are not mentioned in the *Supported Protection Methods* table, are not supported.

3.8.13 protect() - Float array data

Protects the data provided as *float* array. The type of protection applied is defined by *dataElement*.

public void protect(String dataElement, List<Integer> errorIndex, float[] input, float[] output)

Parameters

`dataElement`: Name of the data element used for protection

`errorIndex`: List of the Error Index

`input`: Float *array* of data to be protected

`output`: Float *array* containing protected data

Warning: Ensure that you use the data element with the *No Encryption* method only. Using any other data element might cause corruption of data.

Result

The *output* variable in the method signature contains protected *float* data

Example

```
String applicationId = sparkContext.getConf().getAppId();
Protector protector = new PtySparkProtector (applicationId);
String dataElement = "float";
float[] input = new float[] {123.4f, 454.5f};
float[] output = new float[input.length];
List<Integer> errorIndexList = new ArrayList<Integer>();
protector.protect(dataElement, errorIndexList, input, output);
```

Exception

`PtySparkProtectorException` : If unable to protect data

Table 3-89: Supported Protection Methods

Spark Java APIs	Supported Protection Methods					
	Tokenization	**Encryption**	**FPE**	**No Encryption**	**Masking**	**Monitoring**
protect() - Float array data	No	No	No	Yes	No	Yes

Note: The protection methods, that are not mentioned in the *Supported Protection Methods* table, are not supported.

3.8.14 protect() - Float array data for encryption

Encrypts the data provided as *float* array. The type of encryption applied is defined by *dataElement*.

public void protect(String dataElement, List<Integer> errorIndex, float[] input, byte[][] output)

Parameters

`dataElement`: Name of the data element used for encryption

`errorIndex`: List of the Error Index

`input`: Float *array* of data to be encrypted

`output`: Float *array* containing encrypted data

Warning: Ensure that you use the data element with either the *No Encryption* method or *Encryption* data element only. Using any other data element might cause corruption of data.

Result

The *output* variable in the method signature contains encrypted data

Example

```
String applicationId = sparkContext.getConf().getAppId();
Protector protector = new PtySparkProtector (applicationId);
String dataElement = "AES-256";
float[] input = new float[] {123.4f, 454.5f};
byte[][] output = new byte[input.length][];
List<Integer> errorIndexList = new ArrayList<Integer>();
protector.protect(dataElement, errorIndexList, input, output);
```

Exception

`PtySparkProtectorException` : If unable to encrypt data

Table 3-90: Supported Protection Methods

Spark Java APIs	Supported Protection Methods					
	Tokenization	**Encryption**	**FPE**	**No Encryption**	**Masking**	**Monitoring**
protect() - Float array data for encryption	No	• AES-128 • AES-256 • 3DES • CUSP	No	Yes	No	Yes

Note: The protection methods, that are not mentioned in the *Supported Protection Methods* table, are not supported.

3.8.15 protect() - Double array data

Protects the data provided as *double* array. The type of protection applied is defined by *dataElement*.

public void protect(String dataElement, List<Integer> errorIndex, double[] input, double[] output)

Parameters

`dataElement`: Name of the data element used for protection

`errorIndex`: List of the Error Index

`input`: Double *array* of data to be protected

`output`: Double *array* containing protected data

> **Warning:** Ensure that you use the data element with the *No Encryption* method only. Using any other data element might cause corruption of data.

Result

The *output* variable in the method signature contains protected *double* data

Example

```
String applicationId = sparkContext.getConf().getAppId();
Protector protector = new PtySparkProtector (applicationId);
String dataElement = "double";
double[] input = new double[] {123.4, 454.5};
double[] output = new double[input.length];
List<Integer> errorIndexList = new ArrayList<Integer>();
protector.protect(dataElement, errorIndexList, input, output);
```

Exception

`PtySparkProtectorException` : If unable to protect data

Table 3-91: Supported Protection Methods

Spark Java APIs	Supported Protection Methods					
	Tokenization	Encryption	FPE	No Encryption	Masking	Monitoring
protect() - Double array data	No	No	No	Yes	No	Yes

> **Note:** The protection methods, that are not mentioned in the *Supported Protection Methods* table, are not supported.

3.8.16 protect() - Double array data for encryption

Encrypts the data provided as *double* array. The type of encryption applied is defined by *dataElement*.

public void protect(String dataElement, List<Integer> errorIndex, double[] input, byte[][] output)

Parameters

`dataElement`: Name of the data element used for encryption

`errorIndex`: List of the Error Index

`input`: Double *array* of data to be encrypted

`output`: Double *array* containing encryption data

> **Warning:** Ensure that you use the data element with either the *No Encryption* method or *Encryption* data element only. Using any other data element might cause corruption of data.

Result

The *output* variable in the method signature contains encrypted data

Example

```
String applicationId = sparkContext.getConf().getAppId();
Protector protector = new PtySparkProtector (applicationId);
String dataElement = "AES-256";
double[] input = new double[] {123.4, 454.5};
```

```
byte[][] output = new byte[input.length][];
List<Integer> errorIndexList = new ArrayList<Integer>();
protector.protect(dataElement, errorIndexList, input, output);
```

Exception

`PtySparkProtectorException` : If unable to encrypt data

Table 3-92: Supported Protection Methods

Spark Java APIs	Supported Protection Methods					
	Tokenization	**Encryption**	**FPE**	**No Encryption**	**Masking**	**Monitoring**
protect() - Double array data for encryption	No	• AES-128 • AES-256 • 3DES • CUSP	No	Yes	No	Yes

Note: The protection methods, that are not mentioned in the *Supported Protection Methods* table, are not supported.

3.8.17 protect() - String array data

Protects the data provided as *string* array. The type of protection applied is defined by *dataElement*.

Note: For Date and Datetime type of data elements, the protect API returns an invalid input data error if the input value falls between the non-existent date range from 05-OCT-1582 to 14-OCT-1582 of the Gregorian Calendar.

For more information about the tokenization and de-tokenization of the cutover dates of the Proleptic Gregorian Calendar, refer to the section *Date* and *Datetime* tokenization in the *Protection Method Reference Guide 8.1.0.0*.

public void protect(String dataElement, List<Integer> errorIndex, String[] input, String[] output)

Parameters

`dataElement`: Name of the data element used for protection

`errorIndex`: List of the Error Index

`input`: String *array* of data to be protected

`output`: String *array* containing protection data

Result

The *output* variable in the method signature contains protected *string* data

Example

```
String applicationId = sparkContext.getConf().getAppId();
Protector protector = new PtySparkProtector (applicationId);
String dataElement = "AlphaNum";
String[] input = new String[] {"test1", "test2"};
String[] output = new String[input.length];
List<Integer> errorIndexList = new ArrayList<Integer>();
protector.protect(dataElement, errorIndexList, input, output);
```

Exception

`PtySparkProtectorException` : If unable to encrypt data

Table 3-93: Supported Protection Methods

Spark Java APIs	Supported Protection Methods					
	Tokenization	**Encryption**	**FPE**	**No Encryption**	**Masking**	**Monitoring**
protect() - String array data	• Numeric (0-9)	No	FPE(All)	Yes	Yes	Yes

Spark Java APIs	Supported Protection Methods					
	Tokenization	Encryption	FPE	No Encryption	Masking	Monitoring
	• Credit Card • Alpha (A-Z) • Upper Case Alpha (A-Z) • Alpha Numeric (0-9, a-z, A-Z) • Upper Alpha Numeric (0-9, A-Z) • Lower ASCII • Printable • Datetime (YYYY-MM-DD HH:MM:SS) • Date (YYYY-MM-DD, DD/MM/YYYY, MM.DD.YYYY) • Decimal • Email • Binary • Unicode (Legacy) • Unicode (Base64 - Encoded Byte's Charset should match Dataelement's Encoding Type) • Unicode (Gen2)					

Note: The protection methods, that are not mentioned in the *Supported Protection Methods* table, are not supported.

3.8.18 protect() - String array data for encryption

Encrypts the data provided as *string* array. The type of encryption applied is defined by *dataElement*.

public void protect(String dataElement, List<Integer> errorIndex, String[] input, byte[][] output)

Parameters

`dataElement`: Name of the data element used for encrypted

`errorIndex`: List of the Error Index

`input`: String *array* of data to be encrypted

`output`: Array of a byte *array* containing encrypted data

Result

The *output* variable in the method signature contains encrypted data

Example

```
String applicationId = sparkContext.getConf().getAppId();
Protector protector = new PtySparkProtector (applicationId);
String dataElement = "AES-256";
```

```
String[] input = new String[] {"test1", "test2"};
byte[][] output = new byte[input.length][];
List<Integer> errorIndexList = new ArrayList<Integer>();
protector.protect(dataElement, errorIndexList, input, output);
```

Exception

`PtySparkProtectorException` : If unable to encrypt data

Table 3-94: Supported Protection Methods

Spark Java APIs	Supported Protection Methods					
	Tokenization	**Encryption**	**FPE**	**No Encryption**	**Masking**	**Monitoring**
protect() - String array data for encryption	No	• AES-128 • AES-256 • 3DES • CUSP	No	Yes	Yes	Yes

Note: The protection methods, that are not mentioned in the *Supported Protection Methods* table, are not supported.

3.8.19 unprotect() - Byte array data

Unprotects the protected data provided as a byte *array*. The type of unprotection applied is defined by *dataElement*.

Note: For Date and Datetime type of data elements, the protect API returns an invalid input data error if the input value falls between the non-existent date range from 05-OCT-1582 to 14-OCT-1582 of the Gregorian Calendar.

For more information about the tokenization and de-tokenization of the cutover dates of the Proleptic Gregorian Calendar, refer to the section *Date* and *Datetime* tokenization in the *Protection Method Reference Guide 8.1.0.0*.

public void unprotect(String dataElement, List<Integer> errorIndex, byte[][] input, byte[][] output)

Parameters

`dataElement`: Name of the data element used for unprotection

`errorIndex`: List of the Error Index

`input`: Array of a byte *array* of data to be unprotected

`output`: Array of a byte *array* containing unprotected data

Warning: The Protegrity Spark protector only supports *bytes* converted from the *string* data type.

If any other data type is directly converted to *bytes* and passed as input to the API that supports *byte* as input and provides *byte* as output, then data corruption might occur.

Result

The *output* variable in the method signature unprotected data

Example

```
String applicationId = sparkContext.getConf().getAppId();
Protector protector = new PtySparkProtector (applicationId);
String dataElement = "Binary";
byte[][] input = new byte[][] {“test1”.getbytes(), ”test2”.getbytes()};
byte[][] output = new byte[input.length][];
List<Integer> errorIndexList = new ArrayList<Integer>();
protector.unprotect(dataElement, errorIndexList, input, output);
```

Exception

`PtySparkProtectorException` : If unable to unprotected data

Table 3-95: Supported Protection Methods

Spark Java APIs	Supported Protection Methods					
	Tokenization	**Encryption**	**FPE**	**No Encryption**	**Masking**	**Monitoring**
unprotect() - Byte array data	• Numeric (0-9) • Credit Card • Alpha (A-Z) • Upper-case Alpha (A-Z) • Alpha-Numeric (0-9, a-z, A-Z) • Upper Alpha-Numeric (0-9, A-Z) • Lower ASCII • Printable • Date (YYYY-MM-DD, DD/MM/YYYY, MM.DD.YYYY) • Datetime (YYYY-MM-DD HH:MM:SS) • Decimal • Unicode (Gen2) • Unicode (Legacy) • Unicode (Base64 - Encoded Byte's Charset should match Dataelement's Encoding Type) • Binary • Email	• AES-128 • AES-256 • 3DES • CUSP	FPE (All - Encoded Byte's Charset should match Dataelement's Encoding Type)	Yes	Yes	Yes

Note: The protection methods, that are not mentioned in the *Supported Protection Methods* table, are not supported.

3.8.20 unprotect() - Short array data

Unprotects the protected short format data provided as a short *array*. The type of unprotection applied is defined by *dataElement*.

public void unprotect(String dataElement, List<Integer> errorIndex, short[] input, short[] output)

Parameters

`dataElement`: Name of the data element used for unprotection

`errorIndex`: List of the Error Index

`input`: Short *array* of data to be unprotected

`output`: Short *array* containing unprotected data

Result

The *output* variable in the method signature contains unprotected data

Example

```
String applicationId = sparkContext.getConf().getAppId();
Protector protector = new PtySparkProtector (applicationId);
String dataElement = "short";
short[] input = new short[]{1234, 4545};
short[] output = new short[input.length];
List<Integer> errorIndexList = new ArrayList<Integer>();
protector.unprotect(dataElement, errorIndexList, input, output);
```

Exception

`PtySparkProtectorException` : If unable to unprotected data

Table 3-96: Supported Protection Methods

Spark Java APIs	Supported Protection Methods					
	Tokenization	**Encryption**	**FPE**	**No Encryption**	**Masking**	**Monitoring**
unprotect() - Short array data	Integer (2 Bytes)	No	No	Yes	No	Yes

Note: The protection methods, that are not mentioned in the *Supported Protection Methods* table, are not supported.

3.8.21 unprotect() - Short array data for decryption

Decrypts the encrypted short format data provided as a byte *array*. The type of decryption applied is defined by *dataElement*.

public void unprotect(String dataElement, List<Integer> errorIndex, byte[][] input, short[] output)

Parameters

`dataElement`: Name of the data element used for decryption

`errorIndex`: List of the Error Index

`input`: Array of a byte *array* containing encrypted data

`output`: Short *array* containing decrypted data

Result

The *output* variable in the method signature contains decrypted data

Example

```
String applicationId = sparkContext.getConf().getAppId();
Protector protector = new PtySparkProtector (applicationId);
String dataElement = "AES-256";
// here input is encrypted short array created using our below API
// public void protect(String dataElement, List<Integer> errorIndex, short[] input,
byte[][] output) throws PtySparkProtectorException;
byte[][] input = { <encrypted short array> }
short[] output = new short[input.length];
List<Integer> errorIndexList = new ArrayList<Integer>();
protector.unprotect(dataElement, errorIndexList, input, output);
```

Exception

`PtySparkProtectorException` : If unable to decrypt data

Table 3-97: Supported Protection Methods

Spark Java APIs	Supported Protection Methods					
	Tokenization	**Encryption**	**FPE**	**No Encryption**	**Masking**	**Monitoring**
unprotect() - Short array data for decryption	No	• AES-128 • AES-256 • 3DES	No	Yes	No	Yes

Spark Java APIs	Supported Protection Methods					
	Tokenization	Encryption	FPE	No Encryption	Masking	Monitoring
		• CUSP				

Note: The protection methods, that are not mentioned in the *Supported Protection Methods* table, are not supported.

3.8.22 unprotect() - Int array data

Unprotects the protected data provided as *int* array. The type of unprotection applied is defined by *dataElement*.

public void unprotect(String dataElement, List<Integer> errorIndex, int[] input, int[] output)

Parameters

`dataElement`: Name of the data element used for unprotection

`errorIndex`: List of the Error Index

`input`: Int *array* of data to be unprotected

`output`: Int *array* containing unprotected data

Result

The *output* variable in the method signature contains unprotected data

Example

```
String applicationId = sparkContext.getConf().getAppId();
Protector protector = new PtySparkProtector (applicationId);
String dataElement = "int";
int[] input = new int[]{1234, 4545};
int[] output = new int[input.length];
List<Integer> errorIndexList = new ArrayList<Integer>();
protector.unprotect(dataElement, errorIndexList, input, output);
```

Exception

`PtySparkProtectorException` : If unable to unprotect data

Table 3-98: Supported Protection Methods

Spark Java APIs	Supported Protection Methods					
	Tokenization	Encryption	FPE	No Encryption	Masking	Monitoring
unprotect() - Int array data	Integer (4 Bytes)	No	No	Yes	No	Yes

Note: The protection methods, that are not mentioned in the *Supported Protection Methods* table, are not supported.

3.8.23 unprotect() - Int array data for encryption

Decrypts the encrypted int format data provided as byte array. The type of decryption applied is defined by *dataElement*.

public void unprotect(String dataElement, List<Integer> errorIndex, byte[][] input, int[] output)

Parameters

`dataElement`: Name of the data element used for decryption

`errorIndex`: List of the Error Index

`input`: Array of a byte *array* containing encrypted data

`output`: Int *array* containing decrypted data

Result

The *output* variable in the method signature contains decrypted data

Example

```
String applicationId = sparkContext.getConf().getAppId();
Protector protector = new PtySparkProtector (applicationId);
String dataElement = "AES-256";
// here input is encrypted int array created using our below API
// public void protect(String dataElement, List<Integer> errorIndex, int[] input, byte[]
[] output) throws PtySparkProtectorException;
byte[][] input = {<encrypted int array>};
int[] output = new int[input.length];
List<Integer> errorIndexList = new ArrayList<Integer>();
protector.unprotect(dataElement, errorIndexList, input, output);
```

Exception

`PtySparkProtectorException` : If unable to decrypt data

Table 3-99: Supported Protection Methods

Spark Java APIs	Supported Protection Methods					
	Tokenization	**Encryption**	**FPE**	**No Encryption**	**Masking**	**Monitoring**
unprotect() - Int array data for encryption	No	• AES-128 • AES-256 • 3DES • CUSP	No	Yes	No	Yes

Note: The protection methods, that are not mentioned in the *Supported Protection Methods* table, are not supported.

3.8.24 unprotect() - Long array data

Unprotects the protected data provided as *long* array. The type of unprotection applied is defined by *dataElement*.

public void unprotect(String dataElement, List<Integer> errorIndex, long[] input, long[] output)

Parameters

`dataElement`: Name of the data element used for unprotection

`errorIndex`: List of the Error Index

`input`: Long *array* of data to be unprotected

`output`: Long *array* containing unprotected data

Result

The *output* variable in the method signature contains unprotected data

Example

```
String applicationId = sparkContext.getConf().getAppId();
Protector protector = new PtySparkProtector (applicationId);
String dataElement = "long";
long[] input = new long[] {1234, 4545};
long[] output = new long[input.length];
List<Integer> errorIndexList = new ArrayList<Integer>();
protector.unprotect(dataElement, errorIndexList, input, output);
```

Exception

`PtySparkProtectorException` : If unable to unprotect data

Table 3-100: Supported Protection Methods

Spark Java APIs	Supported Protection Methods					
	Tokenization	**Encryption**	**FPE**	**No Encryption**	**Masking**	**Monitoring**
unprotect() - Long array data	Integer (8 Bytes)	No	No	Yes	No	Yes

Note: The protection methods, that are not mentioned in the *Supported Protection Methods* table, are not supported.

3.8.25 unprotect() - Long array data for decryption

Decrypts the encrypted long format data provided as byte array. The type of decryption applied is defined by *dataElement*.

public void unprotect(String dataElement, List<Integer> errorIndex, byte[][] input, long[] output)

Parameters

`dataElement`: Name of the data element used for decryption

`errorIndex`: List of the Error Index

`input`: Array of a byte *array* containing encrypted data

`output`: Long *array* containing decrypted data

Result

The *output* variable in the method signature contains decrypted data

Example

```
String applicationId = sparkContext.getConf().getAppId();
Protector protector = new PtySparkProtector (applicationId);
String dataElement = "AES-256";
// here input is encrypted long array created using our below API
// public void protect(String dataElement, List<Integer> errorIndex, long[] input,
byte[][] output) throws PtySparkProtectorException;
byte[][] input = { <encrypted long array> };
long[] output = new long[input.length];
List<Integer> errorIndexList = new ArrayList<Integer>();
protector.unprotect(dataElement, errorIndexList, input, output);
```

Exception

`PtySparkProtectorException` : If unable to unprotect data

Table 3-101: Supported Protection Methods

Spark Java APIs	Supported Protection Methods					
	Tokenization	**Encryption**	**FPE**	**No Encryption**	**Masking**	**Monitoring**
unprotect() - Long array data for decryption	No	• AES-128 • AES-256 • 3DES • CUSP	No	Yes	No	Yes

Note: The protection methods, that are not mentioned in the *Supported Protection Methods* table, are not supported.

3.8.26 unprotect() - Float array data

Unprotects the protected data provided as *float* array. The type of unprotection applied is defined by *dataElement*.

public void unprotect(String dataElement, List<Integer> errorIndex, float[] input, float[] output)

Parameters

`dataElement`: Name of the data element used for unprotection

`errorIndex`: List of the Error Index

`input`: Float *array* of data to be unprotected

`output`: Float *array* containing unprotected data

Result

The *output* variable in the method signature contains unprotected data

> **Warning:** Ensure that you use the data element with the *No Encryption* method only. Using any other data element might cause corruption of data.

Example

```
String applicationId = sparkContext.getConf().getAppId();
Protector protector = new PtySparkProtector (applicationId);
String dataElement = "float";
float[] input = new float[] {123.4f, 454.5f};
float[] output = new float[input.length];
List<Integer> errorIndexList = new ArrayList<Integer>();
protector.unprotect(dataElement, errorIndexList, input, output);
```

Exception

`PtySparkProtectorException` : If unable to unprotect data

Table 3-102: Supported Protection Methods

Spark Java APIs	Supported Protection Methods					
	Tokenization	Encryption	FPE	No Encryption	Masking	Monitoring
unprotect() - Float array data	No	No	No	Yes	No	Yes

> **Note:** The protection methods, that are not mentioned in the *Supported Protection Methods* table, are not supported.

3.8.27 unprotect() - Float array data for decryption

Decrypts the encrypted float format data provided as byte array. The type of decryption applied is defined by *dataElement*.

public void unprotect(String dataElement, List<Integer> errorIndex, byte[][] input, float[] output)

Parameters

`dataElement`: Name of the data element used for decryption

`errorIndex`: List of the Error Index

`input`: Array of a byte *array* containing encrypted data

`output`: Float *array* containing decrypted data

> **Warning:** Ensure that you use the data element with either the *No Encryption* method or *Encryption* data element only. Using any other data element might cause corruption of data.

Result

The *output* variable in the method signature contains decrypted data

Example

```
String applicationId = sparkContext.getConf().getAppId();
Protector protector = new PtySparkProtector (applicationId);
String dataElement = "AES-256";
// here input is encrypted float array created using our below API
// public void protect(String dataElement, List<Integer> errorIndex, float[] input,
byte[][] output) throws PtySparkProtectorException;
```

```
byte[][] input = { <encrypted float array> };
float[] output = new float[input.length][];
List<Integer> errorIndexList = new ArrayList<Integer>();
protector.unprotect(dataElement, errorIndexList, input, output);
```

Exception

`PtySparkProtectorException` : If unable to decrypt data

Table 3-103: Supported Protection Methods

Spark Java APIs	Supported Protection Methods					
	Tokenization	**Encryption**	**FPE**	**No Encryption**	**Masking**	**Monitoring**
unprotect() - Float array data for decryption	No	• AES-128 • AES-256 • 3DES • CUSP	No	Yes	No	Yes

Note: The protection methods, that are not mentioned in the *Supported Protection Methods* table, are not supported.

3.8.28 unprotect() - Double array data

Unprotects the protected data provided as *double* array. The type of unprotection applied is defined by *dataElement*.

public void unprotect(String dataElement, List<Integer> errorIndex, double[] input, double[] output)

Parameters

`dataElement`: Name of the data element used for unprotection

`errorIndex`: List of the Error Index

`input`: Double *array* of data to be unprotected

`output`: Double *array* containing unprotected data

Warning: Ensure that you use the data element with the *No Encryption* method only. Using any other data element might cause corruption of data.

Result

The *output* variable in the method signature contains unprotected data

Example

```
String applicationId = sparkContext.getConf().getAppId();
Protector protector = new PtySparkProtector (applicationId);
String dataElement = "double";
double[] input = new double[] {123.4, 454.5};
double[] output = new double[input.length];
List<Integer> errorIndexList = new ArrayList<Integer>();
protector.unprotect(dataElement, errorIndexList, input, output);
```

Exception

`PtySparkProtectorException` : If unable to unprotect data

Table 3-104: Supported Protection Methods

Spark Java APIs	Supported Protection Methods					
	Tokenization	**Encryption**	**FPE**	**No Encryption**	**Masking**	**Monitoring**
unprotect() - Double array data	No	No	No	Yes	No	Yes

Note: The protection methods, that are not mentioned in the *Supported Protection Methods* table, are not supported.

3.8.29 unprotect() - Double array data for decryption

Decrypts the encrypted double format data provided as byte array. The type of decryption applied is defined by *dataElement*.

public void unprotect(String dataElement, List<Integer> errorIndex, byte[][] input, double[] output)

Parameters

`dataElement`: Name of the data element used for decryption

`errorIndex`: List of the Error Index

`input`: Array of a byte *array* containing encrypted data

`output`: Double *array* containing decrypted data

Warning: Ensure that you use the data element with either the *No Encryption* method or *Encryption* data element only. Using any other data element might cause corruption of data.

Result

The *output* variable in the method signature contains decrypted data

Example

```
String applicationId = sparkContext.getConf().getAppId();
Protector protector = new PtySparkProtector (applicationId);
String dataElement = "AES-256";
// here input is encrypted double array created using our below API
// public void protect(String dataElement, List<Integer> errorIndex, double[] input,
byte[][] output) throws PtySparkProtectorException;
byte[][] input = { <encrypted double array> };
double[] output = new double[input.length];
List<Integer> errorIndexList = new ArrayList<Integer>();
protector.unprotect(dataElement, errorIndexList, input, output);
```

Exception

`PtySparkProtectorException` : If unable to unprotect data

Table 3-105: Supported Protection Methods

Spark Java APIs	Supported Protection Methods					
	Tokenization	**Encryption**	**FPE**	**No Encryption**	**Masking**	**Monitoring**
unprotect() - Double array data for decryption	No	• AES-128 • AES-256 • 3DES • CUSP	No	Yes	No	Yes

Note: The protection methods, that are not mentioned in the *Supported Protection Methods* table, are not supported.

3.8.30 unprotect() - String array data

Unprotects the protected data provided as *string* array. The type of unprotection applied is defined by *dataElement*.

> **Note:** For Date and Datetime type of data elements, the protect API returns an invalid input data error if the input value falls between the non-existent date range from 05-OCT-1582 to 14-OCT-1582 of the Gregorian Calendar.
>
> For more information about the tokenization and de-tokenization of the cutover dates of the Proleptic Gregorian Calendar, refer to the section *Date* and *Datetime* tokenization in the *Protection Method Reference Guide 8.1.0.0*.

public void unprotect(String dataElement, List<Integer> errorIndex, String[] input, String[] output)

Parameters

`dataElement`: Name of the data element used for unprotection

`errorIndex`: List of the Error Index

`input`: String *array* of data to be unprotected

`output`: String *array* containing unprotected data

Result

The *output* variable in the method signature contains unprotected data

Example

```
String applicationId = sparkContext.getConf().getAppId();
Protector protector = new PtySparkProtector (applicationId);
String dataElement = "AlphaNum";
String[] input = new String[] {"test1", "test2"};
String[] output = new String[input.length];
List<Integer> errorIndexList = new ArrayList<Integer>();
protector.unprotect(dataElement, errorIndexList, input, output);
```

Exception

`PtySparkProtectorException` : If unable to unprotect data

Table 3-106: Supported Protection Methods

Spark Java APIs	Supported Protection Methods					
	Tokenization	**Encryption**	**FPE**	**No Encryption**	**Masking**	**Monitoring**
unprotect() - String array data	• Numeric (0-9) • Credit Card • Alpha (A-Z) • Upper-case Alpha (A-Z) • Alpha-Numeric (0-9, a-z, A-Z) • Upper Alpha-Numeric (0-9, A-Z) • Lower ASCII • Printable • Date (YYYY-MM-DD, DD/MM/YYYY, MM.DD.YYYY) • Datetime (YYYY-MM-DD HH:MM:SS)	No	FPE(All)	Yes	Yes	Yes

Spark Java APIs	Supported Protection Methods					
	Tokenization	Encryption	FPE	No Encryption	Masking	Monitoring
	• Decimal • Unicode (Gen2) • Unicode (Legacy) • Unicode (Base64 - Encoded Byte's Charset should match Dataelement's Encoding Type) • Binary • Email					

Note: The protection methods, that are not mentioned in the *Supported Protection Methods* table, are not supported.

3.8.31 unprotect() - String array data for decryption

Decrypts the encrypted string format data provided as byte array. The type of decryption applied is defined by *dataElement*.

public void unprotect(String dataElement, List<Integer> errorIndex, byte[][] input, String[] output)

Parameters

`dataElement`: Name of the data element used for decryption

`errorIndex`: List of the Error Index

`input`: Array of a byte *array* containing encrypted data

`output`: String *array* containing decrypted data

Warning: Ensure that you use the data element with the *No Encryption* method only. Using any other data element might cause corruption of data.

Result

The *output* variable in the method signature contains decrypted data

Example

```
String applicationId = sparkContext.getConf().getAppId();
Protector protector = new PtySparkProtector (applicationId);
String dataElement = "AES-256";
// here input is encrypted String array created using our below API
// public void protect(String dataElement, List<Integer> errorIndex, String[] input,
byte[][] output) throws PtySparkProtectorException;
byte[][] input = { <encrypted string array> };
String[] output = new String[input.length];
List<Integer> errorIndexList = new ArrayList<Integer>();
protector.unprotect(dataElement, errorIndexList, input, output);
```

Exception

`PtySparkProtectorException` : If unable to decrypt data

Table 3-107: Supported Protection Methods

Spark Java APIs	Supported Protection Methods					
	Tokenization	Encryption	FPE	No Encryption	Masking	Monitoring
protect() - Short array data	No	• AES-128	No	Yes	Yes	Yes

Spark Java APIs	Supported Protection Methods					
	Tokenization	Encryption	FPE	No Encryption	Masking	Monitoring
		• AES-256 • 3DES • CUSP				

Note: The protection methods, that are not mentioned in the *Supported Protection Methods* table, are not supported.

3.8.32 reprotect() - Byte array data

Reprotects the byte *array* data, which was protected earlier, with a different data element.

public void reprotect(String oldDataElement, String newDataElement, List<Integer> errorIndex, byte[][] input, byte[][] output)

Parameters

`oldDataElement`: Name of the data element with which data was protected earlier

`newDataElement`: Name of the new data element with which data is reprotected

`errorIndex`: List of the Error Index

`input`: Array of a byte *array* of data to be encrypted

`output`: Array of a byte *array* containing reprotected data

Result

The *output* variable in the method signature contains reprotected data

Example

```
String applicationId = sparkContext.getConf().getAppId();
Protector protector = new PtySparkProtector (applicationId);
String oldDataElement = "Binary";
String newDataElement = "Binary_1";
byte[][] input = new byte[][] {"test1".getBytes(), "test2".getBytes()};
byte[][] output = new byte[input.length][];
List<Integer> errorIndexList = new ArrayList<Integer>();
protector.reprotect(oldDataElement, newDataElement, errorIndexList, input, output);
```

Exception

`PtySparkProtectorException` : If errors occur while reprotecting data

Table 3-108: Supported Protection Methods

Spark Java APIs	Supported Protection Methods					
	Tokenization	Encryption	FPE	No Encryption	Masking	Monitoring
reprotect() - Byte array data	• Numeric (0-9) • Credit Card • Alpha (A-Z) • Upper-case Alpha (A-Z) • Alpha-Numeric (0-9, a-z, A-Z) • Upper Alpha-Numeric (0-9, A-Z) • Lower ASCII • Printable	No	FPE (All - Encoded Bytes Character set should match with the Data elements Encoding Type)	Yes	Yes	Yes

Spark Java APIs	Supported Protection Methods					
	Tokenization	Encryption	FPE	No Encryption	Masking	Monitoring
	• Date (YYYY-MM-DD, DD/MM/YYYY, MM.DD.YYYY) • Datetime (YYYY-MM-DD HH:MM:SS) • Decimal • Unicode (Gen2) • Unicode (Legacy) • Unicode (Base64 - Encoded Byte's Charset should match Dataelement's Encoding Type) • Binary • Email					

Note: The protection methods, that are not mentioned in the *Supported Protection Methods* table, are not supported.

3.8.33 reprotect() - Short array data

Reprotects the short *array* data, which was protected earlier, with a different data element.

public void reprotect(String oldDataElement, String newDataElement, List<Integer> errorIndex, short[] input, short[] output)

Parameters

`oldDataElement`: Name of the data element with which data was protected earlier

`newDataElement`: Name of the new data element with which data is reprotected

`errorIndex`: List of the Error Index

`input`: Short *array* of data to be reprotected

`output`: Short *array* containing reprotected data

Result

The *output* variable in the method signature contains reprotected data

Example

```
String applicationId = sparkContext.getConf().getAppId();
Protector protector = new PtySparkProtector (applicationId);
String oldDataElement = "short";
String newDataElement = "short_1";
short[] input = new short[] {135, 136};
short[] output = new short[input.length];
List<Integer> errorIndexList = new ArrayList<Integer>();
protector.reprotect(oldDataElement, newDataElement, errorIndexList, input, output);
```

Exception

`PtySparkProtectorException` : If errors occur while reprotecting data

Table 3-109: Supported Protection Methods

Spark Java APIs	Supported Protection Methods					
	Tokenization	**Encryption**	**FPE**	**No Encryption**	**Masking**	**Monitoring**
reprotect() - Short array data	Integer (2 Bytes)	No	No	Yes	No	Yes

Note: The protection methods, that are not mentioned in the *Supported Protection Methods* table, are not supported.

3.8.34 reprotect() - Int array data

Reprotects the int *array* data, which was protected earlier, with a different data element.

public void reprotect(String oldDataElement, String newDataElement, List<Integer> errorIndex, int[] input, int[] output)

Parameters

`oldDataElement`: Name of the data element with which data was protected earlier

`newDataElement`: Name of the new data element with which data is reprotected

`errorIndex`: List of the Error Index

`input`: Int *array* of data to be reprotected

`output`: Int *array* containing reprotected data

Result

The *output* variable in the method signature contains reprotected data

Example

```
String applicationId = sparkContext.getConf().getAppId();
Protector protector = new PtySparkProtector (applicationId);
String oldDataElement = "int";
String newDataElement = "int_1";
int[] input = new int[] {234,351};
int[] output = new int[input.length];
List<Integer> errorIndexList = new ArrayList<Integer>();
protector.reprotect(oldDataElement, newDataElement, errorIndexList, input, output);
```

Exception

`PtySparkProtectorException`: If errors occur while reprotecting data

Table 3-110: Supported Protection Methods

Spark Java APIs	Supported Protection Methods					
	Tokenization	**Encryption**	**FPE**	**No Encryption**	**Masking**	**Monitoring**
reprotect() - Int array data	Integer (4 Bytes)	No	No	Yes	No	Yes

Note: The protection methods, that are not mentioned in the *Supported Protection Methods* table, are not supported.

3.8.35 reprotect() - Long array data

Reprotects the long *array* data, which was protected earlier, with a different data element.

public void reprotect(String oldDataElement, String newDataElement, List<Integer> errorIndex, long[] input, long[] output)

Parameters

`oldDataElement`: Name of the data element with which data was protected earlier

`newDataElement`: Name of the new data element with which data is reprotected

`errorIndex`: List of the Error Index

`input`: Long *array* of data to be reprotected

`output`: Long *array* containing reprotected data

Result

The *output* variable in the method signature contains reprotected data

Example

```
String applicationId = sparkContext.getConf().getAppId();
Protector protector = new PtySparkProtector (applicationId);
String oldDataElement = "long";
String newDataElement = "long_1";
long[] input = new long[] {1234, 135}};
long[] output = new long[input.length];
List<Integer> errorIndexList = new ArrayList<Integer>();
protector.reprotect(oldDataElement, newDataElement, errorIndexList, input, output);
```

Exception

`PtySparkProtectorException`: If errors occur while reprotecting data

Table 3-111: Supported Protection Methods

Spark Java APIs	Supported Protection Methods					
	Tokenization	**Encryption**	**FPE**	**No Encryption**	**Masking**	**Monitoring**
reprotect() - Long array data	Integer (8 Bytes)	No	No	Yes	No	Yes

Note: The protection methods, that are not mentioned in the *Supported Protection Methods* table, are not supported.

3.8.36 reprotect() - Float array data

Reprotects the float *array* data, which was protected earlier, with a different data element.

public void reprotect(String oldDataElement, String newDataElement, List<Integer> errorIndex, float[] input, float[] output)

Parameters

`oldDataElement`: Name of the data element with which data was protected earlier

`newDataElement`: Name of the new data element with which data is reprotected

`errorIndex`: List of the Error Index

`input`: Float *array* of data to be reprotected

`output`: Float *array* containing reprotected

Warning: Ensure that you use the data element with the *No Encryption* method only. Using any other data element might cause corruption of data.

data

Result

The *output* variable in the method signature contains reprotected data

Example

```
String applicationId = sparkContext.getConf().getAppId();
Protector protector = new PtySparkProtector (applicationId);
String oldDataElement = "NoEnc";
String newDataElement = "NoEnc_1";
float[] input = new float[] {23.56f, 26.43f}};
float[] output = new float[input.length];
List<Integer> errorIndexList = new ArrayList<Integer>();
protector.reprotect(oldDataElement, newDataElement, errorIndexList, input, output);
```

Exception

`PtySparkProtectorException` : If errors occur while reprotecting data

Table 3-112: Supported Protection Methods

Spark Java APIs	Supported Protection Methods					
	Tokenization	**Encryption**	**FPE**	**No Encryption**	**Masking**	**Monitoring**
reprotect() - Float array data	No	No	No	Yes	No	Yes

Note: The protection methods, that are not mentioned in the *Supported Protection Methods* table, are not supported.

3.8.37 reprotect() - Double array data

Reprotects the double *array* data, which was protected earlier, with a different data element.

public void reprotect(String oldDataElement, String newDataElement, List<Integer> errorIndex, double[] input, double[] output)

Parameters

`oldDataElement`: Name of the data element with which data was protected earlier

`newDataElement`: Name of the new data element with which data is reprotected

`errorIndex`: List of the Error Index

`input`: Double *array* of data to be encrypted

`output`: Double *array* containing decrypted data

Note: Ensure that you use the data element with the *No Encryption* method only. Using any other data element might cause corruption of data.

Result

The *output* variable in the method signature contains reprotected data

Example

```
String applicationId = sparkContext.getConf().getAppId();
Protector protector = new PtySparkProtector (applicationId);
String oldDataElement = "NoEnc";
String newDataElement = "NoEnc_1";
double[] input = new double[] {235.5, 1235.66};
double[] output = new double[input.length];
List<Integer> errorIndexList = new ArrayList<Integer>();
protector.reprotect(oldDataElement, newDataElement, errorIndexList, input, output);
```

Exception

`PtySparkProtectorException` : If errors occur while reprotecting data

Table 3-113: Supported Protection Methods

Spark Java APIs	Supported Protection Methods					
	Tokenization	**Encryption**	**FPE**	**No Encryption**	**Masking**	**Monitoring**
reprotect() - Double array data	No	No	No	Yes	No	Yes

Note: The protection methods, that are not mentioned in the *Supported Protection Methods* table, are not supported.

3.8.38 reprotect() - String array data

Reprotects the string *array* data, which was protected earlier, with a different data element.

public void reprotect(String oldDataElement, String newDataElement, List<Integer> errorIndex, String[] input, String[] output)

Parameters

`oldDataElement`: Name of the data element with which data was protected earlier

`newDataElement`: Name of the new data element with which data is reprotected

`errorIndex`: List of the Error Index

`input`: String *array* of data to be reprotected

`output`: String *array* containing reprotected data

Result

The *output* variable in the method signature contains reprotected data

Example

```
String applicationId = sparkContext.getConf().getAppId();
Protector protector = new PtySparkProtector (applicationId);
String oldDataElement = "AlphaNum";
String newDataElement = "AlphaNum_1";
String[] input = new String[] {"test1", "test2"};
String[] output = new String[input.length];
List<Integer> errorIndexList = new ArrayList<Integer>();
protector.reprotect(oldDataElement, newDataElement, errorIndexList, input, output);
```

Exception

`PtySparkProtectorException` : If unable to reprotecting data

Table 3-114: Supported Protection Methods

Spark Java APIs	Supported Protection Methods					
	Tokenization	**Encryption**	**FPE**	**No Encryption**	**Masking**	**Monitoring**
reprotect() - String array data	• Numeric (0-9) • Credit Card • Alpha (A-Z) • Upper-case Alpha (A-Z) • Alpha-Numeric (0-9, a-z, A-Z) • Upper Alpha-Numeric (0-9, A-Z) • Lower ASCII • Printable • Date (YYYY-MM-DD, DD/MM/YYYY, MM.DD.YYYY) • Datetime (YYYY-MM-DD HH:MM:SS) • Decimal • Unicode (Gen2) • Unicode (Legacy) • Unicode (Base64 - Encoded Byte's Charset should match	No	FPE(All)	Yes	Yes	Yes

Spark Java APIs	Supported Protection Methods					
	Tokenization	Encryption	FPE	No Encryption	Masking	Monitoring
	Dataelement's Encoding Type) • Binary • Email					

Note: The protection methods, that are not mentioned in the *Supported Protection Methods* table, are not supported.

3.9 Spark SQL UDFs

This section describes the Spark SQL User Defined Functions (UDFs) that are available for protection and unprotection in Big Data Protector to build secure Big Data applications.

Note: The example code snippets provided in this section utilize SQL queries to invoke the UDFs, after they are registered, using the *sql.Context.sql()* method.

For more information about the different methods of invoking the Spark SQL UDFs, refer to *Big Data Protector Guide 8.1.0.0*.

3.9.1 ptyGetVersion()

This UDF returns the current version of PEP.

ptyGetVersion()

Parameters

None

Result

This UDF returns the current version of PEP.

Example

```
sqlContext.udf.register("ptyGetVersion", com.protegrity.spark.udf.ptyGetVersion _)
sqlContext.sql("select ptyGetVersion()").show()
```

3.9.2 ptyWhoAmI()

This UDF returns the current logged in user.

ptyWhoAmI()

Parameters

None

Result

This UDF returns the current logged in user.

Example

```
sqlContext.udf.register("ptyWhoAmI", com.protegrity.spark.udf.ptyWhoAmI _)
sqlContext.sql("select ptyWhoAmI()").show()
```

3.9.3 ptyProtectStr()

This UDF protects *string* format data, which is provided as input.

> **Note:** For Date and Datetime type of data elements, the protect API returns an invalid input data error if the input value falls between the non-existent date range from 05-OCT-1582 to 14-OCT-1582 of the Gregorian Calendar.
>
> For more information about the tokenization and de-tokenization of the cutover dates of the Proleptic Gregorian Calendar, refer to the section *Date* and *Datetime* tokenization in the *Protection Method Reference Guide 9.1.0.0*.

ptyProtectStr(String colName, String dataElement)

Parameters

`colName` : The column that contains data in *String* format, which needs to be protected

`dataElement` : The data element that will be used to protect *string* format data

Result

This UDF returns *string* format data, which is protected.

Example

```
import sqlContext.implicits._

val df = sc.parallelize(List("hello", "world")).toDF("string_col")

val protectStrUDF = sqlContext.udf
.register("ptyProtectStr", com.protegrity.spark.udf.ptyProtectStr _)

df.registerTempTable("string_test")

sqlContext
.sql(
"select ptyProtectStr(string_col, 'Token_Alphanum') as protected from string_test")
.show(false)
```

Table 3-115: Supported Protection Methods

Spark SQL UDFs	Supported Protection Methods					
	Tokenization	**Encryption**	**FPE**	**No Encryption**	**Masking**	**Monitoring**
ptyProtectStr()	• Numeric (0-9) • Credit Card • Alpha (A-Z) • Upper-case Alpha (A-Z) • Alpha-Numeric (0-9, a-z, A-Z) • Upper Alpha-Numeric (0-9, A-Z) • Lower ASCII • Date (YYYY-MM-DD, DD/MM/YYYY, MM.DD.YYYY) • Datetime (YYYY-MM-DD HH:MM:SS) • Decimal	No	Yes	Yes	Yes	Yes

Spark SQL UDFs	Supported Protection Methods					
	Tokenization	Encryption	FPE	No Encryption	Masking	Monitoring
	• Unicode (Gen2) • Unicode (Legacy) • Unicode (Base64 - Encoded Byte's Charset should match Dataelement's Encoding Type) • Email					

Note: The protection methods, that are not mentioned in the *Supported Protection Methods* table, are not supported.

3.9.4 ptyProtectUnicode()

This UDF protects *string* (Unicode) format data, which is provided as input.

ptyProtectUnicode(String colName, String dataElement)

Parameters

`colName` : The column that contains data in *String* (Unicode) format, which needs to be protected

`dataElement` : The data element that will be used to protect *string* (Unicode) format data

Result

This UDF returns *string* format data, which is protected.

Example

```
import sqlContext.implicits._

val df = sc.parallelize(List("Marylène", "")).toDF("unicode_col")

val protectUnicodeUDF = sqlContext.udf.register(
  "ptyProtectUnicode",
  com.protegrity.spark.udf.ptyProtectUnicode _)

df.registerTempTable("unicode_test")

sqlContext
  .sql(
"select ptyProtectUnicode(unicode_col, 'Token_Unicode') as protected from unicode_test")
  .show(false)
```

Table 3-116: Supported Protection Methods

Spark SQL UDFs	Supported Protection Methods					
	Tokenization	Encryption	FPE	No Encryption	Masking	Monitoring
ptyProtectUnicode()	• Unicode (Legacy) • Unicode Base64	No	No	Yes	No	Yes

Note: The protection methods, that are not mentioned in the *Supported Protection Methods* table, are not supported.

3.9.5 ptyProtectInt()

This UDF protects *integer* format data, which is provided as input.

ptyProtectInt(Int input, String dataElement)

Parameters

`colName` : The column that contains data in *Integer* format, which needs to be protected

`dataElement` : The data element that will be used to protect *string* (Unicode) format data

Result

This UDF returns *string* format data, which is protected.

Example

```
import sqlContext.implicits._

val df = sc.parallelize(List(1234, 2345)).toDF("int_col")

val protectIntUDF = sqlContext.udf
.register("ptyProtectInt", com.protegrity.spark.udf.ptyProtectInt _)

df.registerTempTable("int_test")

sqlContext
.sql("select ptyProtectInt(int_col, 'Token_Int') as protected from int_test")
.show(false)
```

Table 3-117: Supported Protection Methods

Spark SQL UDFs	Supported Protection Methods					
	Tokenization	Encryption	FPE	No Encryption	Masking	Monitoring
ptyProtectInt()	Integer 4 Bytes	No	No	Yes	No	Yes

Note: The protection methods, that are not mentioned in the *Supported Protection Methods* table, are not supported.

3.9.6 ptyProtectShort()

This UDF protects *Short* format data, which is provided as input.

ptyProtectShort(Short colName, String dataElement)

Parameters

`colName` : The column that contains data in *Short*format, which needs to be protected

`dataElement` : The data element that will be used to protect *short*format data

Result

This UDF returns *short* format data, which is protected.

Example

```
import sqlContext.implicits._

val df = sc.parallelize(List(1234, 2345)).map{x =>
ShortClass(x.toShort)
}.toDF("short_col")

val protectShortUDF = sqlContext.udf
.register("ptyProtectShort", com.protegrity.spark.udf.ptyProtectShort _)

df.registerTempTable("short_test")

sqlContext
.sql("select ptyProtectShort(short_col, 'Token_Short') as protected from short_test")
.show(false)
```

Table 3-118: Supported Protection Methods

Spark SQL UDFs	Supported Protection Methods					
	Tokenization	Encryption	FPE	No Encryption	Masking	Monitoring
ptyProtectShort()	Integer 2 Bytes	No	No	Yes	No	Yes

Note: The protection methods, that are not mentioned in the *Supported Protection Methods* table, are not supported.

3.9.7 ptyProtectLong()

This UDF protects *Long* format data, which is provided as input.

ptyProtectLong(Long colName, String dataElement)

Parameters

`colName` : The column that contains data in *Long* format, which needs to be protected

`dataElement` : The data element that will be used to protect *long* format data

Result

This UDF returns *long* format data, which is protected.

Example

```
import sqlContext.implicits._

val df = sc.parallelize(List(1234l, 2345l)).toDF("long_col")

val protectLongUDF = sqlContext.udf
.register("ptyProtectLong", com.protegrity.spark.udf.ptyProtectLong _)

df.registerTempTable("long_test")

sqlContext
.sql("select ptyProtectLong(long_col, 'Token_Long') as protected from long_test")
.show(false)
```

Table 3-119: Supported Protection Methods

Spark SQL UDFs	Supported Protection Methods					
	Tokenization	Encryption	FPE	No Encryption	Masking	Monitoring
ptyProtectLong()	Integer 8 Bytes	No	No	Yes	No	Yes

Note: The protection methods, that are not mentioned in the *Supported Protection Methods* table, are not supported.

3.9.8 ptyProtectDate()

This UDF protects *date* format data, which is provided as input.

ptyProtectDate(Date colName, String dataElement)

Note: In the Big Data Protector, version 7.0 release, the *date* format supported is *YYYY-MM-DD* only.

Parameters

`colName` : The column that contains data in *Date* format, which needs to be protected

`dataElement` : The data element that will be used to protect *date* format data

Result

This UDF returns *date* format data, which is protected.

Example

```
import sqlContext.implicits._


val d1 = Date.valueOf("2016-12-28")
val d2 = Date.valueOf("2016-12-28")
val df = sc.parallelize(Seq((d1, d2))).toDF("date_col1","date_col2")


val protectDateUDF = sqlContext.udf
  .register("ptyProtectDate", com.protegrity.spark.udf.ptyProtectDate _)

df.registerTempTable("date_test")

sqlContext
  .sql("select ptyProtectDate(date_col1, 'Token_Date') as protected from date_test")
  .show(false)
```

Table 3-120: Supported Protection Methods

Spark SQL UDFs	Supported Protection Methods					
	Tokenization	**Encryption**	**FPE**	**No Encryption**	**Masking**	**Monitoring**
ptyProtectDate()	Date (YYYY-MM-DD, DD/MM/YYYY, MM.DD.YYYY)	No	No	Yes	No	Yes

Note: The protection methods, that are not mentioned in the *Supported Protection Methods* table, are not supported.

3.9.9 ptyProtectDateTime()

This UDF protects *timestamp* format data, which is provided as input.

ptyProtectDateTime(Timestamp colName, String dataElement)

Parameters

`colName` : The column that contains data in *Timestamp* format, which needs to be protected

`dataElement` : The data element that will be used to protect *timestamp* format data

Result

This UDF returns *timestamp* format data, which is protected.

Example

```
import sqlContext.implicits._

val d1 = Timestamp.valueOf("2016-12-28 13:09:38.104")
val d2 = Timestamp.valueOf("2016-12-29 12:09:38.104")

val df = sc.parallelize(Seq((d1, d2))).toDF("datetime_col1","datetime_col2")

val protectDateTimeUDF = sqlContext.udf.register(
"ptyProtectDateTime",com.protegrity.spark.udf.ptyProtectDateTime _)
df.registerTempTable("datetime_test")

sqlContext
  .sql(
"select ptyProtectDateTime(datetime_col1, 'Token_Datetime') as protected from
datetime_test")
  .show(false)
```

Table 3-121: Supported Protection Methods

Spark SQL UDFs	Supported Protection Methods					
	Tokenization	Encryption	FPE	No Encryption	Masking	Monitoring
ptyProtectDateTime()	Datetime (YYYY-MM-DD HH:MM:SS)	No	No	Yes	No	Yes

Note: The protection methods, that are not mentioned in the *Supported Protection Methods* table, are not supported.

3.9.10 ptyProtectFloat()

This UDF protects *float* format data, which is provided as input.

ptyProtectFloat(Float colName, String dataElement)

Parameters

`colName` : The column that contains data in *Float* format, which needs to be protected

`dataElement` : The data element that will be used to protect *float* format data

Warning: Ensure that you use the *No Encryption* data element only. Using any other data element might cause corruption of data.

Result

This UDF returns *float* format data, which is protected.

Example

```
import sqlContext.implicits._

val input = Seq((1234.345f, 1343.3345f))

val df = sc.parallelize(input).toDF("float_col1","float_col2")

val protectFloatUDF = sqlContext.udf
  .register("ptyProtectFloat", com.protegrity.spark.udf.ptyProtectFloat _)

df.registerTempTable("float_test")

sqlContext
  .sql(
"select ptyProtectFloat(float_col1, 'Token_NoEncryption') as protected from float_test")
  .show(false)
```

Table 3-122: Supported Protection Methods

Spark SQL UDFs	Supported Protection Methods					
	Tokenization	Encryption	FPE	No Encryption	Masking	Monitoring
ptyProtectFloat()	No	No	No	Yes	No	Yes

Note: The protection methods, that are not mentioned in the *Supported Protection Methods* table, are not supported.

3.9.11 ptyProtectDouble()

This UDF protects *double* format data, which is provided as input.

ptyProtectDouble(Double colName, String dataElement)

Parameters

`colName` : The column that contains data in *Double* format, which needs to be protected

dataElement : The data element that will be used to protect *double* format data

> **Warning:** Ensure that you use the *No Encryption* data element only. Using any other data element might cause corruption of data.

Result

This UDF returns *BigDecimal* format data, which is protected.

Example

```
import sqlContext.implicits._

val input = Seq((1234.345, 1343.3345))

val df = sc.parallelize(input).toDF("double_col1","double_col2")

val protectDoubleUDF = sqlContext.udf.register(
"ptyProtectDouble",com.protegrity.spark.udf.ptyProtectDouble _)
df.registerTempTable("double_test")

sqlContext
  .sql(
"select ptyProtectDouble(double_col1, 'Token_NoEncryption') as protected from
double_test")
  .show(false)
```

Table 3-123: Supported Protection Methods

Spark SQL UDFs	Supported Protection Methods					
	Tokenization	**Encryption**	**FPE**	**No Encryption**	**Masking**	**Monitoring**
ptyProtectDouble()	No	No	No	Yes	No	Yes

> **Note:** The protection methods, that are not mentioned in the *Supported Protection Methods* table, are not supported.

3.9.12 ptyProtectDecimal()

This UDF protects *decimal* format data, which is provided as input.

ptyProtectDec(BigDecimal colName, String dataElement)

Parameters

colName : The column that contains data in *BigDecimal* format data, which needs to be protected

dataElement : The data element that will be used to protect *BigDecimal* format data

> **Warning:** Ensure that you use the *No Encryption* data element only. Using any other data element might cause corruption of data.

> **Caution:** Before the *ptyProtectDecimal()* UDF is called, Spark SQL rounds off the decimal value in the table to 18 digits in scale, irrespective of the length of the data.

Result

This UDF returns *BigDecimal* format data, which is protected.

Example

```
import sqlContext.implicits._

val d1 = Timestamp.valueOf("2016-12-28 13:09:38.104")
val d2 = Timestamp.valueOf("2016-12-29 12:09:38.104")

val df = sc.parallelize(Seq((d1, d2))).toDF("datetime_col1","datetime_col2")

val protectDateTimeUDF = sqlContext.udf.register(
```

```
"ptyProtectDateTime",com.protegrity.spark.udf.ptyProtectDateTime _)
df.registerTempTable("datetime_test")

sqlContext
  .sql(
"select ptyProtectDateTime(datetime_col1, 'Token_Datetime') as protected from
datetime_test")
  .show(false)
```

Table 3-124: Supported Protection Methods

Spark SQL UDFs	Supported Protection Methods					
	Tokenization	**Encryption**	**FPE**	**No Encryption**	**Masking**	**Monitoring**
ptyProtectDecimal()	No	No	No	Yes	No	Yes

Note: The protection methods, that are not mentioned in the *Supported Protection Methods* table, are not supported.

3.9.13 ptyUnprotectStr()

This UDF unprotects the protected string **format** data**, which is provided as input**.

Note: For Date and Datetime type of data elements, the protect API returns an invalid input data error if the input value falls between the non-existent date range from 05-OCT-1582 to 14-OCT-1582 of the Gregorian Calendar.

For more information about the tokenization and de-tokenization of the cutover dates of the Proleptic Gregorian Calendar, refer to the section *Date* and *Datetime* tokenization in the *Protection Method Reference Guide 8.1.0.0*.

ptyUnprotectStr(String colName, String dataElement)

Parameters

`colName` : The column that contains data in *string* format, which needs to be unprotected

`dataElement` : The data element that will be used to unprotected *string* format data

Warning: Ensure that you use the *No Encryption* data element only. Using any other data element might cause corruption of data.

Result

This UDF returns *string* format data, which is unprotected.

Example

```
import sqlContext.implicits._

val df = sc.parallelize(List("A2yae", "2LbRS")).toDF("string_col")

val unprotectStrUDF = sqlContext.udf
  .register("ptyUnprotectStr", com.protegrity.spark.udf.ptyUnprotectStr _)

df.registerTempTable("string_test")

sqlContext
  .sql(
"select ptyUnprotectStr(string_col, 'Token_Alphanum') as unprotected from string_test")
  .show(false)
```

Table 3-125: Supported Protection Methods

Spark SQL UDFs	Supported Protection Methods					
	Tokenization	**Encryption**	**FPE**	**No Encryption**	**Masking**	**Monitoring**
ptyUnprotectStr()	• Numeric (0-9)	No	Yes	Yes	Yes	Yes

Spark SQL UDFs	Supported Protection Methods					
	Tokenization	Encryption	FPE	No Encryption	Masking	Monitoring
	• Credit Card • Alpha (A-Z) • Upper-case Alpha (A-Z) • Alpha-Numeric (0-9, a-z, A-Z) • Upper Alpha-Numeric (0-9, A-Z) • Lower ASCII • Date (YYYY-MM-DD, DD/MM/YYYY, MM.DD.YYYY) • Datetime (YYYY-MM-DD HH:MM:SS) • Decimal • Unicode (Gen2) • Unicode (Legacy) • Unicode (Base64 - Encoded Byte's Charset should match Dataelement's Encoding Type) • Email					

Note: The protection methods, that are not mentioned in the *Supported Protection Methods* table, are not supported.

3.9.14 ptyUnprotectUnicode()

This UDF unprotects the protected string **format** data**, which is provided as input**.

ptyUnprotectUnicode(String colName, String dataElement)

Parameters

`colName` : The column that contains data in protected *string* format, which needs to be unprotected

`dataElement` : The data element that will be used to unprotected *string* format data

Warning: Ensure that you use the *No Encryption* data element only. Using any other data element might cause corruption of data.

Result

This UDF returns *string* (Unicode) format data, which is unprotected.

Example

```
import sqlContext.implicits._

val df =
  sc.parallelize(List("jmR6Dw4Tqzlw441n5qEMtMEUKsI", "Q1dwK")).toDF("unicode_col")

val unprotectUnicodeUDF = sqlContext.udf.register(
```

```
  "ptyUnprotectUnicode",
  com.protegrity.spark.udf.ptyUnprotectUnicode _)

df.registerTempTable("unicode_test")

sqlContext
  .sql(
"select ptyUnprotectUnicode(unicode_col, 'Token_Unicode') as unprotected from
unicode_test")
  .show(false)
```

Table 3-126: Supported Protection Methods

Spark SQL UDFs	Supported Protection Methods					
	Tokenization	**Encryption**	**FPE**	**No Encryption**	**Masking**	**Monitoring**
ptyUnprotectUnicode()	• Unicode (Legacy) • Unicode Base64	No	No	Yes	No	Yes

Note: The protection methods, that are not mentioned in the *Supported Protection Methods* table, are not supported.

3.9.15 ptyUnprotectInt()

This UDF unprotects the protected integer format data , which is provided as input.

ptyUnprotectInt(Int colName, String dataElement)

Parameters

`colName` : The column that contains data in protected *integer* format, which needs to be unprotected

`dataElement` : The data element that will be used to unprotected *integer* format data

Caution: If an unauthorized user, with no privileges to unprotect data in the security policy, and the output value set to NULL, attempts to unprotect the protected data of Numeric type data containing *Short*, *Int*, *Float*, *Long*, *Double*, and *Decimal* format values using the respective Spark SQL UDFs, then the output is *0*.

Result

This UDF returns *integer* (Unicode) format data, which is unprotected.

Example

```
import sqlContext.implicits._

val df = sc.parallelize(List(-834202911, -586348592)).toDF("int_col")

val unprotectIntUDF = sqlContext.udf
  .register("ptyUnprotectInt", com.protegrity.spark.udf.ptyUnprotectInt _)

df.registerTempTable("int_test")

sqlContext
  .sql("select ptyUnprotectInt(int_col, 'Token_Int') as unprotected from int_test")
  .show(false)
```

Table 3-127: Supported Protection Methods

Spark SQL UDFs	Supported Protection Methods					
	Tokenization	**Encryption**	**FPE**	**No Encryption**	**Masking**	**Monitoring**
ptyUnprotectInt()	Integer 4 Bytes	No	No	Yes	No	Yes

Note: The protection methods, that are not mentioned in the *Supported Protection Methods* table, are not supported.

3.9.16 ptyUnprotectShort()

This UDF unprotects the protected short **format** data**, which is provided as input**.

ptyUnprotectShort(Short colName, String dataElement)

Parameters

`colName` : The column that contains data in protected *short* format, which needs to be unprotected

`dataElement` : The data element that will be used to unprotected *short* format data

> **Caution:** If an unauthorized user, with no privileges to unprotect data in the security policy, and the output value set to NULL, attempts to unprotect the protected data of Numeric type data containing *Short*, *Int*, *Float*, *Long*, *Double*, and *Decimal* format values using the respective Spark SQL UDFs, then the output is *0*.

Result

This UDF returns *short* (Unicode) format data, which is unprotected.

Example

```
import sqlContext.implicits._

val df = sc.parallelize(List(-24453, 1827)).map(x =>
ShortClass(x.toShort))toDF("short_col")

val unprotectShortUDF = sqlContext.udf.register(
  "ptyUnprotectShort",
  com.protegrity.spark.udf.ptyUnprotectShort _)

df.registerTempTable("short_test")

sqlContext
  .sql(
"select ptyUnprotectShort(short_col, 'Token_Short') as unprotected from short_test")
  .show(false)
```

Table 3-128: Supported Protection Methods

Spark SQL UDFs	Supported Protection Methods					
	Tokenization	**Encryption**	**FPE**	**No Encryption**	**Masking**	**Monitoring**
ptyUnprotectShort()	Integer 2 Bytes	No	No	Yes	No	Yes

> **Note:** The protection methods, that are not mentioned in the *Supported Protection Methods* table, are not supported.

3.9.17 ptyUnprotectLong()

This UDF unprotects the protected long **format** data**, which is provided as input**.

ptyUnprotectLong(Long colName, String dataElement)

Parameters

`colName` : The column that contains data in protected *long* format, which needs to be unprotected

`dataElement` : The data element that will be used to unprotected *long* format data

> **Caution:** If an unauthorized user, with no privileges to unprotect data in the security policy, and the output value set to NULL, attempts to unprotect the protected data of Numeric type data containing *Short*, *Int*, *Float*, *Long*, *Double*, and *Decimal* format values using the respective Spark SQL UDFs, then the output is *0*.

Result

This UDF returns *long* (Unicode) format data, which is unprotected.

Example

```
import sqlContext.implicits._

val df =
  sc.parallelize(List(4960833108022315290l, -18545667847517265481)).toDF("long_col")

val unprotectLongUDF = sqlContext.udf.register(
  "ptyUnprotectLong",
  com.protegrity.spark.udf.ptyUnprotectLong _)

df.registerTempTable("long_test")

sqlContext
  .sql(
"select ptyUnprotectLong(long_col, 'Token_Long') as unprotected from long_test")
  .show(false)
```

Table 3-129: Supported Protection Methods

Spark SQL UDFs	Supported Protection Methods					
	Tokenization	Encryption	FPE	No Encryption	Masking	Monitoring
ptyUnprotectLong()	Integer 8 Bytes	No	No	Yes	No	Yes

Note: The protection methods, that are not mentioned in the *Supported Protection Methods* table, are not supported.

3.9.18 ptyUnprotectDate()

This UDF unprotects the protected date **format** data**, which is provided as input**.

ptyUnprotectDate(Date colName, String dataElement)

Parameters

`colName` : The column that contains data in protected *date* format, which needs to be unprotected

`dataElement` : The data element that will be used to unprotected *date* format data

Result

This UDF returns *date* (Unicode) format data, which is unprotected.

Example

```
import sqlContext.implicits._

val d1 = Date.valueOf("1881-04-07") //new Date(System.currentTimeMillis())
val d2 = Date.valueOf("2016-12-28") //new Date(System.currentTimeMillis())

val df = sc.parallelize(Seq((d1, d2))).toDF("date_col1", "date_col2")


val unprotectDateUDF = sqlContext.udf.register(
  "ptyUnprotectDate",
  com.protegrity.spark.udf.ptyUnprotectDate _)

df.registerTempTable("date_test")

sqlContext
  .sql(
"select ptyUnprotectDate(date_col1, 'Token_Date') as unprotected from date_test")
  .show(false)
```

Table 3-130: Supported Protection Methods

Spark SQL UDFs	Supported Protection Methods					
	Tokenization	Encryption	FPE	No Encryption	Masking	Monitoring
ptyUnprotectDate()	Date (YYYY-MM-DD, DD/MM/YYYY, MM.DD.YYYY)	No	No	Yes	No	Yes

Note: The protection methods, that are not mentioned in the *Supported Protection Methods* table, are not supported.

3.9.19 ptyUnprotectDateTime()

This UDF unprotects the protected timestamp **format** data**, which is provided as input**.

ptyUnprotectDateTime(Timestamp colName, String dataElement)

Parameters

`colName` : The column that contains data in protected *timestamp* format, which needs to be unprotected

`dataElement` : The data element that will be used to unprotected *timestamp* format data

Result

This UDF returns *timestamp* (Unicode) format data, which is unprotected.

Example

```
import sqlContext.implicits._

val d1 = Timestamp.valueOf("1197-02-10 13:09:38.104")
val d2 = Timestamp.valueOf("2016-12-29 12:09:38.104")

val df = sc.parallelize(Seq((d1, d2))).toDF("datetime_col1", "datetime_col2")

val unprotectDateTimeUDF = sqlContext.udf.register(
  "ptyUnprotectDateTime",
  com.protegrity.spark.udf.ptyUnprotectDateTime _)

df.registerTempTable("datetime_test")

sqlContext
  .sql(
"select ptyUnprotectDateTime(datetime_col1, 'Token_Datetime') as unprotected from
datetime_test")
  .show(false)
```

Table 3-131: Supported Protection Methods

Spark SQL UDFs	Supported Protection Methods					
	Tokenization	Encryption	FPE	No Encryption	Masking	Monitoring
ptyUnprotectDateTime()	Datetime (YYYY-MM-DD HH:MM:SS)	No	No	Yes	No	Yes

Note: The protection methods, that are not mentioned in the *Supported Protection Methods* table, are not supported.

3.9.20 ptyUnprotectFloat()

This UDF unprotects protected *float* format data, which is provided as input.

ptyUnprotectFloat(Float colName, String dataElement)

Parameters

colName : The column that contains data in protected *float* format, which needs to be unprotected

dataElement : The data element that will be used to unprotected *float* format data

> **Warning:** Ensure that you use the *No Encryption* data element only. Using any other data element might cause corruption of data.

> **Caution:** If an unauthorized user, with no privileges to unprotect data in the security policy, and the output value set to NULL, attempts to unprotect the protected data of Numeric type data containing *Short*, *Int*, *Float*, *Long*, *Double*, and *Decimal* format values using the respective Spark SQL UDFs, then the output is *0*.

Result

This UDF returns *float* (Unicode) format data, which is unprotected.

Example

```
import sqlContext.implicits._

val input = Seq((1234.345f, 1343.3345f))

val df = sc.parallelize(input).toDF("float_col1","float_col2")

val unprotectFloatUDF = sqlContext.udf.register(
  "ptyUnprotectFloat",
  com.protegrity.spark.udf.ptyUnprotectFloat _)

df.registerTempTable("float_test")

sqlContext
  .sql(
"select ptyUnprotectFloat(float_col1, 'Token_NoEncryption') as unprotected from
float_test")
  .show(false)
```

Table 3-132: Supported Protection Methods

Spark SQL UDFs	Supported Protection Methods					
	Tokenization	Encryption	FPE	No Encryption	Masking	Monitoring
ptyUnprotectFloat()	No	No	No	Yes	No	Yes

> **Note:** The protection methods, that are not mentioned in the *Supported Protection Methods* table, are not supported.

3.9.21 ptyUnprotectDouble()

This UDF unprotects protected *double* format data, which is provided as input.

ptyUnprotectDouble(Double colName, String dataElement)

Parameters

colName : The column that contains data in protected *double* format, which needs to be unprotected

dataElement : The data element that will be used to unprotected *double* format data

> **Warning:** Ensure that you use the *No Encryption* data element only. Using any other data element might cause corruption of data.

> **Caution:** If an unauthorized user, with no privileges to unprotect data in the security policy, and the output value set to NULL, attempts to unprotect the protected data of Numeric type data containing *Short*, *Int*, *Float*, *Long*, *Double*, and *Decimal* format values using the respective Spark SQL UDFs, then the output is *0*.

Result

This UDF returns *double* (Unicode) format data, which is unprotected.

Example

```
import sqlContext.implicits._

val input = Seq((1234.345, 1343.3345))

val df = sc.parallelize(input).toDF("double_col1", "double_col2'")

val unprotectDoubleUDF = sqlContext.udf.register(
  "ptyUnprotectDouble",
  com.protegrity.spark.udf.ptyUnprotectDouble _)

df.registerTempTable("double_test")

sqlContext
  .sql(
"select ptyUnprotectDouble(double_col1, 'Token_NoEncryption') as unprotected from
double_test")
  .show(false)
```

Table 3-133: Supported Protection Methods

Spark SQL UDFs	Supported Protection Methods					
	Tokenization	Encryption	FPE	No Encryption	Masking	Monitoring
ptyUnprotectDouble( )	No	No	No	Yes	No	Yes

Note: The protection methods, that are not mentioned in the *Supported Protection Methods* table, are not supported.

3.9.22 ptyUnprotectDecimal()

This UDF unprotects protected *decimal* data, which is provided as input.

ptyUnprotectDec(BigDecimal colName, String dataElement)

Parameters

`colName` : The column that contains data in protected *BigDecimal* format, which needs to be unprotected

`dataElement` : The data element that will be used to unprotected *BigDecimal* format data

Warning: Ensure that you use the *No Encryption* data element only. Using any other data element might cause corruption of data.

Caution: Before the *ptyUnprotectDecimal()* UDF is called, Spark SQL rounds off the decimal value in the table to 18 digits in scale, irrespective of the length of the data.

Caution: If an unauthorized user, with no privileges to unprotect data in the security policy, and the output value set to NULL, attempts to unprotect the protected data of Numeric type data containing *Short*, *Int*, *Float*, *Long*, *Double*, and *Decimal* format values using the respective Spark SQL UDFs, then the output is *0*.

Result

This UDF returns *BigDecimal* (Unicode) format data, which is unprotected.

Example

```
import sqlContext.implicits._

val input = Seq((1234.345, 1343.3345))

val df = sc.parallelize(input).toDF("double_col1", "double_col2'")

val unprotectDoubleUDF = sqlContext.udf.register(
```

```
  "ptyUnprotectDouble",
  com.protegrity.spark.udf.ptyUnprotectDouble _)

df.registerTempTable("double_test")

sqlContext
  .sql(
"select ptyUnprotectDouble(double_col1, 'Token_NoEncryption') as unprotected from
double_test")
  .show(false)
```

Table 3-134: Supported Protection Methods

Spark SQL UDFs	Supported Protection Methods					
	Tokenization	Encryption	FPE	No Encryption	Masking	Monitoring
ptyUnprotectDecimal()	No	No	No	Yes	No	Yes

Note: The protection methods, that are not mentioned in the *Supported Protection Methods* table, are not supported.

3.9.23 ptyReprotectStr()

This UDF reprotects *string* format protected data, which was earlier protected using the *ptyProtectStr* UDF, with a different data element.

```
ptyReprotectStr(String colName, String oldDataElement, String newDataElement)
```

Parameters

`colName` : The column that contains data in *String* format, which needs to be reprotected

`oldDataElement` : The data element that was used to protect the data earlier

`newDataElement` : The new data element that will be used to reprotect the data

Result

This UDF returns *string* format data, which is protected.

Example

```
import sqlContext.implicits._

val df = sc.parallelize(List("hello", "world")).toDF("string_col")

val reprotectStrUDF = sqlContext.udf
  .register("ptyReprotectStr", com.protegrity.spark.udf.ptyReprotectStr _)

df.registerTempTable("string_test")

sqlContext
  .sql(
"select ptyReprotectStr(string_col, 'Token_Alphanum', ' Token_Alphanum_1') as
reprotected from string_test")
  .show(false)
```

Table 3-135: Supported Protection Methods

Spark SQL UDFs	Supported Protection Methods					
	Tokenization	Encryption	FPE	No Encryption	Masking	Monitoring
ptyReprotectStr()	• Numeric (0-9) • Credit Card • Alpha (A-Z) • Upper-case Alpha (A-Z)	No	Yes	Yes	Yes	Yes

Spark SQL UDFs	Supported Protection Methods					
	Tokenization	Encryption	FPE	No Encryption	Masking	Monitoring
	• Alpha-Numeric (0-9, a-z, A-Z) • Upper Alpha-Numeric (0-9, A-Z) • Lower ASCII • Date (YYYY-MM-DD, DD/MM/YYYY, MM.DD.YYYY) • Datetime (YYYY-MM-DD HH:MM:SS) • Decimal • Unicode (Gen2) • Unicode (Legacy) • Unicode (Base64 - Encoded Byte's Charset should match Dataelement's Encoding Type) • Email					

Note: The protection methods, that are not mentioned in the *Supported Protection Methods* table, are not supported.

3.9.24 ptyReprotectUnicode()

This UDF reprotects *string* format protected data, which was protected earlier using the *ptyProtectUnicode* UDF, with a different data element.

```
ptyReprotectUnicode(String colName, String oldDataElement, String newDataElement)
```

Parameters

`colName` : The column that contains data in *String* format, which needs to be reprotected

`oldDataElement` : The data element that was used to protect the data earlier

`newDataElement` : The new data element that will be used to reprotect the data

Result

This UDF returns *string* format data, which is protected.

Example

```
import sqlContext.implicits._

val df = sc.parallelize(List("Marylène", "Ã")).toDF("unicode_col")

val reprotectUnicodeUDF = sqlContext.udf.register(
  "ptyReprotectUnicode",
  com.protegrity.spark.udf.ptyReprotectUnicode _)

df.registerTempTable("unicode_test")

sqlContext
  .sql(
```

```
"select ptyReprotectUnicode(unicode_col, 'Token_Unicode', 'Token_Unicode_1') as
reprotected from unicode_test")
  .show(false)
```

Table 3-136: Supported Protection Methods

Spark SQL UDFs	Supported Protection Methods					
	Tokenization	Encryption	FPE	No Encryption	Masking	Monitoring
ptyReprotectUnicode()	• Unicode (Legacy) • Unicode Base64	No	No	Yes	No	Yes

Note: The protection methods, that are not mentioned in the *Supported Protection Methods* table, are not supported.

3.9.25 ptyReprotectInt()

This UDF reprotects *integer* format protected data with a different data element.

```
ptyReprotectInt(Int colName, String oldDataElement, String newDataElement)
```

Parameters

`colName` : The column that contains data in *Integer* format, which needs to be reprotected

`oldDataElement` : The data element that was used to protect the data earlier

`newDataElement` : The new data element that will be used to reprotect the data

Result

This UDF returns *Integer* format data, which is protected.

Example

```
import sqlContext.implicits._

val df = sc.parallelize(List(1234, 2345)).toDF("int_col")

val reprotectIntUDF = sqlContext.udf
  .register("ptyReprotectInt", com.protegrity.spark.udf.ptyReprotectInt _)

df.registerTempTable("int_test")

sqlContext
  .sql(
"select ptyReprotectInt(int_col, 'Token_Int', ' Token_Int_1') as reprotected from
int_test")
  .show(false)
```

Table 3-137: Supported Protection Methods

Spark SQL UDFs	Supported Protection Methods					
	Tokenization	Encryption	FPE	No Encryption	Masking	Monitoring
ptyReprotectInt()	Integer 4 Bytes	No	No	Yes	No	Yes

Note: The protection methods, that are not mentioned in the *Supported Protection Methods* table, are not supported.

3.9.26 ptyReprotectShort()

This UDF reprotects *short* format protected data with a different data element.

```
ptyReprotectShort(Short colName, String oldDataElement, String newDataElement)
```

Parameters

`colName` : The column that contains data in *Integer* format, which needs to be reprotected

`oldDataElement` : The data element that was used to protect the data earlier

`newDataElement` : The new data element that will be used to reprotect the data

Result

This UDF returns *Integer* format data, which is protected.

Example

```
import sqlContext.implicits._

val df = sc.parallelize(List(1234, 2345)).map(x =>
ShortClass(x.toShort)).toDF("short_col")

val reprotectShortUDF = sqlContext.udf.register(
  "ptyReprotectShort",
  com.protegrity.spark.udf.ptyReprotectShort _)

df.registerTempTable("short_test")

sqlContext
  .sql(
"select ptyReprotectShort(short_col, 'Token_Short', ' Token_Short_1') as reprotected
from short_test")
  .show(false)
```

Table 3-138: Supported Protection Methods

Spark SQL UDFs	Supported Protection Methods					
	Tokenization	Encryption	FPE	No Encryption	Masking	Monitoring
ptyReprotectShort()	Integer 2 Bytes	No	No	Yes	No	Yes

Note: The protection methods, that are not mentioned in the *Supported Protection Methods* table, are not supported.

3.9.27 ptyReprotectLong()

This UDF reprotects *long* format protected data with a different data element.

`ptyReprotectLong(Long colName, String oldDataElement, String newDataElement)`

Parameters

`colName` : The column that contains data in *Long* format, which needs to be reprotected

`oldDataElement` : The data element that was used to protect the data earlier

`newDataElement` : The new data element that will be used to reprotect the data

Result

This UDF returns *Long* format data, which is protected.

Example

```
import sqlContext.implicits._

val df = sc.parallelize(List(1234l, 2345l)).toDF("long_col")

val reprotectLongUDF = sqlContext.udf.register(
  "ptyReprotectLong",
  com.protegrity.spark.udf.ptyReprotectLong _)

df.registerTempTable("long_test")

sqlContext
  .sql(
"select ptyReprotectLong(long_col, 'Token_Long', 'Token_Long_1') as reprotected from
```

```
long_test")
  .show(false)
```

Table 3-139: Supported Protection Methods

Spark SQL UDFs	Supported Protection Methods					
	Tokenization	Encryption	FPE	No Encryption	Masking	Monitoring
ptyReprotectLong()	Integer 8 Bytes	No	No	Yes	No	Yes

Note: The protection methods, that are not mentioned in the *Supported Protection Methods* table, are not supported.

3.9.28 ptyReprotectDate()

This UDF reprotects *date* format protected data with a different data element.

```
ptyReprotectDate(Date colName, String oldDataElement, String newDataElement)
```

Note: In the Big Data Protector, version 7.0 release, the *date* format supported is *YYYY-MM-DD* only.

Parameters

`colName` : The column that contains data in *date* format, which needs to be reprotected

`oldDataElement` : The data element that was used to protect the data earlier

`newDataElement` : The new data element that will be used to reprotect the data

Result

This UDF returns *date* format data, which is protected.

Example

```
import sqlContext.implicits._

val d1 = Date.valueOf("2016-12-28")
val d2 = Date.valueOf("2016-12-28")

val df = sc.parallelize(Seq((d1, d2))).toDF("date_col1", "date_col2")

val reprotectDateUDF = sqlContext.udf.register(
  "ptyReprotectDate",
  com.protegrity.spark.udf.ptyReprotectDate _)

df.registerTempTable("date_test")

sqlContext
  .sql(
"select ptyReprotectDate(date_col1, 'Token_Date', 'Token_Date_1') as reprotected from
date_test")
  .show(false)
```

Table 3-140: Supported Protection Methods

Spark SQL UDFs	Supported Protection Methods					
	Tokenization	Encryption	FPE	No Encryption	Masking	Monitoring
ptyReprotectDate()	Date (YYYY-MM-DD, DD/MM/YYYY, MM.DD.YYYY)	No	No	Yes	No	Yes

Note: The protection methods, that are not mentioned in the *Supported Protection Methods* table, are not supported.

3.9.29 ptyReprotectDateTime()

This UDF reprotects *timestamp* format protected data with a different data element.

```
ptyReprotectDateTime(Timestamp colName, String oldDataElement, String
newDataElement)
```

Parameters

`colName` : The column that contains data in *Timestamp* format, which needs to be reprotected

`oldDataElement` : The data element that was used to protect the data earlier

`newDataElement` : The new data element that will be used to reprotect the data

Result

This UDF returns *Timestamp* format data, which is protected.

Example

```
import sqlContext.implicits._

val d1 = Timestamp.valueOf("2016-12-28 13:09:38.104")
val d2 = Timestamp.valueOf("2016-12-29 12:09:38.104")

val df = sc.parallelize(Seq((d1, d2))).toDF("datetime_col1", "datetime_col2")

val reprotectDateTimeUDF = sqlContext.udf.register(
  "ptyReprotectDateTime",
  com.protegrity.spark.udf.ptyReprotectDateTime _)

df.registerTempTable("datetime_test")

sqlContext
  .sql(
"select ptyReprotectDateTime(datetime_col1, 'Token_Datetime', 'Token_Datetime_1') as
reprotected from datetime_test")
  .show(false)
```

Table 3-141: Supported Protection Methods

Spark SQL UDFs	Supported Protection Methods					
	Tokenization	**Encryption**	**FPE**	**No Encryption**	**Masking**	**Monitoring**
ptyReprotectDateTime()	Datetime (YYYY-MM-DD HH:MM:SS)	No	No	Yes	No	Yes

Note: The protection methods, that are not mentioned in the *Supported Protection Methods* table, are not supported.

3.9.30 ptyReprotectFloat()

This UDF reprotects *float* format protected data with a different data element.

```
ptyReprotectFloat(Float colName, String oldDataElement, String newDataElement)
```

Parameters

`colName` : The column that contains data in *Float* format, which needs to be reprotected

`oldDataElement` : The data element that was used to protect the data earlier

`newDataElement` : The new data element that will be used to reprotect the data

Warning: Ensure that you use the *No Encryption* data element only. Using any other data element might cause corruption of data.

Result

This UDF returns *Float* format data, which is protected.

Example

```
import sqlContext.implicits._

val input = Seq((1234.345f, 1343.3345f))

val df = sc.parallelize(input).toDF("float_col1", "float_col2")

val reprotectFloatUDF = sqlContext.udf.register(
  "ptyReprotectFloat",
  com.protegrity.spark.udf.ptyReprotectFloat _)

df.registerTempTable("float_test")

sqlContext
  .sql(
"select ptyReprotectFloat(float_col1, 'Token_NoEncryption', 'Token_NoEncryption') as
reprotected from float_test")
  .show(false)
```

Table 3-142: Supported Protection Methods

Spark SQL UDFs	Supported Protection Methods					
	Tokenization	Encryption	FPE	No Encryption	Masking	Monitoring
ptyReprotectFloat()	No	No	No	Yes	No	Yes

Note: The protection methods, that are not mentioned in the *Supported Protection Methods* table, are not supported.

3.9.31 ptyReprotectDouble()

This UDF reprotects *double* format protected data with a different data element.

```
ptyReprotectDouble(Double colName, String oldDataElement, String newDataElement)
```

Parameters

`colName` : The column that contains data in *Double* format, which needs to be reprotected

`oldDataElement` : The data element that was used to protect the data earlier

`newDataElement` : The new data element that will be used to reprotect the data

Warning: Ensure that you use the *No Encryption* data element only. Using any other data element might cause corruption of data.

Result

This UDF returns *Double* format data, which is protected.

Example

```
import sqlContext.implicits._

val input = Seq((1234.345, 1343.3345))

val df = sc.parallelize(input).toDF("double_col1", "double_col2")

val reprotectDoubleUDF = sqlContext.udf.register(
  "ptyReprotectDouble",
  com.protegrity.spark.udf.ptyReprotectDouble _)

df.registerTempTable("double_test")

sqlContext
  .sql(
 "select ptyReprotectDouble(double_col1, 'Token_NoEncryption', 'Token_NoEncryption') as
```

```
reprotected from double_test")
  .show(false)
```

Table 3-143: Supported Protection Methods

Spark SQL UDFs	Supported Protection Methods					
	Tokenization	Encryption	FPE	No Encryption	Masking	Monitoring
ptyReprotectDouble( )	No	No	No	Yes	No	Yes

Note: The protection methods, that are not mentioned in the *Supported Protection Methods* table, are not supported.

3.9.32 ptyReprotectDecimal()

This UDF reprotects *decimal* format protected data with a different data element.

`ptyReprotectDecimal(BigDecimal colName, String oldDataElement, String newDataElement)`

Parameters

`colName` : The column that contains data in *BigDecimal* format, which needs to be reprotected

`oldDataElement` : The data element that was used to protect the data earlier

`newDataElement` : The new data element that will be used to reprotect the data

Warning: Ensure that you use the *No Encryption* data element only. Using any other data element might cause corruption of data.

Caution: Before the *ptyReprotectDecimal()* UDF is called, Spark SQL rounds off the decimal value in the table to 18 digits in scale, irrespective of the length of the data.

Result

This UDF returns *BigDecimal* format data, which is protected.

Example

```
import sqlContext.implicits._

val input = Seq(
  (math.BigDecimal.valueOf(1234.345), math.BigDecimal.valueOf(1343.3345)))

val df = sc.parallelize(input).toDF("decimal_col1", "decimal_col2")

val reprotectDecimalUDF = sqlContext.udf.register(
  "ptyReprotectDecimal",
  com.protegrity.spark.udf.ptyReprotectDecimal _)

df.registerTempTable("decimal_test")

sqlContext
  .sql(
"select ptyReprotectDecimal(decimal_col1, 'Token_NoEncryption', 'Token_NoEncryption')
as reprotected from decimal_test")
  .show(false)
```

Table 3-144: Supported Protection Methods

Spark SQL UDFs	Supported Protection Methods					
	Tokenization	Encryption	FPE	No Encryption	Masking	Monitoring
ptyReprotectDecimal()	No	No	No	Yes	No	Yes

Note: The protection methods, that are not mentioned in the *Supported Protection Methods* table, are not supported.

3.9.33 ptyStringEnc()

This UDF encrypts a *string* value.

ptyStringEnc(String input, String DataElement)

Parameters

`String input:` *String* value to encrypt

`String DataElement:` Name of the data element to encrypt *string* value

Result

This UDF returns an encrypted *binary* value.

Note: To store the binary output of *ptyStringEnc* UDF in a string column, use the built-in Base64 Spark SQL function to convert the output encrypted bytes into a Base64 encoded string.

Example

```
import org.apache.spark.sql.SQLContext
val sqlContext = new SQLContext(sc)
import sqlContext.implicits._

val protectStrEncUDF =
sqlContext.udf.register("ptyStringEnc",com.protegrity.spark.udf.ptyStringEnc _)

val pepTest = sc.parallelize(List("hello", "world")).toDF("col1")
pepTest.registerTempTable("spark_clear_table")

val encr_spark = sqlContext.sql("select base64(ptyStringEnc(col1,'AES128_CRC')) as col1
spark_clear_table").toDF()
encr_spark.show()
encr_spark.registerTempTable("encrypted_spark")
```

Exception

`java.lang.OutOfMemoryError: Requested array size exceeds VM limit`: The length of the input needs to be less than the maximum limit of 512 MB.

Table 3-145: Supported Protection Methods

Spark SQL UDFs	Supported Protection Methods					
	Tokenization	Encryption	FPE	No Encryption	Masking	Monitoring
ptyStringEnc()	No	• AES-128 • AES-256 • 3DES • CUSP	No	Yes	No	Yes

Note: The protection methods that are not mentioned in the *Supported Protection Methods* table are not supported.

3.9.33.1 Guidelines for Estimating the Field Size of Data

The encryption algorithm and the field sizes (in bytes) required by the features, such as, Key ID (KID), Initialization Vector (IV), and Integrity Check (CRC) is listed in the following table:

Table 3-146: Encryption Algorithm and Field Sizes Required

Encryption Algorithm	KID (size in Bytes)	IV (size in Bytes)	CRC (size in Bytes)
AES	16	16	4
3DES	8	8	4
CUSP_TRDES	2	N/A	4
CUSP_AES	2	N/A	4

The byte sizes required by the input file and the encryption algorithm with the features selected is listed in the following table:

Table 3-147: Byte sizes for the input file and the encryption algorithm

Encryption Algorithm	Maximum Input size in bytes eligible for Encryption	Maximum Input size in bytes eligible for Decryption and Re-Encryption
3DES	Less than <= 535000000 Approx 512 MB	Less than <= 715120000 Approx 682 MB
AES-128		
AES-256		
CUSP 3DES		
CUSP AES-128		
CUSP AES-256		

3.9.34 ptyStringDec()

This UDF decrypts a *binary* value.

ptyStringDec(Binary input, String DataElement)

Parameters

`Binary input:` Protected *Binary* value to unprotect

`String DataElement:` Name of data element that was used to encrypt the *string* value, to decrypt the *binary* value

Result

This UDF returns the decrypted *string* value.

> **Note:** If you have previously stored the encrypted bytes as a Base64-encoded string, then decode them using the unbase64 Spark SQL built-in function before passing to the *ptyStringDec* UDF.

Example

```
import org.apache.spark.sql.SQLContext
val sqlContext = new SQLContext(sc)
import sqlContext.implicits._

val protectStrDecUDF =
sqlContext.udf.register("ptyStringDec",com.protegrity.spark.udf.ptyStringDec _)

val decyrpt_spark = sqlContext.sql("select ptyStringDec(unbase64(col1),'AES128_CRC') as
col1 from encrypted_spark").toDF()
decyrpt_spark.show()
```

Exception

`java.lang.OutOfMemoryError: Requested array size exceeds VM limit`: The length of the input needs to be less than the maximum limit of 512 MB.

Table 3-148: Supported Protection Methods

Spark SQL UDFs	Supported Protection Methods					
	Tokenization	Encryption	FPE	No Encryption	Masking	Monitoring
ptyStringDec()	No	• AES-128 • AES-256 • 3DES • CUSP	No	Yes	No	Yes

Note: The protection methods that are not mentioned in the *Supported Protection Methods* table are not supported.

3.9.35 ptyStringReEnc()

This UDF re-encrypts the *Binary* format encrypted data with a different data element.

ptyStringReEnc(Binary input, String oldDataElement, String newDataElement)

Parameters

`Binary input:` *Binary* value to re-encrypt

`String oldDataElement:` Name of data element used to encrypt the data earlier

`String newDataElement:` Name of new data element to re-encrypt the data

Result

This UDF returns *binary* format data, which is re-encrypted.

Note:

- If you have previously stored the encrypted bytes as a Base64 encoded string, then decode them using the unbase64 Spark SQL built-in function before passing to the *ptyStringReEnc* UDF.
- To store the Binary output of the *ptyStringReEnc* UDF in a String column, use the Base64 Spark SQL built-in function to convert the output re-encrypted bytes into a Base64 encoded string.

Example

```
import org.apache.spark.sql.SQLContext
val sqlContext = new SQLContext(sc)
import sqlContext.implicits._

val protectStrReEncUDF =
sqlContext.udf.register("ptyStringReEnc",com.protegrity.spark.udf.ptyStringReEnc _)

val reencyrpt_spark = sqlContext.sql("select
base64(ptyStringReEnc(unbase64(col1),'AES128_CRC','AES128_CRC')) as col1 from
encrypted_spark").toDF()
reencyrpt_spark.show()
```

Exception

`java.lang.OutOfMemoryError: Requested array size exceeds VM limit`: The length of the input needs to be less than the maximum limit of 512 MB.

Table 3-149: Supported Protection Methods

Spark SQL UDFs	Supported Protection Methods					
	Tokenization	Encryption	FPE	No Encryption	Masking	Monitoring
ptyStringReEnc()	No	• AES-128	No	Yes	No	Yes

Spark SQL UDFs	Supported Protection Methods					
	Tokenization	Encryption	FPE	No Encryption	Masking	Monitoring
		• AES-256 • 3DES • CUSP				

Note: The protection methods that are not mentioned in the *Supported Protection Methods* table are not supported.

3.10 PySpark - Scala Wrapper UDFs

This section describes the Scala Wrapper User Defined Functions (UDFs) that are available for protection and unprotection in Big Data Protector to build secure Big Data applications.

For each of the Spark SQL UDF in section *Spark SQL UDFs*, a Scala UDF wrapper class is created so that it can be registered in the PySpark and invoked using the *spark.sql()* method.

Caution: The Scala Wrapper UDFs for PySpark are only available for the Databricks environment.

3.10.1 ptyGetVersionScalaWrapper()

This UDF returns the current version of PEP.

ptyGetVersionScalaWrapper()

Parameters

None

Result

This UDF returns the current version of PEP.

Example

```
spark.udf.registerJavaFunction("ptyGetVersionScalaWrapper",
"com.protegrity.spark.wrapper.ptyGetVersion")
spark.sql("select ptyGetVersionScalaWrapper()").show(truncate = False)
```

3.10.2 ptyWhoAmIScalaWrapper()

This UDF returns the current logged in user.

ptyWhoAmIScalaWrapper()

Parameters

None

Result

This UDF returns the current logged in user.

Example

```
spark.udf.registerJavaFunction("ptyWhoAmIScalaWrapper",
"com.protegrity.spark.wrapper.ptyWhoAmI")
spark.sql("select ptyWhoAmIScalaWrapper()").show(truncate = False)
```

3.10.3 ptyProtectStrScalaWrapper()

This UDF protects *string* format data, which is provided as input.

Note:

- For Date and Datetime type of data elements, the protect API returns an *invalid input data* error if the input value falls between the non-existent date range from 05-OCT-1582 to 14-OCT-1582 of the Gregorian Calendar.
- For more information about the tokenization and de-tokenization of the cutover dates of the Proleptic Gregorian Calendar, refer to the *Date* and *Datetime* tokenization sections in the *Protection Method Reference Guide 9.1.0.0*.

ptyProtectStrScalaWrapper(String colName, String dataElement)

Parameters

`colName` : The column that contains data in *String* format, which needs to be protected

`dataElement` : The data element that will be used to protect *string* format data

Result

This UDF returns *string* format data, which is protected.

Example

```
from pyspark.sql.types import *
spark.udf.registerJavaFunction("ptyProtectStrScalaWrapper",
"com.protegrity.spark.wrapper.ptyProtectStr", StringType())
spark.sql("select ptyProtectStrScalaWrapper(column1, 'Data_Element') from
table1;").show(truncate = False)
```

3.10.4 ptyProtectUnicodeScalaWrapper()

This UDF protects *string* (Unicode) format data, which is provided as input.

ptyProtectUnicodeScalaWrapper(String colName, String dataElement)

Parameters

`colName` : The column that contains data in *String* (Unicode) format, which needs to be protected

`dataElement` : The data element that will be used to protect *string* (Unicode) format data

Result

This UDF returns *string* format data, which is protected.

Example

```
from pyspark.sql.types import *
spark.udf.registerJavaFunction("ptyProtectUnicodeScalaWrapper",
"com.protegrity.spark.wrapper.ptyProtectUnicode", StringType())
spark.sql("select ptyProtectUnicodeScalaWrapper(column1, 'Data_Element') from
table1;").show(truncate = False)
```

3.10.5 ptyProtectIntScalaWrapper()

This UDF protects *integer* format data, which is provided as input.

ptyProtectIntScalaWrapper(Int input, String dataElement)

Parameters

`colName` : The column that contains data in *Integer* format, which needs to be protected

`dataElement` : The data element that will be used to protect *string* (Unicode) format data

Result

This UDF returns *string* format data, which is protected.

Example

```
from pyspark.sql.types import *
spark.udf.registerJavaFunction("ptyProtectIntScalaWrapper",
"com.protegrity.spark.wrapper.ptyProtectInt", IntegerType())
spark.sql("select ptyProtectIntScalaWrapper(column1, 'Data_Element') from
table1;").show(truncate = False)
```

3.10.6 ptyProtectShortScalaWrapper()

This UDF protects *Short* format data, which is provided as input.

ptyProtectShortScalaWrapper(Short colName, String dataElement)

Parameters

`colName` : The column that contains data in *Short* format, which needs to be protected

`dataElement` : The data element that will be used to protect *short* format data

Result

This UDF returns *short* format data, which is protected.

Example

```
from pyspark.sql.types import *
spark.udf.registerJavaFunction("ptyProtectShortScalaWrapper",
"com.protegrity.spark.wrapper.ptyProtectShort", ShortType())
spark.sql("select ptyProtectShortScalaWrapper(column1, 'Data_Element') from
table1;").show(truncate = False)
```

3.10.7 ptyProtectLongScalaWrapper()

This UDF protects *Long* format data, which is provided as input.

ptyProtectLongScalaWrapper(Long colName, String dataElement)

Parameters

`colName` : The column that contains data in *Long* format, which needs to be protected

`dataElement` : The data element that will be used to protect *long* format data

Result

This UDF returns *long* format data, which is protected.

Example

```
from pyspark.sql.types import *
spark.udf.registerJavaFunction("ptyProtectLongScalaWrapper",
"com.protegrity.spark.wrapper.ptyProtectLong", LongType())
spark.sql("select ptyProtectLongScalaWrapper(column1, 'Data_Element') from
table1;").show(truncate = False)
```

3.10.8 ptyProtectDateScalaWrapper()

This UDF protects *date* format data, which is provided as input.

ptyProtectDateScalaWrapper(Date colName, String dataElement)

> **Note:** Starting with the Big Data Protector, version 7.0 release, the *date* format supported is *YYYY-MM-DD* only.

Parameters

`colName` : The column that contains data in *Date* format, which needs to be protected

`dataElement` : The data element that will be used to protect *date* format data

Result

This UDF returns *date* format data, which is protected.

Example

```
from pyspark.sql.types import *
spark.udf.registerJavaFunction("ptyProtectDateScalaWrapper",
"com.protegrity.spark.wrapper.ptyProtectDate", DateType())
spark.sql("select ptyProtectDateScalaWrapper(column1, 'Data_Element') from
table1;").show(truncate = False)
```

3.10.9 ptyProtectDateTimeScalaWrapper()

This UDF protects the *timestamp* format data, which is provided as input.

ptyProtectDateTimeScalaWrapper(Timestamp colName, String dataElement)

Parameters

`colName` : The column that contains data in the *Timestamp* format, which needs to be protected

`dataElement` : The data element that will be used to protect the *timestamp* format data

Result

This UDF returns *timestamp* format data, which is protected.

Example

```
from pyspark.sql.types import *
spark.udf.registerJavaFunction("ptyProtectDateTimeScalaWrapper",
"com.protegrity.spark.wrapper.ptyProtectDateTime", TimestampType())
spark.sql("select ptyProtectDateTimeScalaWrapper(column1, 'Data_Element') from
table1;").show(truncate = False)
```

3.10.10 ptyProtectFloatScalaWrapper()

This UDF protects *float* format data, which is provided as input.

Caution:

- The Float, Double, and Decimal UDFs will be deprecated in a future version of the Big Data Protector and should not be used.
- It is recommended not to use the Float or Double or Decimal data type directly in the Float or Double or Decimal UDFs of Protegrity.
- If you want to protect the Decimal data type, then convert the Decimal data to String data type and pass the Decimal converted String data type to the *ptyProtectStrScalaWrapper()* UDF with the Decimal tokenizer. Ensure that the right precision and scale of input data are maintained during conversion.
- If there is a Decimal datatype UDF with the Decimal input, then convert the Decimal to string data type and pass the Decimal converted string data type to *ptyProtectStrScalaWrapper()* UDF with the decimal tokenizer.

Warning: Protegrity will not be responsible for any type of data conversion error that might occur during conversion.

ptyProtectFloatScalaWrapper(Float colName, String dataElement)

Parameters

`colName` : The column that contains data in *Float* format, which needs to be protected

`dataElement` : The data element that will be used to protect *float* format data

Warning: Ensure that you use the *No Encryption* data element only. Using any other data element might cause corruption of data.

Result

This UDF returns *float* format data, which is protected.

Example

```
from pyspark.sql.types import *
spark.udf.registerJavaFunction("ptyProtectFloatScalaWrapper",
"com.protegrity.spark.wrapper.ptyProtectFloat", FloatType())
spark.sql("select ptyProtectFloatScalaWrapper(column1, 'Data_Element') from
table1;").show(truncate = False)
```

3.10.11 ptyProtectDoubleScalaWrapper()

This UDF protects *double* format data, which is provided as input.

Caution:

- The Float, Double, and Decimal UDFs will be deprecated in a future version of the Big Data Protector and should not be used.
- It is recommended not to use the Float or Double or Decimal data type directly in the Float or Double or Decimal UDFs of Protegrity.
- If you want to protect the Decimal data type, then convert the Decimal data to String data type and pass the Decimal converted String data type to the *ptyProtectStrScalaWrapper()* UDF with the Decimal tokenizer. Ensure that the right precision and scale of input data are maintained during conversion.
- If there is a Decimal datatype UDF with the Decimal input, then convert the Decimal to string data type and pass the Decimal converted string data type to *ptyProtectStrScalaWrapper()* UDF with the decimal tokenizer.

Warning: Protegrity will not be responsible for any type of data conversion error that might occur during conversion.

ptyProtectDoubleScalaWrapper(Double colName, String dataElement)

Parameters

`colName` : The column that contains data in *Double* format, which needs to be protected

`dataElement` : The data element that will be used to protect the *double* format data

Warning: Ensure that you use the *No Encryption* data element only. Using any other data element might cause corruption of data.

Result

This UDF returns *BigDecimal* format data, which is protected.

Example

```
from pyspark.sql.types import *
spark.udf.registerJavaFunction("ptyProtectDoubleScalaWrapper",
"com.protegrity.spark.wrapper.ptyProtectDouble", DoubleType())
spark.sql("select ptyProtectDoubleScalaWrapper(column1, 'Data_Element') from
table1;").show(truncate = False)
```

3.10.12 ptyProtectDecimalScalaWrapper()

This UDF protects *decimal* format data, which is provided as input.

Caution:

- The Float, Double, and Decimal UDFs will be deprecated in a future version of the Big Data Protector and should not be used.
- It is recommended not to use the Float or Double or Decimal data type directly in the Float or Double or Decimal UDFs of Protegrity.
- If you want to protect the Decimal data type, then convert the Decimal data to String data type and pass the Decimal converted String data type to the *ptyProtectStrScalaWrapper()* UDF with the Decimal tokenizer. Ensure that the right precision and scale of input data are maintained during conversion.

- If there is a Decimal datatype UDF with the Decimal input, then convert the Decimal to string data type and pass the Decimal converted string data type to *ptyProtectStrScalaWrapper()* UDF with the decimal tokenizer.

Warning: Protegrity will not be responsible for any type of data conversion error that might occur during conversion.

ptyProtectDecimalScalaWrapper(BigDecimal colName, String dataElement)

Parameters

`colName` : The column that contains data in *BigDecimal* format data, which needs to be protected

`dataElement` : The data element that will be used to protect *BigDecimal* format data

Warning: Ensure that you use the *No Encryption* data element only. Using any other data element might cause corruption of data.

Caution: Before the *ptyProtectDecimalScalaWrapper()* UDF is called, Spark SQL rounds off the decimal value in the table to 18 digits in scale, irrespective of the length of the data.

Result

This UDF returns *BigDecimal* format data, which is protected.

Example

```
from pyspark.sql.types import *
spark.udf.registerJavaFunction("ptyProtectDecimalScalaWrapper",
"com.protegrity.spark.wrapper.ptyProtectDecimal", DecimalType(precision=10, scale=4))
spark.sql("select ptyProtectDecimalScalaWrapper(column1, 'Data_Element') from
table1;").show(truncate = False)
```

3.10.13 ptyUnprotectStrScalaWrapper()

This UDF unprotects the protected string **format** data, **which is provided as input**.

Note: For Date and Datetime type of data elements, the protect API returns an *invalid input data* error if the input value falls between the non-existent date range from 05-OCT-1582 to 14-OCT-1582 of the Gregorian Calendar.

For more information about the tokenization and de-tokenization of the cutover dates of the Proleptic Gregorian Calendar, refer to the *Date* and *Datetime* tokenization sections in the *Protection Method Reference Guide 9.1.0.0*.

ptyUnprotectStrScalaWrapper(String colName, String dataElement)

Parameters

`colName` : The column that contains data in *string* format, which needs to be unprotected

`dataElement` : The data element that will be used to unprotected *string* format data

Warning: Ensure that you use the *No Encryption* data element only. Using any other data element might cause corruption of data.

Result

This UDF returns *string* format data, which is unprotected.

Example

```
from pyspark.sql.types import *
spark.udf.registerJavaFunction("ptyUnprotectStrScalaWrapper",
"com.protegrity.spark.wrapper.ptyUnprotectStr", StringType())
spark.sql("select ptyUnprotectStrScalaWrapper(column1, 'Data_Element') from
table1;").show(truncate = False)
```

3.10.14 ptyUnprotectUnicodeScalaWrapper()

This UDF unprotects the protected string **format** data, **which is provided as input**.

ptyUnprotectUnicodeScalaWrapper(String colName, String dataElement)

Parameters

`colName` : The column that contains data in protected *string* format, which needs to be unprotected

`dataElement` : The data element that will be used to unprotected *string* format data

> **Warning:** Ensure that you use the *No Encryption* data element only. Using any other data element might cause corruption of data.

Result

This UDF returns *string* (Unicode) format data, which is unprotected.

Example

```
from pyspark.sql.types import *
spark.udf.registerJavaFunction("ptyUnprotectUnicodeScalaWrapper",
"com.protegrity.spark.wrapper.ptyUnprotectUnicode", StringType())
spark.sql("select ptyUnprotectUnicodeScalaWrapper(column1, 'Data_Element') from
table1;").show(truncate = False)
```

3.10.15 ptyUnprotectIntScalaWrapper()

This UDF unprotects the protected integer format data, which is provided as input.

ptyUnprotectIntScalaWrapper(Int colName, String dataElement)

Parameters

`colName` : The column that contains data in protected *integer* format, which needs to be unprotected

`dataElement` : The data element that will be used to unprotected *integer* format data

> **Caution:** If an unauthorized user, with no privileges to unprotect data in the security policy, and the output value set to NULL, attempts to unprotect the protected data of Numeric type data containing *Short*, *Int*, *Float*, *Long*, *Double*, and *Decimal* format values using the respective Spark SQL UDFs, then the output is *0*.

Result

This UDF returns *integer* (Unicode) format data, which is unprotected.

Example

```
from pyspark.sql.types import *
spark.udf.registerJavaFunction("ptyUnprotectIntScalaWrapper",
"com.protegrity.spark.wrapper.ptyUnprotectInt", IntegerType())
spark.sql("select ptyUnprotectIntScalaWrapper(column1, 'Data_Element') from
table1;").show(truncate = False)
```

3.10.16 ptyUnprotectShortScalaWrapper()

This UDF unprotects the protected short **format** data, **which is provided as input**.

ptyUnprotectShortScalaWrapper(Short colName, String dataElement)

Parameters

`colName` : The column that contains data in protected *short* format, which needs to be unprotected

dataElement : The data element that will be used to unprotected *short* format data

> **Caution:** If an unauthorized user, with no privileges to unprotect data in the security policy, and the output value set to NULL, attempts to unprotect the protected data of Numeric type data containing *Short*, *Int*, *Float*, *Long*, *Double*, and *Decimal* format values using the respective Spark SQL UDFs, then the output is *0*.

Result

This UDF returns *short* (Unicode) format data, which is unprotected.

Example

```
from pyspark.sql.types import *
spark.udf.registerJavaFunction("ptyUnprotectShortScalaWrapper",
"com.protegrity.spark.wrapper.ptyUnprotectShort", ShortType())
spark.sql("select ptyUnprotectShortScalaWrapper(column1, 'Data_Element') from
table1;").show(truncate = False)
```

3.10.17 ptyUnprotectLongScalaWrapper()

This UDF unprotects the protected long **format** data, **which is provided as input**.

ptyUnprotectLongScalaWrapper(Long colName, String dataElement)

Parameters

colName : The column that contains data in protected *long* format, which needs to be unprotected

dataElement : The data element that will be used to unprotected *long* format data

> **Caution:** If an unauthorized user, with no privileges to unprotect data in the security policy, and the output value set to NULL, attempts to unprotect the protected data of Numeric type data containing *Short*, *Int*, *Float*, *Long*, *Double*, and *Decimal* format values using the respective Spark SQL UDFs, then the output is *0*.

Result

This UDF returns *long* (Unicode) format data, which is unprotected.

Example

```
from pyspark.sql.types import *
spark.udf.registerJavaFunction("ptyUnprotectLongScalaWrapper",
"com.protegrity.spark.wrapper.ptyUnprotectLong", LongType())
spark.sql("select ptyUnprotectLongScalaWrapper(column1, 'Data_Element') from
table1;").show(truncate = False)
```

3.10.18 ptyUnprotectDateScalaWrapper()

This UDF unprotects the protected date **format** data, **which is provided as input**.

ptyUnprotectDateScalaWrapper(Date colName, String dataElement)

Parameters

colName : The column that contains data in protected *date* format, which needs to be unprotected

dataElement : The data element that will be used to unprotected *date* format data

Result

This UDF returns *date* (Unicode) format data, which is unprotected.

Example

```
from pyspark.sql.types import *
spark.udf.registerJavaFunction("ptyUnprotectDateScalaWrapper",
"com.protegrity.spark.wrapper.ptyUnprotectDate", DateType())
```

```
spark.sql("select ptyUnprotectDateScalaWrapper(column1, 'Data_Element') from
table1;").show(truncate = False)
```

3.10.19 ptyUnprotectDateTimeScalaWrapper()

This UDF unprotects the protected timestamp **format** data, **which is provided as input**.

ptyUnprotectDateTimeScalaWrapper(Timestamp colName, String dataElement)

Parameters

`colName` : The column that contains data in protected *timestamp* format, which needs to be unprotected

`dataElement` : The data element that will be used to unprotected *timestamp* format data

Result

This UDF returns *timestamp* (Unicode) format data, which is unprotected.

Example

```
from pyspark.sql.types import *
spark.udf.registerJavaFunction("ptyUnprotectDateTimeScalaWrapper",
"com.protegrity.spark.wrapper.ptyUnprotectDateTime", TimestampType())
spark.sql("select ptyUnprotectDateTimeScalaWrapper(column1, 'Data_Element') from
table1;").show(truncate = False)
```

3.10.20 ptyUnprotectFloatScalaWrapper()

This UDF unprotects protected *float* format data, which is provided as input.

Caution:

- The Float, Double, and Decimal UDFs will be deprecated in a future version of the Big Data Protector and should not be used.
- It is recommended not to use the Float or Double or Decimal data type directly in the Float or Double or Decimal UDFs of Protegrity.
- If you want to protect the Decimal data type, then convert the Decimal data to String data type and pass the Decimal converted String data type to the *ptyProtectStrScalaWrapper()* UDF with the Decimal tokenizer. Ensure that the right precision and scale of input data are maintained during conversion.
- If there is a Decimal datatype UDF with the Decimal input, then convert the Decimal to string data type and pass the Decimal converted string data type to *ptyProtectStrScalaWrapper()* UDF with the decimal tokenizer.

Warning: Protegrity will not be responsible for any type of data conversion error that might occur during conversion.

ptyUnprotectFloatScalaWrapper(Float colName, String dataElement)

Parameters

`colName` : The column that contains data in protected *float* format, which needs to be unprotected

`dataElement` : The data element that will be used to unprotected *float* format data

Warning: Ensure that you use the *No Encryption* data element only. Using any other data element might cause corruption of data.

Caution: If an unauthorized user, with no privileges to unprotect data in the security policy, and the output value set to NULL, attempts to unprotect the protected data of Numeric type data containing *Short*, *Int*, *Float*, *Long*, *Double*, and *Decimal* format values using the respective Spark SQL UDFs, then the output is *0*.

Result

This UDF returns *float* (Unicode) format data, which is unprotected.

Example

```
from pyspark.sql.types import *
spark.udf.registerJavaFunction("ptyUnprotectFloatScalaWrapper",
"com.protegrity.spark.wrapper.ptyUnprotectFloat", FloatType())
spark.sql("select ptyUnprotectFloatScalaWrapper(column1, 'Data_Element') from
table1;").show(truncate = False)
```

3.10.21 ptyUnprotectDoubleScalaWrapper()

This UDF unprotects protected *double* format data, which is provided as input.

Caution:

- The Float, Double, and Decimal UDFs will be deprecated in a future version of the Big Data Protector and should not be used.
- It is recommended not to use the Float or Double or Decimal data type directly in the Float or Double or Decimal UDFs of Protegrity.
- If you want to protect the Decimal data type, then convert the Decimal data to String data type and pass the Decimal converted String data type to the *ptyProtectStrScalaWrapper()* UDF with the Decimal tokenizer. Ensure that the right precision and scale of input data are maintained during conversion.
- If there is a Decimal datatype UDF with the Decimal input, then convert the Decimal to string data type and pass the Decimal converted string data type to *ptyProtectStrScalaWrapper()* UDF with the decimal tokenizer.

Warning: Protegrity will not be responsible for any type of data conversion error that might occur during conversion.

ptyUnprotectDoubleScalaWrapper(Double colName, String dataElement)

Parameters

`colName` : The column that contains data in protected *double* format, which needs to be unprotected

`dataElement` : The data element that will be used to unprotected *double* format data

Warning: Ensure that you use the *No Encryption* data element only. Using any other data element might cause corruption of data.

Caution: If an unauthorized user, with no privileges to unprotect data in the security policy, and the output value set to NULL, attempts to unprotect the protected data of Numeric type data containing *Short*, *Int*, *Float*, *Long*, *Double*, and *Decimal* format values using the respective Spark SQL UDFs, then the output is *0*.

Result

This UDF returns *double* (Unicode) format data, which is unprotected.

Example

```
from pyspark.sql.types import *
spark.udf.registerJavaFunction("ptyUnprotectDoubleScalaWrapper",
"com.protegrity.spark.wrapper.ptyUnprotectDouble", DoubleType())
spark.sql("select ptyUnprotectDoubleScalaWrapper(column1, 'Data_Element') from
table1;").show(truncate = False)
```

3.10.22 ptyUnprotectDecimalScalaWrapper()

This UDF unprotects protected *decimal* data, which is provided as input.

Caution:

- The Float, Double, and Decimal UDFs will be deprecated in a future version of the Big Data Protector and should not be used.
- It is recommended not to use the Float or Double or Decimal data type directly in the Float or Double or Decimal UDFs of Protegrity.

- If you want to protect the Decimal data type, then convert the Decimal data to String data type and pass the Decimal converted String data type to the *ptyProtectStrScalaWrapper()* UDF with the Decimal tokenizer. Ensure that the right precision and scale of input data are maintained during conversion.
- If there is a Decimal datatype UDF with the Decimal input, then convert the Decimal to string data type and pass the Decimal converted string data type to *ptyProtectStrScalaWrapper()* UDF with the decimal tokenizer.

Warning: Protegrity will not be responsible for any type of data conversion error that might occur during conversion.

ptyUnprotectDecimalScalaWrapper(BigDecimal colName, String dataElement)

Parameters

`colName` : The column that contains data in protected *BigDecimal* format, which needs to be unprotected

`dataElement` : The data element that will be used to unprotected *BigDecimal* format data

Warning: Ensure that you use the *No Encryption* data element only. Using any other data element might cause corruption of data.

Caution: Before the *ptyUnprotectDecimal()* UDF is called, Spark SQL rounds off the decimal value in the table to 18 digits in scale, irrespective of the length of the data.

Caution: If an unauthorized user, with no privileges to unprotect data in the security policy, and the output value set to NULL, attempts to unprotect the protected data of Numeric type data containing *Short*, *Int*, *Float*, *Long*, *Double*, and *Decimal* format values using the respective Spark SQL UDFs, then the output is *0*.

Result

This UDF returns *BigDecimal* (Unicode) format data, which is unprotected.

Example

```
from pyspark.sql.types import *
spark.udf.registerJavaFunction("ptyUnprotectDecimalScalaWrapper",
"com.protegrity.spark.wrapper.ptyUnprotectDecimal", DecimalType(precision=10, scale=4))
spark.sql("select ptyUnprotectDecimalScalaWrapper(column1, 'Data_Element') from
table1;").show(truncate = False)
```

3.10.23 ptyReprotectStrScalaWrapper()

This UDF reprotects *string* format protected data, which was earlier protected using the *ptyProtectStrScalaWrapper* UDF, with a different data element.

`ptyReprotectStrScalaWrapper(String colName, String oldDataElement, String newDataElement)`

Parameters

`colName` : The column that contains data in *String* format, which needs to be reprotected

`oldDataElement` : The data element that was used to protect the data earlier

`newDataElement` : The new data element that will be used to reprotect the data

Result

This UDF returns *string* format data, which is protected.

Example

```
from pyspark.sql.types import *
spark.udf.registerJavaFunction("ptyReprotectStrScalaWrapper",
"com.protegrity.spark.wrapper.ptyReprotectStr", StringType())
spark.sql("select ptyReprotectStrScalaWrapper(column1, 'Data_Element') from
table1;").show(truncate = False)
```

3.10.24 ptyReprotectUnicodeScalaWrapper()

This UDF reprotects *string* format protected data, which was protected earlier using the *ptyProtectUnicodeScalaWrapper* UDF, with a different data element.

```
ptyReprotectUnicodeScalaWrapper(String colName, String oldDataElement, String
newDataElement)
```

Parameters

`colName` : The column that contains data in *String* format, which needs to be reprotected

`oldDataElement` : The data element that was used to protect the data earlier

`newDataElement` : The new data element that will be used to reprotect the data

Result

This UDF returns *string* format data, which is protected.

Example

```
from pyspark.sql.types import *
spark.udf.registerJavaFunction("ptyReprotectUnicodeScalaWrapper",
"com.protegrity.spark.wrapper.ptyReprotectUnicode", StringType())
spark.sql("select ptyReprotectUnicodeScalaWrapper(column1, 'Data_Element') from
table1;").show(truncate = False)
```

3.10.25 ptyReprotectIntScalaWrapper()

This UDF reprotects *integer* format protected data with a different data element.

```
ptyReprotectIntScalaWrapper(Int colName, String oldDataElement, String
newDataElement)
```

Parameters

`colName` : The column that contains data in *Integer* format, which needs to be reprotected

`oldDataElement` : The data element that was used to protect the data earlier

`newDataElement` : The new data element that will be used to reprotect the data

Result

This UDF returns *Integer* format data, which is protected.

Example

```
from pyspark.sql.types import *
spark.udf.registerJavaFunction("ptyReprotectIntScalaWrapper",
"com.protegrity.spark.wrapper.ptyReprotectInt", IntegerType())
spark.sql("select ptyReprotectIntScalaWrapper(column1, 'Data_Element') from
table1;").show(truncate = False)
```

3.10.26 ptyReprotectShortScalaWrapper()

This UDF reprotects *short* format protected data with a different data element.

```
ptyReprotectShortScalaWrapper(Short colName, String oldDataElement, String
newDataElement)
```

Parameters

`colName` : The column that contains data in *Integer* format, which needs to be reprotected

`oldDataElement` : The data element that was used to protect the data earlier

`newDataElement` : The new data element that will be used to reprotect the data

Result

This UDF returns *Integer* format data, which is protected.

Example

```
from pyspark.sql.types import *
spark.udf.registerJavaFunction("ptyReprotectShortScalaWrapper",
"com.protegrity.spark.wrapper.ptyReprotectShort", ShortType())
spark.sql("select ptyReprotectShortScalaWrapper(column1, 'Data_Element') from
table1;").show(truncate = False)
```

3.10.27 ptyReprotectLongScalaWrapper()

This UDF reprotects *long* format protected data with a different data element.

```
ptyReprotectLongScalaWrapper(Long colName, String oldDataElement, String
newDataElement)
```

Parameters

`colName` : The column that contains data in *Long* format, which needs to be reprotected

`oldDataElement` : The data element that was used to protect the data earlier

`newDataElement` : The new data element that will be used to reprotect the data

Result

This UDF returns *Long* format data, which is protected.

Example

```
from pyspark.sql.types import *
spark.udf.registerJavaFunction("ptyReprotectLongScalaWrapper",
"com.protegrity.spark.wrapper.ptyReprotectLong", LongType())
spark.sql("select ptyReprotectLongScalaWrapper(column1, 'Data_Element') from
table1;").show(truncate = False)
```

3.10.28 ptyReprotectDateScalaWrapper()

This UDF reprotects *date* format protected data with a different data element.

```
ptyReprotectDateScalaWrapper(Date colName, String oldDataElement, String
newDataElement)
```

> **Note:** In the Big Data Protector, version 7.0 release, the *date* format supported is *YYYY-MM-DD* only.

Parameters

`colName` : The column that contains data in *date* format, which needs to be reprotected

`oldDataElement` : The data element that was used to protect the data earlier

`newDataElement` : The new data element that will be used to reprotect the data

Result

This UDF returns *date* format data, which is protected.

Example

```
from pyspark.sql.types import *
spark.udf.registerJavaFunction("ptyReprotectDateScalaWrapper",
"com.protegrity.spark.wrapper.ptyReprotectDate", DateType())
spark.sql("select ptyReprotectDateScalaWrapper(column1, 'Data_Element') from
table1;").show(truncate = False)
```

3.10.29 ptyReprotectDateTimeScalaWrapper()

This UDF reprotects *timestamp* format protected data with a different data element.

`ptyReprotectDateTimeScalaWrapper(Timestamp colName, String oldDataElement, String newDataElement)`

Parameters

`colName` : The column that contains data in *Timestamp* format, which needs to be reprotected

`oldDataElement` : The data element that was used to protect the data earlier

`newDataElement` : The new data element that will be used to reprotect the data

Result

This UDF returns *Timestamp* format data, which is protected.

Example

```
from pyspark.sql.types import *
spark.udf.registerJavaFunction("ptyReprotectDateTimeScalaWrapper",
"com.protegrity.spark.wrapper.ptyReprotectDateTime", TimestampType())
spark.sql("select ptyReprotectDateTimeScalaWrapper(column1, 'Data_Element') from
table1;").show(truncate = False)
```

3.10.30 ptyReprotectFloatScalaWrapper()

This UDF reprotects *float* format protected data with a different data element.

Caution:

- The Float, Double, and Decimal UDFs will be deprecated in a future version of the Big Data Protector and should not be used.
- It is recommended not to use the Float or Double or Decimal data type directly in the Float or Double or Decimal UDFs of Protegrity.
- If you want to protect the Decimal data type, then convert the Decimal data to String data type and pass the Decimal converted String data type to the *ptyProtectStrScalaWrapper()* UDF with the Decimal tokenizer. Ensure that the right precision and scale of input data are maintained during conversion.
- If there is a Decimal datatype UDF with the Decimal input, then convert the Decimal to string data type and pass the Decimal converted string data type to *ptyProtectStrScalaWrapper()* UDF with the decimal tokenizer.

Warning: Protegrity will not be responsible for any type of data conversion error that might occur during conversion.

`ptyReprotectFloatScalaWrapper(Float colName, String oldDataElement, String newDataElement)`

Parameters

`colName` : The column that contains data in *Float* format, which needs to be reprotected

`oldDataElement` : The data element that was used to protect the data earlier

`newDataElement` : The new data element that will be used to reprotect the data

Warning: Ensure that you use the *No Encryption* data element only. Using any other data element might cause corruption of data.

Result

This UDF returns *Float* format data, which is protected.

Example

```
from pyspark.sql.types import *
spark.udf.registerJavaFunction("ptyReprotectFloatScalaWrapper",
"com.protegrity.spark.wrapper.ptyReprotectFloat", FloatType())
spark.sql("select ptyReprotectFloatScalaWrapper(column1, 'Data_Element') from
table1;").show(truncate = False)
```

3.10.31 ptyReprotectDoubleScalaWrapper()

This UDF reprotects *double* format protected data with a different data element.

Caution:

- The Float, Double, and Decimal UDFs will be deprecated in a future version of the Big Data Protector and should not be used.
- It is recommended not to use the Float or Double or Decimal data type directly in the Float or Double or Decimal UDFs of Protegrity.
- If you want to protect the Decimal data type, then convert the Decimal data to String data type and pass the Decimal converted String data type to the *ptyProtectStr()* UDF with the Decimal tokenizer. Ensure that the right precision and scale of input data are maintained during conversion.
- If there is a Decimal datatype UDF with the Decimal input, then convert the Decimal to string data type and pass the Decimal converted string data type to *ptyProtectStr()* UDF with the decimal tokenizer.

Warning: Protegrity will not be responsible for any type of data conversion error that might occur during conversion.

```
ptyReprotectDoubleScalaWrapper(Double colName, String oldDataElement, String
newDataElement)
```

Parameters

colName : The column that contains data in *Double* format, which needs to be reprotected

oldDataElement : The data element that was used to protect the data earlier

newDataElement : The new data element that will be used to reprotect the data

Warning: Ensure that you use the *No Encryption* data element only. Using any other data element might cause corruption of data.

Result

This UDF returns *Double* format data, which is protected.

Example

```
from pyspark.sql.types import *
spark.udf.registerJavaFunction("ptyReprotectDoubleScalaWrapper",
"com.protegrity.spark.wrapper.ptyReprotectDouble", DoubleType())
spark.sql("select ptyReprotectDoubleScalaWrapper(column1, 'Data_Element') from
table1;").show(truncate = False)
```

3.10.32 ptyReprotectDecimalScalaWrapper()

This UDF reprotects *decimal* format protected data with a different data element.

Caution:

- The Float, Double, and Decimal UDFs will be deprecated in a future version of the Big Data Protector and should not be used.
- It is recommended not to use the Float or Double or Decimal data type directly in the Float or Double or Decimal UDFs of Protegrity.
- If you want to protect the Decimal data type, then convert the Decimal data to String data type and pass the Decimal converted String data type to the *ptyProtectStrScalaWrapper()* UDF with the Decimal tokenizer. Ensure that the right precision and scale of input data are maintained during conversion.
- If there is a Decimal datatype UDF with the Decimal input, then convert the Decimal to string data type and pass the Decimal converted string data type to *ptyProtectStrScalaWrapper()* UDF with the decimal tokenizer.

Warning: Protegrity will not be responsible for any type of data conversion error that might occur during conversion.

`ptyReprotectDecimalScalaWrapper(BigDecimal colName, String oldDataElement, String newDataElement)`

Parameters

`colName` : The column that contains data in *BigDecimal* format, which needs to be reprotected

`oldDataElement` : The data element that was used to protect the data earlier

`newDataElement` : The new data element that will be used to reprotect the data

Warning: Ensure that you use the *No Encryption* data element only. Using any other data element might cause corruption of data.

Caution: Before the *ptyReprotectDecimal()* UDF is called, Spark SQL rounds off the decimal value in the table to 18 digits in scale, irrespective of the length of the data.

Result

This UDF returns *BigDecimal* format data, which is protected.

Example

```
from pyspark.sql.types import *
spark.udf.registerJavaFunction("ptyReprotectDecimalScalaWrapper",
"com.protegrity.spark.wrapper.ptyReprotectDecimal", DecimalType(precision=10, scale=4))
spark.sql("select ptyReprotectDecimalScalaWrapper(column1, 'Data_Element') from
table1;").show(truncate = False)
```

3.10.33 ptyStringEncScalaWrapper()

This UDF encrypts *string* value provided as input.

ptyStringEncScalaWrapper(String colName, String dataElement)

Parameters

`colName` : The column that contains data in *String* format, which needs to be encrypted

`dataElement` : The data element in *String* format that will be used to encrypt the data

Result

This UDF returns encrypted binary format data.

Example

```
from pyspark.sql.types import *
spark.udf.registerJavaFunction("ptyStringEncScalaWrapper",
"com.protegrity.spark.wrapper.ptyStringEnc", BinaryType())
spark.sql("select ptyStringEncScalaWrapper (column1, 'Data_Element') from
table1;").show(truncate = False)
```

3.10.34 ptyStringDecScalaWrapper()

This UDF decrypts binary value provided as input.

ptyStringDecScalaWrapper(Binary colName, String dataElement)

Parameters

`colName` : The column that contains data in *Binary* format, which needs to be decrypted

`dataElement` : The data element in *String* format that will be used to decrypt the data

Result

This UDF returns decrypted string format data.

Example

```
from pyspark.sql.types import *
spark.udf.registerJavaFunction("ptyStringDecScalaWrapper",
"com.protegrity.spark.wrapper.ptyStringDec", StringType())
spark.sql("select ptyStringDecScalaWrapper (column1, 'Data_Element') from
table1;").show(truncate = False)
```

3.10.35 ptyStringReEncScalaWrapper()

This UDF re-encrypts *binary* value provided as input.

ptyStringReEncScalaWrapper (Binary colName, String oldDataElement, String newDataElement)

Parameters

`colName` : The column that contains data in *Binary* format, which needs to be re-encrypted

`oldDataElement` : The data element name in *String* format that was used previously to encrypt the data

`newDataElement` : The data element name in *String* format to be used to re-encrypt the data

Result

This UDF returns re-encrypted binary format data.

Example

```
from pyspark.sql.types import *
spark.udf.registerJavaFunction("ptyStringReEncScalaWrapper",
"com.protegrity.spark.wrapper.ptyStringReEnc", BinaryType())
spark.sql("select ptyStringReEncScalaWrapper (column1, 'Old_Data_Element',
'New_Data_Element' ) from table1;").show(truncate = False)
```

Chapter 4

Database Protector

Protegrity Database Protector provides database security solutions for multiple databases that include Oracle, MS SQL, Teradata, Netezza, DB2/Open Systems, and Greenplum. It is tightly integrated into the target system, which makes data protection an integral part of the database.

The Database Protector contains user-defined functions (UDF), which perform the following:

- Fetches the policy related information from the shared memory
- Applies the access control settings that are derived on the basis of policy settings
- Encrypts or tokenizes the data based on the policy settings
- Generates audit logs that are sent to the PEP Server

Note:
To avoid any performance issues resulting due to casting of the data, a general best practice is to protect the data and present the decryption related API/UDFs/commands, as applicable, in the tables as views to authorized users only. This eliminates the unauthorized user's access to the decryption API/UDFs/commands by limiting the access to the protected data only.

The decryption process is limited to authorized users and thus, doesn't cause any performance impact as the API/UDFs/commands are executed restrictively.

Warning:
With database protectors, you cannot use different data elements for different rows in the same query because of the caching feature. The caching feature will cache the data element that you pass and it will use the same data element for protect/unprotect actions in the column.

4.1 DB2 Open Systems User-Defined Functions

This section provides a detailed list of the User Defined Functions or UDFs for general information, and protection and unprotection of different data types.

In DB2, before calling the UDF, DB2 uses the call-type parameter to pass the data types. For any UDF, the call type has the following values:

- -1 - indicates that this is the first call to the UDF. In a first call, all the SQL argument values are passed to the UDF.
- 0 - indicates that this is the normal call to the UDF. In a normal call, all the normal input argument values are passed to the UDF.
- 1 - indicates that this is the final call to the UDF. In a final call, no input parameters are passed to the UDF. The UDF can execute SQL statements when a value of 1 is passed. In a final call, the UDF should release the system resources, such as memory that is acquired during the first and the normal call.

> **Note:**
> You can pass only one data element between the first call and the final call in a UDF. This means that all the subsequent calls, after the first call, in the query will use the same data element that you pass in the first call.

4.1.1 General UDFs

This section includes list of general UDFs that can be used to retrieve the DB2 Protector version and the current user.

4.1.1.1 pty.whoami

This function returns the name of the user.

Signature

pty.whoami()

Parameters

None

Returns

The name of user logged on to the database as VARCHAR(256).

Example

```
SELECT pty.whoami() FROM SYSIBM.SYSDUMMY1;
```

4.1.1.2 pty.getversion

This function returns the version number of the DB2 protector.

Signature

pty.getversion()

Parameters

None

Returns

The version number of the DB2 protector as VARCHAR(256).

Example

```
SELECT pty.getversion() FROM SYSIBM.SYSDUMMY1;
```

4.1.1.3 pty.getcoreversion

This function returns the core version of the protector.

pty.getcoreversion()

Parameters

None

Returns

The core version of the DB2 protector as VARCHAR(256).

Example

```
SELECT pty.getcoreversion() FROM SYSIBM.SYSDUMMY1;
```

4.1.1.4 pty.getcurrentkeyid

This function returns the current active key identification number for a data element. It is typically used together with *getkeyid* to determine if some data is protected with the most recent key for a given data element.

Signature

pty.getcurrentkeyid(communicationid INTEGER, dataelement VARCHAR)

Parameters

Name	Type	Description
communicationid	*INTEGER*	Specifies the location where the UDF will find the policy. Must be the same as configured in the *pepserver.cfg* file.
dataelement	*VARCHAR(64)*	Specifies the name of the data element.

Returns

The current key ID as *INTEGER* data type.

Exception

If you configure a data element or a security operation in the policy and if the user does not have access to it, then the function will terminate with an appropriate error message.

Example

```
SELECT pty.getcurrentkeyid(0, 'AES128_IV_CRC_KID') FROM SYSIBM.SYSDUMMY1;
```

4.1.1.5 pty.getkeyid

This function returns the key ID that was used to protect a value of data. It is typically used together with *getcurrentkeyid* to determine if the data is protected with the most recent key for a given data element.

pty.getkeyid(communicationid INTEGER, dataelement VARCHAR, data VARCHAR)

Parameters

Name	Type	Description
communicationid	INTEGER	Specifies the location where the UDF will find the policy. Must be the same as configured in the *pepserver.cfg* file.
dataelement	VARCHAR(64)	Specifies the name of the data element.
data	VARCHAR(32672) FOR BIT DATA	Specifies the data that has been protected with encryption and is using the key ID.

Returns

The Key ID as *INTEGER* data type.

Exception

If set in policy and user does not have access, then the UDF terminates.

Example

```
SELECT pty.getkeyid(0, 'AES128', 'Protegrity', 0) FROM SYSIBM.SYSDUMMY1;
```

4.1.2 Access Check UDFs

These UDFs are used to check access permissions. The functions returns 1 if the user has access; otherwise it returns a 0 (zero).

4.1.2.1 pty.have_sel_perm

This function verifies whether the user has select(unprotect) access to a data element.

Signature

pty.have_sel_perm(communicationid INTEGER, dataelement VARCHAR)

Parameters

Name	Type	Description
communicationid	*INTEGER*	Specifies the location where the UDF will find the policy. Must be the same as configured in the *pepserver.cfg* file.
dataelement	*VARCHAR(64)*	Specifies the name of the data element.

Returns

This UDF returns either of the following values:

Value	Condition
1	If the user has **select(unprotect)** access
0	If the user does not have access

Exception

If you configure a data element or a security operation in the policy and if the user does not have access to it, then the function will terminate with an appropriate error message.

Example

```
SELECT pty.have_sel_perm(0, 'NoEncryption') FROM SYSIBM.SYSDUMMY1;
```

4.1.2.2 pty.have_upd_perm

This function verifies whether the user has update(reprotect) access to a data element.

Signature

pty.have_upd_perm(communicationid INTEGER, dataelement VARCHAR)

Parameters

Name	Type	Description
communicationid	*INTEGER*	Specifies the location where the UDF will find the policy. Must be the same as configured in the*pepserver.cfg* file.
dataelement	VARCHAR(64)	Specifies the name of the data element.

Returns

This UDF returns either of the following values:

Value	Condition
1	If the user has **update** access

Value	Condition
0	If the user does not have access

Exception

If you configure a data element or a security operation in the policy and if the user does not have access to it, then the function will terminate with an appropriate error message.

Example

```
SELECT pty.have_upd_perm(0, 'NoEncryption') FROM SYSIBM.SYSDUMMY1;
```

4.1.2.3 pty.have_ins_perm

This function verifies whether the user has insert(protect) access to a data element.

Signature

pty.have_ins_perm(communicationid INTEGER, dataelement VARCHAR)

Parameters

Name	Type	Description
communicationid	*INTEGER*	Specifies the location where the UDF will find the policy. Must be the same as configured in the *pepserver.cfg* file.
dataelement	*VARCHAR(64)*	Specifies the name of the data element.

Returns

This UDF returns either of the following values:

Value	Condition
1	If the user has **select(unprotected)** access
0	If the user does not have access

Exception

If you configure a data element or a security operation in the policy and if the user does not have access to it, then the function will terminate with an appropriate error message.

Example

```
SELECT pty.have_ins_perm(0, 'NoEncryption') FROM SYSIBM.SYSDUMMY1;
```

4.1.2.4 pty.have_del_perm

This function verifies whether the user has delete access to a data element.

Signature

pty.have_del_perm(communicationid INTEGER, dataelement VARCHAR, scid BINARY_INTEGER)

Parameters

Name	Type	Description
communicationid	*INTEGER*	Specifies the location where the UDF will find the policy. Must be the same as configured in the *pepserver.cfg* file.
dataelement	*VARCHAR(64)*	Specifies the name of the data element.

Name	Type	Description
scid	INTEGER	Specifies the Security Coordinate ID. It is not used and should be set to zero.

Returns

This UDF returns either of the following values:

Value	Condition
1	If the user has **delete** access
0	If the user does not have access

Exception

If you configure a data element or a security operation in the policy and if the user does not have access to it, then the function will terminate with an appropriate error message.

Example

```
SELECT pty.have_del_perm(0, 'NoEncryption', 10) FROM SYSIBM.SYSDUMMY1;
```

4.1.2.5 pty.del_check

This function verifies whether the user has delete access to a data element.

Signature

pty.have_del_check(communicationid INTEGER, dataelement VARCHAR, scid BINARY_INTEGER)

Parameters

Name	Type	Description
communicationid	*INTEGER*	Specifies the location where the UDF will find the policy. Must be the same as configured in the *pepserver.cfg* file.
dataelement	*VARCHAR(64)*	Specifies the name of the data element.
scid	INTEGER	Specifies the Security Coordinate ID. It is not used and should be set to zero.

Returns

This UDF returns either of the following values:

Value	Condition
1	If the user has **delete** access
0	If the user does not have access

Exception

If you configure a data element or a security operation in the policy and if the user does not have access to it, then the function will terminate with an appropriate error message.

Example

```
SELECT pty. del_check(0, 'NoEncryption', 10) FROM SYSIBM.SYSDUMMY1;
```

4.1.3 VARCHAR UDFs

These UDFs are used to protect and unprotect VARCHAR data.

4.1.3.1 pty.ins_enc_varchar

This function encrypts the VARCHAR data with a data element for encryption.

pty.ins_enc_varchar(communicationid INTEGER, dataelement VARCHAR, data VARCHAR, scid INTEGER)

Parameters

Name	Type	Description
communicationid	INTEGER	Specifies the location where the UDF will find the policy. Must be the same as configured in the *pepserver.cfg* file.
dataelement	VARCHAR(64)	Specifies the name of the data element.
data	VARCHAR(32672)	Specifies the input data for the UDF.
scid	INTEGER	Specifies the Security Coordinate ID. It is not used and should be set to zero.

Returns

Encrypted value as VARCHAR(32672).

NULL: When user has no access to database.

Exception

If set in policy and user does not have access, then the UDF terminates.

Example

```
SELECT pty.ins_enc_varchar(0, 'AES128', ‘Protegrity’, 0) FROM SYSIBM.SYSDUMMY1;
```

4.1.3.2 pty.upd_enc_varchar

This function re-encrypts the VARCHAR data that has been updated with a data element for encryption.

pty.upd_enc_varchar(communicationid INTEGER, dataelement VARCHAR, data VARCHAR, scid INTEGER)

Parameters

Name	Type	Description
communicationid	INTEGER	Specifies the location where the UDF will find the policy. Must be the same as configured in the *pepserver.cfg* file.
dataelement	VARCHAR(64)	Specifies the name of the data element.
data	VARCHAR(32672)	Specifies the input data for the UDF.
scid	INTEGER	Specifies the Security Coordinate ID. It is not used and is set to zero

Returns

Encrypted value as VARCHAR(32672).

NULL: When user has no access to database.

Exception

If set in policy and user does not have access, then the UDF terminates.

Example

```
SELECT pty.upd_enc_varchar(0, 'AES128', ‘Protegrity’, 0) FROM SYSIBM.SYSDUMMY1;
```

4.1.3.3 pty.sel_dec_varchar

This function decrypts the VARCHAR data with a data element.

pty.sel_dec_varchar(communicationid INTEGER, dataelement VARCHAR, data VARCHAR, scid INTEGER)

Parameters

Name	Type	Description
communicationid	INTEGER	Specifies the location where the UDF will find the policy. Must be the same as configured in the *pepserver.cfg* file.
dataelement	VARCHAR(64)	Specifies the name of the data element.
data	VARCHAR(32672)	Specifies the input data for the UDF.
scid	INTEGER	Specifies the Security Coordinate ID. It is not used and should be set to zero.

Returns

Encrypted value as VARCHAR(32672).

NULL: When user has no access to database.

Exception

If set in policy and user does not have access, then the UDF terminates.

Example

```
SELECT pty.sel_dec_varchar(0, 'AES128', pty.ins_enc_varchar(0, 'AES128', 'Protegrity',
0), 0) FROM SYSIBM.SYSDUMMY1;
```

4.1.3.4 pty.ins_varchar

This function protects the VARCHAR data with the following data elements:

- Tokenization
- Format Preserving Encryption(FPE) with the Plaintext Encoding type as ASCII
- No Encryption

Note: Masking is supported for FPE ASCII only. The *pty.ins_varchar* UDF supports FPE ASCII data elements only.

pty.ins_varchar(comm_id INTEGER, dataelement VARCHAR, input_data VARCHAR, scid INTEGER)

Parameters

Name	Type	Description
comm_id	INTEGER	Specifies the location where the UDF will find the policy. Must be the same as configured in the *pepserver.cfg* file.
dataelement	VARCHAR(64)	Specifies the name of the data element.
input_data	VARCHAR(32672)	Specifies the input data for the UDF.
scid	INTEGER	Specifies the Security Coordinate ID. It is not used and should be set to zero.

Returns

Data as VARCHAR(32672).

NULL: When user has no access to database.

Exception

If set in policy and user does not have access, then the UDF terminates.

Example for Tokenization

```
SELECT pty.ins_varchar(0, 'TE_A_S13_L1R2_Y', 'Protegrity', 0) FROM SYSIBM.SYSDUMMY1;
```

Example for FPE

```
SELECT
pty.ins_varchar('FPE_Alpha_Numeric_ASCII_Minlen2_ID_CC_L0R0_ASTNE','ProtegrityIndia',0)
FROM SYSIBM.SYSDUMMY1;
```

4.1.3.5 pty.upd_varchar

This function re-protects the VARCHAR data with the data elements, such as, Tokenization and No Encryption method.

pty.upd_varchar(comm_id INTEGER, dataelement VARCHAR, input_data VARCHAR, scid INTEGER)

Parameters

Name	Type	Description
comm_id	INTEGER	Specifies the location where the UDF will find the policy. Must be the same as configured in the *pepserver.cfg* file.
dataelement	VARCHAR(64)	Specifies the name of the data element.
input_data	VARCHAR(32672)	Specifies the input data for the UDF.
scid	INTEGER	Specifies the Security Coordinate ID. It is not used and should be set to zero.

Returns

Data as VARCHAR(32672).

NULL: When user has no access to database.

Exception

If set in policy and user does not have access, then the UDF terminates.

Example

```
SELECT pty.upd_varchar(0, 'TE_A_S13_L1R2_Y', 'Protegrity', 0) FROM SYSIBM.SYSDUMMY1;
```

4.1.3.6 pty.sel_varchar

This function unprotects the VARCHAR data with the following data elements:

- Tokenization
- Format Preserving Encryption(FPE) with the Plaintext Encoding type as ASCII
- No Encryption

pty.sel_varchar(comm_id INTEGER, dataelement VARCHAR, input_data VARCHAR, scid INTEGER)

Parameters

Name	Type	Description
comm_id	INTEGER	Specifies the location where the UDF will find the policy. Must be the same as configured in the *pepserver.cfg* file.
dataelement	VARCHAR(64)	Specifies the name of the data element.
input_data	VARCHAR(32672)	Specifies the input data for the UDF.
scid	INTEGER	Specifies the Security Coordinate ID. It is not used and should be set to zero.

Returns

Data as VARCHAR(32672).

NULL: When user has no access to database.

Exception

If set in policy and user does not have access, then the UDF terminates.

Example for Tokenization

```
SELECT pty.sel_varchar(0, 'TE_A_S13_L1R2_Y', pty.ins_varchar(0, 'TE_A_S13_L1R2_Y',
'Protegrity', 0), 0) FROM SYSIBM.SYSDUMMY1;
```

Example for FPE

```
SELECT pty.sel_varchar(0,
'FPE_FF1_AES256_ASCII_APIP_AN_L0R0_ASTNE_ML2',pty.ins_varchar(0,'FPE_FF1_AES256_ASCII_AP
IP_AN_L0R0_ASTNE_ML2','ProtegrityIndia',0),0) FROM SYSIBM.SYSDUMMY1
```

4.1.3.7 pty.ins_hash_varchar

This UDF uses the hash function to protect the VARCHAR data with a data element for hashing to return a protected value.

pty.ins_hash_varchar(comm_id INTEGER, dataelement VARCHAR, input_data VARCHAR, scid INTEGER)

Parameters

Name	Type	Description
comm_id	INTEGER	Specifies the location where the UDF will find the policy. Must be the same as configured in the *pepserver.cfg* file.
dataelement	VARCHAR(64)	Specifies the name of the data element.
input_data	VARCHAR(32672)	Specifies the input data for the UDF.
scid	INTEGER	Specifies the Security Coordinate ID. It is not used and should be set to zero.

Returns

Hash value as VARCHAR(32672).

NULL: When user has no access to database.

Exception

If set in policy and user does not have access, then the UDF terminates.

Example

```
SELECT pty.ins_hash_varchar( 0, 'HMAC_SHA1', 'Protegrity', 0) FROM SYSIBM.SYSDUMMY1;
```

4.1.3.8 pty.upd_hash_varchar

This UDF uses the hash function to protect the VARCHAR data with a data element for hashing to return a protected value.

pty.upd_hash_varchar(comm_id INTEGER, dataelement VARCHAR, input_data VARCHAR, scid INTEGER)

Parameters

Name	Type	Description
comm_id	INTEGER	Specifies the location where the UDF will find the policy. Must be the same as configured in the *pepserver.cfg* file.
dataelement	VARCHAR(64)	Specifies the name of the data element.
input_data	VARCHAR(32672)	Specifies the input data for the UDF.
scid	INTEGER	Specifies the Security Coordinate ID. It is not used and should be set to zero.

Returns

Hashed value as VARCHAR(32672) for bit data.

NULL: When user has no access to database.

Exception

If set in policy and user does not have access, then the UDF terminates.

Example

```
SELECT pty.upd_hash_varchar( 0, 'HMAC_SHA1', 'Protegrity', 0) FROM SYSIBM.SYSDUMMY1;
```

4.1.3.9 pty.ins_unicode_varchar

This UDF protects the Unicode VARCHAR data with the following data elements:

- Tokenization types, such as, Unicode, Unicode Base64, Unicode Gen2
- Format Preserving Encryption (FPE) with UTF8, UTF16LE, UTF16BE as the plaintext encoding
- No Encryption

pty.ins_unicode_varchar(comm_id INTEGER, dataelement VARCHAR, input_data VARCHAR, scid INTEGER)

Parameters

Name	Type	Description
comm_id	INTEGER	Specifies the location where the UDF will find the policy. Must be the same as configured in the *pepserver.cfg* file.
dataelement	VARCHAR(64)	Specifies the name of the data element.
input_data	VARCHAR(32672)	Specifies the input data for the UDF.
scid	INTEGER	Specifies the Security Coordinate ID. It is not used and should be set to zero.

Returns

Data as VARCHAR(32672).

NULL: When user has no access to database.

Exception

If set in policy and user does not have access, then the UDF terminates.

Example for FPE

```
SELECT
pty.ins_unicode_varchar('FPE_FF1_AES256_UTF8_APIP_AN_L0R0_ASTNE_ML2','ProtegrityIndia',0
) FROM SYSIBM.SYSDUMMY1;
```

Example for Base64

```
SELECT pty.ins_unicode_varchar(0, 'TE_UNICODE_BASE64_SLT13_ASTYES',
'Protegrityprotegrity', 0) FROM SYSIBM.SYSDUMMY1;
```

Example for Unicode Gen2

> **Note:** Unicode Gen2 data elements supports newly introduced SLT_X_1 tokenizer from 9.1 version onwards along with existing SLT_1_3 tokenizer. For more information, refer to section *3.4.15 Unicode Gen2* in the *Protection Methods Reference Guide 9.1.0.0*.

```
SELECT
pty.ins_unicode_varchar('TE_UG2_UTF16LE_LL1AN_SLT13_L2R0_ASTYES',N'xyzÀÁÂÃÄÅÆÇÈÉÊ',0)
FROM SYSIBM.SYSDUMMY1;
```

```
SELECT pty.ins_unicode_varchar('TE_UG2_SLTX1_L2R2_N_IPA_Greek_Coptic_UTF16LE',N'▒ 0)
FROM SYSIBM.SYSDUMMY1;
```

4.1.3.10 pty.upd_unicode_varchar

This UDF re-protects the Unicode VARCHAR data with the following data elements:

- Tokenization types, such as, Unicode, Unicode Base64, Unicode Gen2
- Format Preserving Encryption (FPE) with the Plaintext Encoding type as UTF8, UTF16LE, UTF16BE
- No Encryption

pty.upd_unicode_varchar(comm_id INTEGER, dataelement VARCHAR, input_data VARCHAR, scid INTEGER)

Parameters

Name	Type	Description
comm_id	INTEGER	Specifies the location where the UDF will find the policy. Must be the same as configured in the *pepserver.cfg* file.
dataelement	VARCHAR(64)	Specifies the name of the data element.
input_data	VARCHAR(32672)	Specifies the input data for the UDF.
scid	INTEGER	Specifies the Security Coordinate ID. It is not used and should be set to zero.

Returns

Data as VARCHAR(32672).

NULL: When user has no access to database.

Exception

If set in policy and user does not have access, then the UDF terminates.

Example for Unicode

```
SELECT  pty.upd_unicode_varchar(0, 'TE_Unicode_S13', 'Protegrity', 0) FROM
SYSIBM.SYSDUMMY1;
```

Example for Base64

```
SELECT pty.upd_unicode_varchar(0, 'TE_UNICODE_BASE64_SLT13_ASTYES',
'Protegrityprotegrity', 0) FROM SYSIBM.SYSDUMMY1;
```

Example for Unicode Gen2

> **Note:** Unicode Gen2 data elements supports newly introduced SLT_X_1 tokenizer from 9.1 version onwards along with existing SLT_1_3 tokenizer. For more information, refer to section *3.4.15 Unicode Gen2* in the *Protection Methods Reference Guide 9.1.0.0*.

```
SELECT
pty.upd_unicode_varchar('TE_UG2_UTF16LE_LL1AN_SLT13_L2R0_ASTYES',N'xyzÀÁÂÃÄÅÆÇÈÉÊ',0)
FROM SYSIBM.SYSDUMMY1;
```

```
SELECT pty.upd_unicode_varchar('TE_UG2_SLTX1_L2R2_N_IPA_Greek_Coptic_UTF16LE',N'▒,0)
FROM SYSIBM.SYSDUMMY1;
```

4.1.3.11 pty.sel_unicode_varchar

This UDF unprotects the Unicode VARCHAR data with the following data elements:

- Tokenization types, such as, Unicode, Unicode Base64, Unicode Gen2
- Format Preserving Encryption (FPE) with UTF8, UTF16LE, UTF16BE as the plaintext encoding
- No Encryption

pty.sel_unicode_varchar(comm_id INTEGER, dataelement VARCHAR, input_data VARCHAR, scid INTEGER)

Parameters

Name	Type	Description
comm_id	INTEGER	Specifies the location where the UDF will find the policy. Must be the same as configured in the *pepserver.cfg* file.
dataelement	VARCHAR(64)	Specifies the name of the data element.
input_data	VARCHAR(32672)	Specifies the input data for the UDF.
scid	INTEGER	Specifies the Security Coordinate ID. It is not used and should be set to zero.

Returns

Data as VARCHAR(32672).

NULL: When user has no access to database.

Exception

If set in policy and user does not have access, then the UDF terminates.

Example for FPE

```
SELECT pty.sel_unicode_varchar(0, 'TE_Unicode_S13', pty.ins_varchar(0,
'TE_Unicode_S13', 'Protegrity', 0), 0) FROM SYSIBM.SYSDUMMY1;
```

Example for Base64

```
SELECT pty.sel_unicode_varchar(0, 'TE_UNICODE_BASE64_SLT13_ASTYES',
pty.ins_unicode_varchar(0, 'TE_UNICODE_BASE64_SLT13_ASTYES', 'Protegrityprotegrity',
0), 0) FROM SYSIBM.SYSDUMMY1;
```

Example for Unicode Gen2

> **Note:** Unicode Gen2 data elements supports newly introduced SLT_X_1 tokenizer from 9.1 version onwards along with existing SLT_1_3 tokenizer. For more information, refer to section *3.4.15 Unicode Gen2* in the *Protection Methods Reference Guide 9.1.0.0*.

```
SELECT pty.sel_unicode_varchar(0, 'TE_UG2_UTF16LE_LL1AN_SLT13_L2R0_ASTYES',
pty.ins_unicode_varchar('TE_UG2_UTF16LE_LL1AN_SLT13_L2R0_ASTYES',N'xyzÀÁÂÃÄÅÆÇÈÉÊ',0),
0) FROM SYSIBM.SYSDUMMY1;
```

```
SELECT pty.sel_unicode_varchar(0, 'TE_UG2_SLTX1_L2R2_N_IPA_Greek_Coptic_UTF16LE',
pty.ins_unicode_varchar('TE_UG2_SLTX1_L2R2_N_IPA_Greek_Coptic_UTF16LE',N'▒,0), 0) FROM
SYSIBM.SYSDUMMY1;
```

4.1.4 VARCHAR FOR BIT DATA UDFs

These UDFs are used to encrypt or decrypt VARCHAR FOR BIT DATA.

4.1.4.1 pty.ins_enc_varcharfbd

This function encrypts the VARCHARFBD value with a data element for encryption.

pty.ins_enc_varcharfbd(comm_id INTEGER, dataelement VARCHAR, input_data VARCHAR, scid INTEGER)

Parameters

Name	Type	Description
comm_id	INTEGER	Specifies the location where the UDF will find the policy. Must be the same as configured in the *pepserver.cfg* file.
dataelement	VARCHAR(64)	Specifies the name of the data element.
input_data	VARCHAR(32672) FOR BIT DATA	Specifies the input data for the UDF.
scid	INTEGER	Specifies the Security Coordinate ID. It is not used and should be set to zero.

Returns

Encrypted value as VARCHAR(32672) FOR BIT DATA.

NULL: When user has no access to database.

Exception

If set in policy and user does not have access, then the UDF terminates.

Example

```
SELECT pty.ins_enc_varcharfbd( 0, 'AES128', CAST( 'Protegrity' AS VARCHAR(10) FOR BIT
DATA ), 0) FROM SYSIBM.SYSDUMMY1;
```

4.1.4.2 pty.upd_enc_varcharfbd

This function re-encrypts the VARCHARFBD value that has been updated with a data element for encryption.

pty.upd_enc_varcharfbd(comm_id INTEGER, dataelement VARCHAR, input_data VARCHAR, scid INTEGER)

Parameters

Name	Type	Description
comm_id	INTEGER	Specifies the location where the UDF will find the policy. Must be the same as configured in the *pepserver.cfg* file.
dataelement	VARCHAR(64)	Specifies the name of the data element.
input_data	VARCHAR(32672) FOR BIT DATA	Specifies the input data for the UDF.
scid	INTEGER	Specifies the Security Coordinate ID. It is not used and should be set to zero.

Returns

Encrypted value as VARCHAR(32672) FOR BIT DATA.

NULL: When user has no access to database.

Exception

If set in policy and user does not have access, then the UDF terminates.

Example

```
SELECT pty.upd_enc_varcharfbd( 0, 'AES128', CAST( 'Protegrity' AS VARCHAR(10) FOR BIT
DATA ), 0) FROM SYSIBM.SYSDUMMY1;
```

4.1.4.3 pty.sel_dec_varcharfbd

This function decrypts the VARCHARFBD value with a data element.

pty.sel_dec_varcharfbd(comm_id INTEGER, dataelement VARCHAR, input_data VARCHAR, scid INTEGER)

Parameters

Name	Type	Description
comm_id	INTEGER	Specifies the location where the UDF will find the policy. Must be the same as configured in the *pepserver.cfg* file.
dataelement	VARCHAR(64)	Specifies the name of the data element.
input_data	VARCHAR(32672) FOR BIT DATA	Specifies the input data for the UDF.
scid	INTEGER	Specifies the Security Coordinate ID. It is not used and should be set to zero.

Returns

Decrypted value as VARCHAR(32672) FOR BIT DATA.

NULL: When user has no access to database.

Exception

If set in policy and user does not have access, then the UDF terminates.

Example

```
SELECT pty.sel_dec_varcharfbd
(0,'AES128',pty.ins_enc_varcharfbd(0,'AES128',CAST('Protegrity' AS VARCHAR(10) FOR BIT
DATA),0),0) FROM SYSIBM.SYSDUMMY1;
```

4.1.4.4 pty.ins_varcharfbd

This function protects the VARCHARFBD value with the following data elements:

- Tokenization
- Format Preserving Encryption(FPE) with the Plaintext Encoding type as ASCII
- No Encryption

pty.ins_varcharfbd(comm_id INTEGER, dataelement VARCHAR, input_data VARCHAR, scid INTEGER)

Parameters

Name	Type	Description
comm_id	INTEGER	Specifies the location where the UDF will find the policy. Must be the same as configured in the *pepserver.cfg* file.
dataelement	VARCHAR(64)	Specifies the name of the data element.
input_data	VARCHAR(32672) FOR BIT DATA	Specifies the input data for the UDF.
scid	INTEGER	Specifies the Security Coordinate ID. It is not used and should be set to zero.

Returns

Data value as VARCHAR(32672) FOR BIT DATA.

NULL: When user has no access to database.

Exception

If set in policy and user does not have access, then the UDF terminates.

Example

```
SELECT pty.ins_varcharfbd(0,'TE_A_S13_L1R2_Y',CAST('Protegrity' AS VARCHAR(10) FOR BIT
DATA),0) FROM SYSIBM.SYSDUMMY1;
```

4.1.4.5 pty.upd_varcharfbd

This function re-protects the VARCHARFBD value with the data elements, such as, Tokenization and No Encryption method.

pty.upd_varcharfbd(comm_id INTEGER, dataelement VARCHAR, input_data VARCHAR, scid INTEGER)

Parameters

Name	Type	Description
comm_id	INTEGER	Specifies the location where the UDF will find the policy. Must be the same as configured in the *pepserver.cfg* file.

Name	Type	Description
dataelement	VARCHAR(64)	Specifies the name of the data element.
input_data	VARCHAR(32672) FOR BIT DATA	Specifies the input data for the UDF.
scid	INTEGER	Specifies the Security Coordinate ID. It is not used and should be set to zero.

Returns

Data value as VARCHAR(32672) FOR BIT DATA.

NULL: When user has no access to database.

Exception

If set in policy and user does not have access, then the UDF terminates.

Example

```
SELECT pty.upd_varcharfbd(0,'TE_A_S13_L1R2_Y',CAST('Protegrity' AS VARCHAR(10) FOR BIT
DATA),0) FROM SYSIBM.SYSDUMMY1;
```

4.1.4.6 pty.sel_varcharfbd

This function unprotects the VARCHARFBD value with the following data elements:

- Tokenization
- Format Preserving Encryption(FPE) with the Plaintext Encoding type as ASCII
- No Encryption

pty.sel_varcharfbd(comm_id INTEGER, dataelement VARCHAR, input_data VARCHAR, scid INTEGER)

Parameters

Name	Type	Description
comm_id	INTEGER	Specifies the location where the UDF will find the policy. Must be the same as configured in the *pepserver.cfg* file.
dataelement	VARCHAR(64)	Specifies the name of the data element.
input_data	VARCHAR(32672) FOR BIT DATA	Specifies the input data for the UDF.
scid	INTEGER	Specifies the Security Coordinate ID. It is not used and should be set to zero.

Returns

Data value as VARCHAR(32672) FOR BIT DATA.

NULL: When user has no access to database.

Exception

If set in policy and user does not have access, then the UDF terminates.

Example

```
SELECT
pty.sel_varcharfbd(0,'TE_A_S13_L1R2_Y',pty.ins_varcharfbd(0,'TE_A_S13_L1R2_Y',CAST('Prot
egrity' AS VARCHAR(10) FOR BIT DATA),0),0) FROM SYSIBM.SYSDUMMY1;
```

4.1.5 CHAR UDFs

These UDFs are used to encrypt or decrypt CHAR data.

4.1.5.1 pty.ins_enc_char

This function encrypts the CHAR data with a data element for encryption.

pty.ins_enc_char(comm_id INTEGER, dataelement VARCHAR, input_data CHAR, scid INTEGER)

Parameters

Name	Type	Description
comm_id	INTEGER	Specifies the location where the UDF will find the policy. Must be the same as configured in the *pepserver.cfg* file.
dataelement	VARCHAR(64)	Specifies the name of the data element.
input_data	VARCHAR(254)	Specifies the input data for the UDF.
scid	INTEGER	Specifies the Security Coordinate ID. It is not used and should be set to zero.

Returns

Encrypted value as VARCHAR(290) FOR BIT DATA.

NULL: When user has no access to database.

Exception

If set in policy and user does not have access, then the UDF terminates.

Example

```
SELECT pty.ins_enc_char(0,'AES128','Protegrity',0) FROM SYSIBM.SYSDUMMY1;
```

4.1.5.2 pty.upd_enc_char

This function re-encrypts the CHAR data that has been updated with a data element for encryption.

pty.upd_enc_char(comm_id INTEGER, dataelement VARCHAR, input_data VARCHAR, scid INTEGER)

Parameters

Name	Type	Description
comm_id	INTEGER	Specifies the location where the UDF will find the policy. Must be the same as configured in the *pepserver.cfg* file.
dataelement	VARCHAR(64)	Specifies the name of the data element.

Name	Type	Description
input_data	VARCHAR(254)	Specifies the input data for the UDF.
scid	INTEGER	Specifies the Security Coordinate ID. It is not used and should be set to zero.

Returns

Encrypted value as VARCHAR(290) FOR BIT DATA.

NULL: When user has no access to database.

Exception

If set in policy and user does not have access, then the UDF terminates.

Example

```
SELECT pty.upd_enc_char(0,'AES128','Protegrity',0) FROM SYSIBM.SYSDUMMY1;
```

4.1.5.3 pty.sel_dec_char

This function decrypts the CHAR data with a data element.

pty.sel_dec_char(comm_id INTEGER, dataelement VARCHAR, input_data VARCHAR, scid INTEGER)

Parameters

Name	Type	Description
comm_id	INTEGER	Specifies the location where the UDF will find the policy. Must be the same as configured in the *pepserver.cfg* file.
dataelement	VARCHAR(64)	Specifies the name of the data element.
input_data	VARCHAR(290) FOR BIT DATA	Specifies the input data for the UDF.
scid	INTEGER	Specifies the Security Coordinate ID. It is not used and should be set to zero.

Returns

Decrypted value as VARCHAR(254).

NULL: When user has no access to database.

Exception

If set in policy and user does not have access, then the UDF terminates.

Example

```
SELECT PTY.sel_dec_char(0,'AES128',pty.ins_enc_char(0,'AES128','Protegrity',0),0) FROM
SYSIBM.SYSDUMMY1;
```

4.1.5.4 pty.ins_char

This function protects the CHAR data with the data elements, such as, Tokenization and No Encryption method.

> **Caution:** The `pty.ins_char` UDF cannot be used for FPE data element.

pty.ins_char(comm_id INTEGER, dataelement VARCHAR, input_data VARCHAR, scid INTEGER)

Parameters

Name	Type	Description
comm_id	INTEGER	Specifies the location where the UDF will find the policy. Must be the same as configured in the *pepserver.cfg* file.
dataelement	VARCHAR(64)	Specifies the name of the data element.
input_data	VARCHAR(254)	Specifies the input data for the UDF.
scid	INTEGER	Specifies the Security Coordinate ID. It is not used and should be set to zero.

Returns

Data as VARCHAR(305).

NULL: When user has no access to database.

Exception

If set in policy and user does not have access, then the UDF terminates.

Example

```
SELECT pty.ins_char(0,'TE_A_S13_L1R2_Y','Protegrity',0) FROM SYSIBM.SYSDUMMY1;
```

4.1.5.5 pty.upd_char

This function re-protects the CHAR data with the data elements, such as, Tokenization and No Encryption method.

pty.upd_char(comm_id INTEGER, dataelement VARCHAR, input_data VARCHAR, scid INTEGER)

Parameters

Name	Type	Description
comm_id	INTEGER	Specifies the location where the UDF will find the policy. Must be the same as configured in the *pepserver.cfg* file.
dataelement	VARCHAR(64)	Specifies the name of the data element.
input_data	VARCHAR(254)	Specifies the input data for the UDF.
scid	INTEGER	Specifies the Security Coordinate ID. It is not used and should be set to zero.

Returns

Encrypted value as VARCHAR(305).

NULL: When user has no access to database.

Exception

If set in policy and user does not have access, then the UDF terminates.

Example

```
SELECT pty.upd_char(0,'TE_A_S13_L1R2_Y','Protegrity',0) FROM SYSIBM.SYSDUMMY1;
```

4.1.5.6 pty.sel_char

This function unprotects the CHAR data with the data elements, such as, Tokenization and No Encryption method.

pty.sel_char(comm_id INTEGER, dataelement VARCHAR, input_data VARCHAR, scid INTEGER)

Parameters

Name	Type	Description
comm_id	INTEGER	Specifies the location where the UDF will find the policy. Must be the same as configured in the *pepserver.cfg* file.
dataelement	VARCHAR(64)	Specifies the name of the data element.
input_data	VARCHAR(305)	Specifies the input data for the UDF.
scid	INTEGER	Specifies the Security Coordinate ID. It is not used and should be set to zero.

Returns

Data as VARCHAR(254).

NULL: When user has no access to database.

Exception

If set in policy and user does not have access, then the UDF terminates.

Example

```
SELECT
pty.sel_char(0,'TE_A_S13_L1R2_Y',pty.ins_char(0,'TE_A_S13_L1R2_Y','Protegrity',0),0)
FROM SYSIBM.SYSDUMMY1;
```

4.1.6 CHAR FOR BIT DATA UDFs

These UDFs are used to encrypt or decrypt CHAR FOR BIT DATA.

4.1.6.1 pty.ins_enc_charfbd

This function encrypts the CHARFBD data with a data element for encryption.

pty.ins_enc_charfbd(comm_id INTEGER, dataelement VARCHAR, input_data VARCHAR, scid INTEGER)

Parameters

Name	Type	Description
comm_id	INTEGER	Specifies the location where the UDF will find the policy. Must be the same as configured in the *pepserver.cfg* file.
dataelement	VARCHAR(64)	Specifies the name of the data element.
input_data	VARCHAR(254) FOR BIT DATA	Specifies the input data for the UDF.
scid	INTEGER	Specifies the Security Coordinate ID. It is not used and should be set to zero.

Returns

Encrypted data as VARCHAR(290) FOR BIT DATA.

NULL: When user has no access to database.

Exception

If set in policy and user does not have access, then the UDF terminates.

Example

```
SELECT pty.ins_enc_charfbd(0,'AES128',CAST('Protegrity' AS CHAR(10) FOR BIT DATA),0)
FROM SYSIBM.SYSDUMMY1;
```

4.1.6.2 pty.upd_enc_charfbd

This function re-encrypts the CHARFBD data that has been updated with a data element for encryption.

pty.upd_enc_charfbd(comm_id INTEGER, dataelement VARCHAR, input_data VARCHAR, scid INTEGER)

Parameters

Name	Type	Description
comm_id	INTEGER	Specifies the location where the UDF will find the policy. Must be the same as configured in the *pepserver.cfg* file.
dataelement	VARCHAR(64)	Specifies the name of the data element.
input_data	VARCHAR(254) FOR BIT DATA	Specifies the input data for the UDF.
scid	INTEGER	Specifies the Security Coordinate ID. It is not used and should be set to zero.

Returns

Encrypted data as VARCHAR(290) FOR BIT DATA.

NULL: When user has no access to database.

Exception

If set in policy and user does not have access, then the UDF terminates.

Example

```
SELECT pty.upd_enc_charfbd(0,'AES128',CAST('Protegrity' AS CHAR(10) FOR BIT DATA),0)
FROM SYSIBM.SYSDUMMY1;
```

4.1.6.3 pty.sel_dec_charfbd

This function decrypts the CHARFBD data with a data element.

pty.upd_enc_charfbd(comm_id INTEGER, dataelement VARCHAR, input_data VARCHAR, scid INTEGER)

Parameters

Name	Type	Description
comm_id	INTEGER	Specifies the location where the UDF will find the policy. Must be the same as configured in the *pepserver.cfg* file.
dataelement	VARCHAR(64)	Specifies the name of the data element.
input_data	VARCHAR(290) FOR BIT DATA	Specifies the input data for the UDF.
scid	INTEGER	Specifies the Security Coordinate ID. It is not used and should be set to zero.

Returns

Decrypted data as VARCHAR(254) FOR BIT DATA.

NULL: When user has no access to database.

Exception

If set in policy and user does not have access, then the UDF terminates.

Example

```
SELECT pty.sel_dec_charfbd (0, 'AES128', pty.ins_enc_charfbd (0, 'AES128',
pty.ins_enc_charfbd (0, 'AES128', CAST ('Protegrity' AS CHAR(10) FOR BIT DATA), 0), 0),
0) FROM SYSIBM.SYSDUMMY1;
```

4.1.6.4 pty.ins_charfbd

This function protects the CHARFBD data with the data elements, such as, Tokenization and No Encryption method..

pty.ins_charfbd(comm_id INTEGER, dataelement VARCHAR, input_data VARCHAR, scid INTEGER)

Parameters

Name	Type	Description
comm_id	INTEGER	Specifies the location where the UDF will find the policy. Must be the same as configured in the *pepserver.cfg* file.
dataelement	VARCHAR(64)	Specifies the name of the data element.
input_data	VARCHAR(254) FOR BIT DATA	Specifies the input data for the UDF.

Name	Type	Description
scid	INTEGER	Specifies the Security Coordinate ID. It is not used and should be set to zero.

Returns

Data as VARCHAR(254) FOR BIT DATA.

NULL: When user has no access to database.

Exception

If set in policy and user does not have access, then the UDF terminates.

Example

```
SELECT pty.ins_charfbd(0,'TE_A_S13_L1R2_Y',CAST('Protegrity' AS CHAR(10) FOR BIT
DATA),0) FROM SYSIBM.SYSDUMMY1;
```

4.1.6.5 pty.upd_charfbd

This function re-protects the CHARFBD data with the data elements, such as, Tokenization and No Encryption method.

pty.upd_charfbd(comm_id INTEGER, dataelement VARCHAR, input_data VARCHAR, scid INTEGER)

Parameters

Name	Type	Description
comm_id	INTEGER	Specifies the location where the UDF will find the policy. Must be the same as configured in the *pepserver.cfg* file.
dataelement	VARCHAR(64)	Specifies the name of the data element.
input_data	VARCHAR(254) FOR BIT DATA	Specifies the input data for the UDF.
scid	INTEGER	Specifies the Security Coordinate ID. It is not used and should be set to zero.

Returns

Data as VARCHAR(254) FOR BIT DATA.

NULL: When user has no access to database.

Exception

If set in policy and user does not have access, then the UDF terminates.

Example

```
SELECT pty.upd_charfbd(0,'TE_A_S13_L1R2_Y',CAST('Protegrity' AS CHAR(10) FOR BIT
DATA),0) FROM SYSIBM.SYSDUMMY1;
```

4.1.6.6 pty.sel_charfbd

This function unprotects the CHARFBD data with the data elements, such as, Tokenization and No Encryption method.

pty.sel_charfbd(comm_id INTEGER, dataelement VARCHAR, input_data VARCHAR, scid INTEGER)

Parameters

Name	Type	Description
comm_id	INTEGER	Specifies the location where the UDF will find the policy. Must be the same as configured in the *pepserver.cfg* file.
dataelement	VARCHAR(64)	Specifies the name of the data element.
input_data	VARCHAR(254) FOR BIT DATA	Specifies the input data for the UDF.
scid	INTEGER	Specifies the Security Coordinate ID. It is not used and should be set to zero.

Returns

Data as VARCHAR(254) FOR BIT DATA.

NULL: When user has no access to database.

Exception

If set in policy and user does not have access, then the UDF terminates.

Example

```
SELECT
pty.sel_charfbd(0,'TE_A_S13_L1R2_Y',pty.ins_charfbd(0,'TE_A_S13_L1R2_Y',pty.ins_charfbd(
0, 'TE_A_S13_L1R2_Y',CAST('Protegrity' AS CHAR(10) FOR BIT DATA),0),0),0) FROM
SYSIBM.SYSDUMMY1;
```

4.1.7 LONG VARCHAR UDFs

These UDFs are used to encrypt and decrypt LONG VARCHAR data.

4.1.7.1 pty.ins_enc_lvarchar

This function encrypts LONG VARCHAR data with a data element for encryption.

pty.ins_enc_lvarchar(comm_id INTEGER, dataelement VARCHAR, input_data LONG VARCHAR, scid INTEGER)

Parameters

Name	Type	Description
comm_id	INTEGER	Specifies the location where the UDF will find the policy. Must be the same as configured in the *pepserver.cfg* file.
dataelement	VARCHAR(64)	Specifies the name of the data element.
input_data	LONG VARCHAR	Specifies the input data for the UDF.
scid	INTEGER	Specifies the Security Coordinate ID. It is not used and should be set to zero.

Returns

Encrypted data as LONG VARCHAR FOR BIT DATA.

NULL: When user has no access to database.

Exception

If set in policy and user does not have access, then the UDF terminates.

Example

```
SELECT pty.ins_enc_lvarchar(0,'AES128', CAST('Protegrity' AS LONG VARCHAR),0) FROM
SYSIBM.SYSDUMMY1;
```

4.1.7.2 pty.upd_enc_lvarchar

This re-encrypts the LONG VARCHAR data that has been updated with a data element for encryption.

pty.upd_enc_lvarchar(comm_id INTEGER, dataelement VARCHAR, input_data LONG VARCHAR, scid INTEGER)

Parameters

Name	Type	Description
comm_id	INTEGER	Specifies the location where the UDF will find the policy. Must be the same as configured in the *pepserver.cfg* file.
dataelement	VARCHAR(64)	Specifies the name of the data element.
input_data	LONG VARCHAR	Specifies the input data for the UDF.
scid	INTEGER	Specifies the Security Coordinate ID. It is not used and should be set to zero.

Returns

Encrypted data as LONG VARCHAR FOR BIT DATA.

NULL: When user has no access to database.

Exception

If set in policy and user does not have access, then the UDF terminates.

Example

```
SELECT pty.upd_enc_lvarchar(0,'AES128', CAST('Protegrity' AS LONG VARCHAR),0) FROM
SYSIBM.SYSDUMMY1;
```

4.1.7.3 pty.sel_dec_lvarchar

This function decrypts the LONG VARCHAR data with a data element.

pty.upd_enc_lvarchar(comm_id INTEGER, dataelement VARCHAR, input_data LONG VARCHAR, scid INTEGER)

Parameters

Name	Type	Description
comm_id	INTEGER	Specifies the location where the UDF will find the policy. Must be the same as configured in the *pepserver.cfg* file.
dataelement	VARCHAR(64)	Specifies the name of the data element.

Name	Type	Description
input_data	LONG VARCHAR FOR BIT DATA	Specifies the input data for the UDF.
scid	INTEGER	Specifies the Security Coordinate ID. It is not used and should be set to zero.

Returns

Encrypted data as LONG VARCHAR.

NULL: When user has no access to database.

Exception

If set in policy and user does not have access, then the UDF terminates.

Example

```
SELECT PTY.sel_dec_lvarchar(0,'AES128',pty.ins_enc_lvarchar(0,'AES128',
CAST('Protegrity' AS LONG VARCHAR),0),0) FROM SYSIBM.SYSDUMMY1;
```

4.1.7.4 pty.ins_lvarchar

This function protects the LONG VARCHAR data with the data elements, such as, Tokenization and No Encryption method.

pty.ins_lvarchar(comm_id INTEGER, dataelement VARCHAR, input_data LONG VARCHAR, scid INTEGER)

Parameters

Name	Type	Description
comm_id	INTEGER	Specifies the location where the UDF will find the policy. Must be the same as configured in the *pepserver.cfg* file.
dataelement	VARCHAR(64)	Specifies the name of the data element.
input_data	LONG VARCHAR	Specifies the input data for the UDF.
scid	INTEGER	Specifies the Security Coordinate ID. It is not used and should be set to zero.

Returns

Encrypted data as LONG VARCHAR.

NULL: When user has no access to database.

Exception

If set in policy and user does not have access, then the UDF terminates.

Example

```
SELECT pty.ins_lvarchar(0,'TE_A_S13_L1R2_Y', CAST('Protegrity' AS LONG VARCHAR),0) FROM
SYSIBM.SYSDUMMY1;
```

4.1.7.5 pty.upd_lvarchar

This function re-protects the LONG VARCHAR data with the data elements, such as, Tokenization and No Encryption method.

pty.upd_lvarchar(comm_id INTEGER, dataelement VARCHAR, input_data LONG VARCHAR, scid INTEGER)

Parameters

Name	Type	Description
comm_id	INTEGER	Specifies the location where the UDF will find the policy. Must be the same as configured in the *pepserver.cfg* file.
dataelement	VARCHAR(64)	Specifies the name of the data element.
input_data	LONG VARCHAR	Specifies the input data for the UDF.
scid	INTEGER	Specifies the Security Coordinate ID. It is not used and should be set to zero.

Returns

Encrypted data as LONG VARCHAR.

NULL: When user has no access to database.

Exception

If set in policy and user does not have access, then the UDF terminates.

Example

```
SELECT pty.upd_lvarchar(0,'TE_A_S13_L1R2_Y', CAST('Protegrity' AS LONG VARCHAR),0) FROM
SYSIBM.SYSDUMMY1;
```

4.1.7.6 pty.sel_lvarchar

This function unprotects the LONG VARCHAR data with the data elements, such as, Tokenization and No Encryption method.

pty.sel_lvarchar(comm_id INTEGER, dataelement VARCHAR, input_data LONG VARCHAR, scid INTEGER)

Parameters

Name	Type	Description
comm_id	INTEGER	Specifies the location where the UDF will find the policy. Must be the same as configured in the *pepserver.cfg* file.
dataelement	VARCHAR(64)	Specifies the name of the data element.
input_data	LONG VARCHAR	Specifies the input data for the UDF.
scid	INTEGER	Specifies the Security Coordinate ID. It is not used and should be set to zero.

Returns

Data as LONG VARCHAR.

NULL: When user has no access to database.

Exception

If set in policy and user does not have access, then the UDF terminates.

Example

```
SELECT pty.sel_lvarchar(0,'TE_A_S13_L1R2_Y',pty.ins_lvarchar(0,'TE_A_S13_L1R2_Y',
CAST('Protegrity' AS LONG VARCHAR),0),0) FROM SYSIBM.SYSDUMMY1;
```

4.1.8 LONG VARCHAR FOR BIT DATA UDFs

These UDFs are used to encrypt and decrypt VARCHAR FOR BIT DATA.

4.1.8.1 pty.ins_enc_lvcfbd

This function encrypts LONG VARCHARFBD data with a data element for encryption.

pty.ins_enc_lvcfbd(comm_id INTEGER, dataelement VARCHAR, input_data LONG VARCHAR, scid INTEGER)

Parameters

Name	Type	Description
comm_id	INTEGER	Specifies the location where the UDF will find the policy. Must be the same as configured in the *pepserver.cfg* file.
dataelement	VARCHAR(64)	Specifies the name of the data element.
input_data	LONG VARCHAR FOR BIT DATA	Specifies the input data for the UDF.
scid	INTEGER	Specifies the Security Coordinate ID. It is not used and should be set to zero.

Returns

Encrypted data as LONG VARCHAR FOR BIT DATA.

NULL: When user has no access to database.

Exception

If set in policy and user does not have access, then the UDF terminates.

Example

```
SELECT pty.ins_enc_lvcfbd(0,'AES128',CAST('Protegrity' AS LONG VARCHAR FOR BIT DATA),0)
FROM SYSIBM.SYSDUMMY1;
```

4.1.8.2 pty.upd_enc_lvcfbd

This re-encrypts the LONG VARCHARFBD data that has been updated with a data element for encryption.

pty.upd_enc_lvcfbd(comm_id INTEGER, dataelement VARCHAR, input_data LONG VARCHAR, scid INTEGER)

Parameters

Name	Type	Description
comm_id	INTEGER	Specifies the location where the UDF will find the policy. Must be the same as configured in the *pepserver.cfg* file.
dataelement	VARCHAR(64)	Specifies the name of the data element.

Name	Type	Description
input_data	LONG VARCHAR FOR BIT DATA	Specifies the input data for the UDF.
scid	INTEGER	Specifies the Security Coordinate ID. It is not used and should be set to zero.

Returns

Encrypted data as LONG VARCHAR FOR BIT DATA.

NULL: When user has no access to database.

Exception

If set in policy and user does not have access, then the UDF terminates.

Example

```
SELECT pty.upd_enc_lvcfbd(0,'AES128','Protegrity',0) FROM SYSIBM.SYSDUMMY1;
```

4.1.8.3 pty.sel_dec_lvcfbd

This function decrypts the LONG VARCHARFBD data with a data element.

pty.sel_dec_lvcfbd(comm_id INTEGER, dataelement VARCHAR, input_data LONG VARCHAR, scid INTEGER)

Parameters

Name	Type	Description
comm_id	INTEGER	Specifies the location where the UDF will find the policy. Must be the same as configured in the *pepserver.cfg* file.
dataelement	VARCHAR(64)	Specifies the name of the data element.
input_data	LONG VARCHAR FOR BIT DATA	Specifies the input data for the UDF.
scid	INTEGER	Specifies the Security Coordinate ID. It is not used and should be set to zero.

Returns

Decrypted data as LONG VARCHAR FOR BIT DATA.

NULL: When user has no access to database.

Exception

If set in policy and user does not have access, then the UDF terminates.

Example

```
SELECT pty.sel_dec_lvcfbd(0,'AES128', pty.ins_enc_lvcfbd(0,'AES128',CAST('Protegrity'
AS LONG VARCHAR FOR BIT DATA),0),0) FROM SYSIBM.SYSDUMMY1;
```

4.1.8.4 pty.ins_lvcfbd

This function protects the LONG VARCHARFBD data with the data elements, such as, Tokenization and No Encryption method.

pty.ins_lvcfbd(comm_id INTEGER, dataelement VARCHAR, input_data LONG VARCHAR, scid INTEGER)

Parameters

Name	Type	Description
comm_id	INTEGER	Specifies the location where the UDF will find the policy. Must be the same as configured in the *pepserver.cfg* file.
dataelement	VARCHAR(64)	Specifies the name of the data element.
input_data	LONG VARCHAR FOR BIT DATA	Specifies the input data for the UDF.
scid	INTEGER	Specifies the Security Coordinate ID. It is not used and should be set to zero.

Returns

Decrypted data as LONG VARCHAR FOR BIT DATA.

NULL: When user has no access to database.

Exception (and Error Codes)

Exception, if call fails.

Example

```
SELECT pty.ins_lvcfbd(0,'TE_A_S13_L1R2_Y',CAST('Protegrity' AS LONG VARCHAR FOR BIT
DATA),0) FROM SYSIBM.SYSDUMMY1;
```

4.1.8.5 pty.upd_lvcfbd

This function re-protects the LONG VARCHARFBD data with the data elements, such as, Tokenization and No Encryption method.

pty.upd_lvcfbd(comm_id INTEGER, dataelement VARCHAR, input_data LONG VARCHAR, scid INTEGER)

Parameters

Name	Type	Description
comm_id	INTEGER	Specifies the location where the UDF will find the policy. Must be the same as configured in the *pepserver.cfg* file.
dataelement	VARCHAR(64)	Specifies the name of the data element.
input_data	LONG VARCHAR FOR BIT DATA	Specifies the input data for the UDF.
scid	INTEGER	Specifies the Security Coordinate ID. It is not used and should be set to zero.

Returns

Decrypted data as LONG VARCHAR FOR BIT DATA.

NULL: When user has no access to database.

Exception

If set in policy and user does not have access, then the UDF terminates.

Example

```
SELECT pty.upd_lvcfbd(0,'TE_A_S13_L1R2_Y',CAST('Protegrity' AS LONG VARCHAR FOR BIT
DATA),0) FROM SYSIBM.SYSDUMMY1;
```

4.1.8.6 pty.sel_lvcfbd

This function unprotects the LONG VARCHARFBD data with the data elements, such as, Tokenization and No Encryption method.

pty.sel_lvcfbd(comm_id INTEGER, dataelement VARCHAR, input_data LONG VARCHAR, scid INTEGER)

Parameters

Name	Type	Description
comm_id	INTEGER	Specifies the location where the UDF will find the policy. Must be the same as configured in the *pepserver.cfg* file.
dataelement	VARCHAR(64)	Specifies the name of the data element.
input_data	LONG VARCHAR FOR BIT DATA	Specifies the input data for the UDF.
scid	INTEGER	Specifies the Security Coordinate ID. It is not used and should be set to zero.

Returns

Data as LONG VARCHAR FOR BIT DATA.

NULL: When user has no access to database.

Exception

If set in policy and user does not have access, then the UDF terminates.

Example

```
SELECT pty.sel_lvcfbd(0,'TE_A_S13_L1R2_Y',
pty.ins_lvcfbd(0,'TE_A_S13_L1R2_Y',CAST('Protegrity' AS LONG VARCHAR FOR BIT
DATA),0),0) FROM SYSIBM.SYSDUMMY1;
```

4.1.9 DATE UDFs

These UDFs are used to encrypt and decrypt DATE data.

4.1.9.1 pty.ins_enc_date

This function encrypts DATE object with a data element for encryption.

pty.ins_enc_date(comm_id INTEGER, dataelement VARCHAR, input_data DATE, scid INTEGER)

Parameters

Name	Type	Description
comm_id	INTEGER	Specifies the location where the UDF will find the policy. Must be the same as configured in the *pepserver.cfg* file.
dataelement	VARCHAR(64)	Specifies the name of the data element.
input_data	DATE	Specifies the input data for the UDF.
scid	INTEGER	Specifies the Security Coordinate ID. It is not used and should be set to zero.

Returns

Encrypted data as VARCHAR(34) FOR BIT DATA.

NULL: When user has no access to database.

Exception

If set in policy and user does not have access, then the UDF terminates.

Example

```
SELECT pty.ins_enc_date(0,'AES128',current_date,0) FROM SYSIBM.SYSDUMMY1;
```

4.1.9.2 pty.upd_enc_date

This re-encrypts the DATE object that has been updated with a data element for encryption.

pty.upd_enc_date(comm_id INTEGER, dataelement VARCHAR, input_data DATE, scid INTEGER)

Parameters

Name	Type	Description
comm_id	INTEGER	Specifies the location where the UDF will find the policy. Must be the same as configured in the *pepserver.cfg* file.
dataelement	VARCHAR(64)	Specifies the name of the data element.
input_data	DATE	Specifies the input data for the UDF.
scid	INTEGER	Specifies the Security Coordinate ID. It is not used and should be set to zero.

Returns

Encrypted data as VARCHAR(34) FOR BIT DATA.

NULL: When user has no access to database.

Exception

If set in policy and user does not have access, then the UDF terminates.

Example

```
SELECT pty.upd_enc_date(0,'AES128',current_date,0) FROM SYSIBM.SYSDUMMY1;
```

4.1.9.3 pty.sel_dec_date

This function decrypts the DATE object with a data element.

pty.sel_dec_date(comm_id INTEGER, dataelement VARCHAR, input_data VARCHAR, scid INTEGER)

Parameters

Name	Type	Description
comm_id	INTEGER	Specifies the location where the UDF will find the policy. Must be the same as configured in the *pepserver.cfg* file.
dataelement	VARCHAR(64)	Specifies the name of the data element.
input_data	VARCHAR(34) FOR BIT DATA	Specifies the input data for the UDF.
scid	INTEGER	Specifies the Security Coordinate ID. It is not used and should be set to zero.

Returns

Decrypted data as DATE.

NULL: When user has no access to database.

Exception

If set in policy and user does not have access, then the UDF terminates.

Example

```
SELECT PTY.sel_dec_date(0,'AES128',pty.ins_enc_date(0,'AES128',current_date,0),0) FROM
SYSIBM.SYSDUMMY1;
```

4.1.9.4 pty.ins_date

This function protects the DATE object with the data elements, such as, Tokenization and No Encryption method.

pty.ins_char(comm_id INTEGER, dataelement VARCHAR, input_data DATE, scid INTEGER)

Parameters

Name	Type	Description
comm_id	INTEGER	Specifies the location where the UDF will find the policy. Must be the same as configured in the *pepserver.cfg* file.
dataelement	VARCHAR(64)	Specifies the name of the data element.
input_data	DATE	Specifies the input data for the UDF.
scid	INTEGER	Specifies the Security Coordinate ID. It is not used and should be set to zero.

Returns

Returns data as DATE.

NULL: When user has no access to database.

Exception

If set in policy and user does not have access, then the UDF terminates.

Example

```
SELECT pty.ins_date(0,'TE_Date_YMD_S13',current_date,0) FROM SYSIBM.SYSDUMMY1;
```

4.1.9.5 pty.upd_date

This function re-protects the DATE object with the data elements, such as, Tokenization and No Encryption method.

pty.upd_char(comm_id INTEGER, dataelement VARCHAR, input_data DATE, scid INTEGER)

Parameters

Name	Type	Description
comm_id	INTEGER	Specifies the location where the UDF will find the policy. Must be the same as configured in the *pepserver.cfg* file.
dataelement	VARCHAR(64)	Specifies the name of the data element.
input_data	DATE	Specifies the input data for the UDF.
scid	INTEGER	Specifies the Security Coordinate ID. It is not used and should be set to zero.

Returns

Data as DATE.

NULL: When user has no access to database.

Exception

If set in policy and user does not have access, then the UDF terminates.

Example

```
SELECT pty.upd_date(0,'TE_Date_YMD_S13',current_date,0) FROM SYSIBM.SYSDUMMY1;
```

4.1.9.6 pty.sel_date

This function unprotects the DATE object with the data elements, such as, Tokenization and No Encryption method.

pty.sel_char(comm_id INTEGER, dataelement VARCHAR, input_data DATE, scid INTEGER)

Parameters

Name	Type	Description
comm_id	INTEGER	Specifies the location where the UDF will find the policy. Must be the same as configured in the *pepserver.cfg* file.
dataelement	VARCHAR(64)	Specifies the name of the data element.

Name	Type	Description
input_data	DATE	Specifies the input data for the UDF.
scid	INTEGER	Specifies the Security Coordinate ID. It is not used and should be set to zero.

Returns

Returns data as DATE.

NULL: When user has no access to database.

Exception

If set in policy and user does not have access, then the UDF terminates.

Example

```
SELECT
pty.sel_date(0,'TE_Date_YMD_S13',pty.ins_date(0,'TE_Date_YMD_S13',current_date,0),0)
FROM SYSIBM.SYSDUMMY1;
```

4.1.10 TIMESTAMP UDFs

These UDFs are used to encrypt and decrypt TIMESTAMP data.

4.1.10.1 pty.ins_enc_timestamp

This function encrypts TIMESTAMP object with a data element for encryption.

pty.ins_enc_timestamp(comm_id INTEGER, dataelement VARCHAR, input_data TIMESTAMP, scid INTEGER)

Parameters

Name	Type	Description
comm_id	INTEGER	Specifies the location where the UDF will find the policy. Must be the same as configured in the *pepserver.cfg* file.
dataelement	VARCHAR(64)	Specifies the name of the data element.
input_data	TIMESTAMP	Specifies the input data for the UDF.
scid	INTEGER	Specifies the Security Coordinate ID. It is not used and should be set to zero.

Returns

Encrypted data as VARCHAR(50) FOR BIT DATA.

NULL: When user has no access to database.

Exception

If set in policy and user does not have access, then the UDF terminates.

Example

```
SELECT pty.ins_enc_timestamp(0, 'AES128', current_timestamp, 10) FROM SYSIBM.SYSDUMMY1;
```

4.1.10.2 pty.upd_enc_timestamp

This re-encrypts the TIMESTAMP object that has been updated with a data element for encryption.

pty.upd_enc_timestamp(comm_id INTEGER, dataelement VARCHAR, input_data TIMESTAMP, scid INTEGER)

Parameters

Name	Type	Description
comm_id	INTEGER	Specifies the location where the UDF will find the policy. Must be the same as configured in the *pepserver.cfg* file.
dataelement	VARCHAR(64)	Specifies the name of the data element.
input_data	TIMESTAMP	Specifies the input data for the UDF.
scid	INTEGER	Specifies the Security Coordinate ID. It is not used and should be set to zero.

Returns

Encrypted data as VARCHAR(50) FOR BIT DATA.

NULL: When user has no access to database.

Exception

If set in policy and user does not have access, then the UDF terminates.

Example

```
SELECT pty.upd_enc_timestamp(0, 'AES128', current_timestamp, 10) FROM SYSIBM.SYSDUMMY1;
```

4.1.10.3 pty.sel_dec_timestamp

This function decrypts the TIMESTAMP object with a data element.

pty.upd_enc_timestamp(comm_id INTEGER, dataelement VARCHAR, input_data VARCHAR, scid INTEGER)

Parameters

Name	Type	Description
comm_id	INTEGER	Specifies the location where the UDF will find the policy. Must be the same as configured in the *pepserver.cfg* file.
dataelement	VARCHAR(64)	Specifies the name of the data element.
input_data	VARCHAR(50) FOR BIT DATA	Specifies the input data for the UDF.
scid	INTEGER	Specifies the Security Coordinate ID. It is not used and should be set to zero.

Returns

Decrypted data as TIMESTAMP.

NULL: When user has no access to database.

Exception

If set in policy and user does not have access, then the UDF terminates.

Example

```
SELECT pty.sel_dec_timestamp(0, 'AES128', pty.ins_enc_timestamp(0, 'AES128',
current_timestamp, 10), 10) FROM SYSIBM.SYSDUMMY1;
```

4.1.10.4 pty.ins_timestamp

This function protects the TIMESTAMP object with the No Encryption data element.

pty.ins_timestamp(comm_id INTEGER, dataelement VARCHAR, input_data TIMESTAMP, scid INTEGER)

Parameters

Name	Type	Description
comm_id	INTEGER	Specifies the location where the UDF will find the policy. Must be the same as configured in the *pepserver.cfg* file.
dataelement	VARCHAR(64)	Specifies the name of the data element.
input_data	TIMESTAMP	Specifies the input data for the UDF.
scid	INTEGER	Specifies the Security Coordinate ID. It is not used and should be set to zero.

Returns

Data as TIMESTAMP.

NULL: When user has no access to database.

Exception

If set in policy and user does not have access, then the UDF terminates.

Example

```
SELECT pty.ins_timestamp(0,'NoEncryption',current_timestamp,0) FROM SYSIBM.SYSDUMMY1;
```

4.1.10.5 pty.upd_timestamp

This function re-protects the TIMESTAMP object with the No Encryption data element.

pty.upd_timestamp(comm_id INTEGER, dataelement VARCHAR, input_data TIMESTAMP, scid INTEGER)

Parameters

Name	Type	Description
comm_id	INTEGER	Specifies the location where the UDF will find the policy. Must be the same as configured in the *pepserver.cfg* file.
dataelement	VARCHAR(64)	Specifies the name of the data element.

Name	Type	Description
input_data	TIMESTAMP	Specifies the input data for the UDF.
scid	INTEGER	Specifies the Security Coordinate ID. It is not used and should be set to zero.

Returns

Data as TIMESTAMP.

NULL: When user has no access to database.

Exception

If set in policy and user does not have access, then the UDF terminates.

Example

```
SELECT pty.upd_timestamp(0, 'NoEncryption', current_timestamp, 10) FROM
SYSIBM.SYSDUMMY1;
```

4.1.10.6 pty.sel_timestamp

This function unprotects the TIMESTAMP object with the No Encryption data element.

pty.sel_timestamp(comm_id INTEGER, dataelement VARCHAR, input_data TIMESTAMP, scid INTEGER)

Parameters

Name	Type	Description
comm_id	INTEGER	Specifies the location where the UDF will find the policy. Must be the same as configured in the *pepserver.cfg* file.
dataelement	VARCHAR(64)	Specifies the name of the data element.
input_data	TIMESTAMP	Specifies the input data for the UDF.
scid	INTEGER	Specifies the Security Coordinate ID. It is not used and should be set to zero.

Returns

Data as TIMESTAMP.

NULL: When user has no access to database.

Exception

If set in policy and user does not have access, then the UDF terminates.

Example

```
SELECT pty.sel_timestamp(0, 'NoEncryption', pty.ins_timestamp(0, 'NoEncryption',
current_timestamp, 10), 10) FROM SYSIBM.SYSDUMMY1;
```

4.1.11 TIME UDFs

These UDFs are used to encrypt and decrypt TIME data.

4.1.11.1 pty.ins_enc_time

This function encrypts TIME object with a data element for encryption.

pty.ins_enc_time(comm_id INTEGER, dataelement VARCHAR, input_data TIME, scid INTEGER)

Parameters

Name	Type	Description
comm_id	INTEGER	Specifies the location where the UDF will find the policy. Must be the same as configured in the *pepserver.cfg* file.
dataelement	VARCHAR(64)	Specifies the name of the data element.
input_data	TIME	Specifies the input data for the UDF.
scid	INTEGER	Specifies the Security Coordinate ID. It is not used and should be set to zero.

Returns

Encrypted data as VARCHAR(34) FOR BIT DATA.

NULL: When user has no access to database.

Exception

If set in policy and user does not have access, then the UDF terminates.

Example

```
SELECT pty.ins_enc_time(0,'AES128',current_time,0) FROM SYSIBM.SYSDUMMY1;
```

4.1.11.2 pty.upd_enc_time

This re-encrypts the TIME object that has been updated with a data element for encryption.

pty.upd_enc_time(comm_id INTEGER, dataelement VARCHAR, input_data TIME, scid INTEGER)

Parameters

Name	Type	Description
comm_id	INTEGER	Specifies the location where the UDF will find the policy. Must be the same as configured in the *pepserver.cfg* file.
dataelement	VARCHAR(64)	Specifies the name of the data element.
input_data	TIME	Specifies the input data for the UDF.
scid	INTEGER	Specifies the Security Coordinate ID. It is not used and should be set to zero.

Returns

Encrypted data as VARCHAR(34) FOR BIT DATA.

NULL: When user has no access to database.

Exception

If set in policy and user does not have access, then the UDF terminates.

Example

```
SELECT pty.upd_enc_time(0,'AES128',current_time,0) FROM SYSIBM.SYSDUMMY1;
```

4.1.11.3 pty.sel_dec_time

This function decrypts the TIME object with a data element.

pty.sel_dec_time(comm_id INTEGER, dataelement VARCHAR, input_data VARCHAR, scid INTEGER)

Parameters

Name	Type	Description
comm_id	INTEGER	Specifies the location where the UDF will find the policy. Must be the same as configured in the *pepserver.cfg* file.
dataelement	VARCHAR(64)	Specifies the name of the data element.
input_data	VARCHAR(34) FOR BIT DATA	Specifies the input data for the UDF.
scid	INTEGER	Specifies the Security Coordinate ID. It is not used and should be set to zero.

Returns

Decrypted data as TIME.

NULL: When user has no access to database.

Exception

If set in policy and user does not have access, then the UDF terminates.

Example

```
SELECT pty.sel_dec_time(0,'AES128',pty.ins_enc_time(0,'AES128',current_time,0),0) FROM
SYSIBM.SYSDUMMY1;
```

4.1.11.4 pty.ins_time

This function protects the TIME object with the No Encryption data element.

pty.ins_time(comm_id INTEGER, dataelement VARCHAR, input_data TIME, scid INTEGER)

Parameters

Name	Type	Description
comm_id	INTEGER	Specifies the location where the UDF will find the policy. Must be the same as configured in the *pepserver.cfg* file.
dataelement	VARCHAR(64)	Specifies the name of the data element.
input_data	TIME	Specifies the input data for the UDF.

Name	Type	Description
scid	INTEGER	Specifies the Security Coordinate ID. It is not used and should be set to zero.

Returns

Data as TIME.

NULL: When user has no access to database.

Exception

If set in policy and user does not have access, then the UDF terminates.

Example

```
SELECT pty.ins_time(0,'NoEncryption',current_time,0) FROM SYSIBM.SYSDUMMY1;
```

4.1.11.5 pty.upd_time

This function re-protects the TIME object with the No Encryption data element.

pty.upd_time(comm_id INTEGER, dataelement VARCHAR, input_data TIME, scid INTEGER)

Parameters

Name	Type	Description
comm_id	INTEGER	Specifies the location where the UDF will find the policy. Must be the same as configured in the *pepserver.cfg* file.
dataelement	VARCHAR(64)	Specifies the name of the data element.
input_data	TIME	Specifies the input data for the UDF.
scid	INTEGER	Specifies the Security Coordinate ID. It is not used and should be set to zero.

Returns

Data as TIME.

NULL: When user has no access to database.

Exception

If set in policy and user does not have access, then the UDF terminates.

Example

```
SELECT pty.upd_time(0,'NoEncryption',current_time,0) FROM SYSIBM.SYSDUMMY1;
```

4.1.11.6 pty.sel_time

This function unprotects the TIME object with the No Encryption data element.

pty.sel_time(comm_id INTEGER, dataelement VARCHAR, input_data TIME, scid INTEGER)

Parameters

Name	Type	Description
comm_id	INTEGER	Specifies the location where the UDF will find the policy. Must be the same as configured in the *pepserver.cfg* file.
dataelement	VARCHAR(64)	Specifies the name of the data element.
input_data	TIME	Specifies the input data for the UDF.
scid	INTEGER	Specifies the Security Coordinate ID. It is not used and should be set to zero.

Returns

Data as TIME.

NULL: When user has no access to database.

Exception

If set in policy and user does not have access, then the UDF terminates.

Example

```
SELECT pty.sel_time(0,'NoEncryption',pty.ins_time(0,'NoEncryption',current_time,0),0)
FROM SYSIBM.SYSDUMMY1;
```

4.1.12 INTEGER UDFs

These UDFs are used to encrypt and decrypt INTEGER data.

4.1.12.1 pty.ins_enc_integer

This function encrypts INTEGER data with a data element for encryption.

pty.ins_enc_integer(comm_id INTEGER, dataelement VARCHAR, input_data TIME, scid INTEGER)

Parameters

Name	Type	Description
comm_id	INTEGER	Specifies the location where the UDF will find the policy. Must be the same as configured in the *pepserver.cfg* file.
dataelement	VARCHAR(64)	Specifies the name of the data element.
input_data	INTEGER	Specifies the input data for the UDF.
scid	INTEGER	Specifies the Security Coordinate ID. It is not used and should be set to zero.

Returns

Encrypted data as VARCHAR(34) FOR BIT DATA.

NULL: When user has no access to database.

Exception

If set in policy and user does not have access, then the UDF terminates.

Example

```
SELECT pty.ins_enc_integer(0, 'AES128', CAST(123456 AS INTEGER), 10) FROM
SYSIBM.SYSDUMMY1;
```

4.1.12.2 pty.upd_enc_integer

This re-encrypts the INTEGER data that has been updated with a data element for encryption.

pty.upd_enc_integer(comm_id INTEGER, dataelement VARCHAR, input_data INTEGER, scid INTEGER)

Parameters

Name	Type	Description
comm_id	INTEGER	Specifies the location where the UDF will find the policy. Must be the same as configured in the *pepserver.cfg* file.
dataelement	VARCHAR(64)	Specifies the name of the data element.
input_data	INTEGER	Specifies the input data for the UDF.
scid	INTEGER	Specifies the Security Coordinate ID. It is not used and should be set to zero.

Returns

Encrypted data as VARCHAR(34) FOR BIT DATA.

NULL: When user has no access to database.

Exception

If set in policy and user does not have access, then the UDF terminates.

Example

```
SELECT pty.upd_enc_integer(0, 'AES128', CAST(123456 AS INTEGER), 10) FROM
SYSIBM.SYSDUMMY1;
```

4.1.12.3 pty.sel_dec_integer

This function decrypts the INTEGER data with a data element.

pty.sel_dec_integer(comm_id INTEGER, dataelement VARCHAR, input_data VARCHAR, scid INTEGER)

Parameters

Name	Type	Description
comm_id	INTEGER	Specifies the location where the UDF will find the policy. Must be the same as configured in the *pepserver.cfg* file.
dataelement	VARCHAR(64)	Specifies the name of the data element.
input_data	VARCHAR(34) FOR BIT DATA	Specifies the input data for the UDF.

Name	Type	Description
scid	INTEGER	Specifies the Security Coordinate ID. It is not used and should be set to zero.

Returns

Decrypted data as INTEGER.

NULL: When user has no access to database.

Exception

If set in policy and user does not have access, then the UDF terminates.

Example

```
SELECT pty.sel_dec_integer(0, 'AES128', pty.ins_enc_integer(0, 'AES128', CAST(123456 AS
INTEGER), 10), 10) FROM SYSIBM.SYSDUMMY1;
```

4.1.12.4 pty.ins_integer

This function protects the INTEGER data with the data elements, such as, Tokenization and No Encryption method.

pty.ins_integer(comm_id INTEGER, dataelement VARCHAR, input_data INTEGER, scid INTEGER)

Parameters

Name	Type	Description
comm_id	INTEGER	Specifies the location where the UDF will find the policy. Must be the same as configured in the *pepserver.cfg* file.
dataelement	VARCHAR(64)	Specifies the name of the data element.
input_data	INTEGER	Specifies the input data for the UDF.
scid	INTEGER	Specifies the Security Coordinate ID. It is not used and should be set to zero.

Returns

Data as INTEGER.

NULL: When user has no access to database.

Exception (and Error Codes)

Exception, if call fails.

Example

```
SELECT pty.ins_integer(0, 'NoEncryption', CAST(123456 AS INTEGER), 10) FROM
SYSIBM.SYSDUMMY1;
```

4.1.12.5 pty.upd_integer

This function re-protects the INTEGER data with the data elements, such as, Tokenization and No Encryption method.

pty.upd_integer(comm_id INTEGER, dataelement VARCHAR, input_data INTEGER, scid INTEGER)

Parameters

Name	Type	Description
comm_id	INTEGER	Specifies the location where the UDF will find the policy. Must be the same as configured in the *pepserver.cfg* file.
dataelement	VARCHAR(64)	Specifies the name of the data element.
input_data	INTEGER	Specifies the input data for the UDF.
scid	INTEGER	Specifies the Security Coordinate ID. It is not used and should be set to zero.

Returns

Data as INTEGER.

NULL: When user has no access to database.

Exception

If set in policy and user does not have access, then the UDF terminates.

Example

```
SELECT pty.upd_integer(0, 'NoEncryption', CAST(123456 AS INTEGER), 10) FROM
SYSIBM.SYSDUMMY1;
```

4.1.12.6 pty.sel_integer

TThis function unprotects the INTEGER data with the data elements, such as, Tokenization and No Encryption method.

pty.sel_integer(comm_id INTEGER, dataelement VARCHAR, input_data INTEGER, scid INTEGER)

Parameters

Name	Type	Description
comm_id	INTEGER	Specifies the location where the UDF will find the policy. Must be the same as configured in the *pepserver.cfg* file.
dataelement	VARCHAR(64)	Specifies the name of the data element.
input_data	INTEGER	Specifies the input data for the UDF.
scid	INTEGER	Specifies the Security Coordinate ID. It is not used and should be set to zero.

Returns

Data as INTEGER.

NULL: When user has no access to database.

Exception

If set in policy and user does not have access, then the UDF terminates.

Example

```
SELECT pty.sel_integer(0, 'NoEncryption', pty.ins_integer(0, 'NoEncryption',
CAST(123456 AS INTEGER), 10), 10) FROM SYSIBM.SYSDUMMY1;
```

4.1.13 SMALLINT UDFs

These UDFs are used to encrypt and decrypt SMALLINT data.

4.1.13.1 pty.ins_enc_smallint

This function encrypts SMALL INTEGER data with a data element for encryption.

pty.ins_enc_smallint(comm_id INTEGER, dataelement VARCHAR, input_data SMALLINT, scid INTEGER)

Parameters

Name	Type	Description
comm_id	INTEGER	Specifies the location where the UDF will find the policy. Must be the same as configured in the *pepserver.cfg* file.
dataelement	VARCHAR(64)	Specifies the name of the data element.
input_data	SMALLINT	Specifies the input data for the UDF.
scid	INTEGER	Specifies the Security Coordinate ID. It is not used and should be set to zero.

Returns

Encrypted data as VARCHAR(34) FOR BIT DATA.

NULL: When user has no access to database.

Exception

If set in policy and user does not have access, then the UDF terminates.

Example

```
SELECT pty.ins_enc_smallint(0, 'AES128', CAST(1234 AS SMALLINT), 10) FROM
SYSIBM.SYSDUMMY1;
```

4.1.13.2 pty.upd_enc_smallint

This re-encrypts the SMALL INTEGER data that has been updated with a data element for encryption.

pty.upd_enc_smallint(comm_id INTEGER, dataelement VARCHAR, input_data SMALLINT, scid INTEGER)

Parameters

Name	Type	Description
comm_id	INTEGER	Specifies the location where the UDF will find the policy. Must be the same as configured in the *pepserver.cfg* file.
dataelement	VARCHAR(64)	Specifies the name of the data element.

Name	Type	Description
input_data	SMALLINT	Specifies the input data for the UDF.
scid	INTEGER	Specifies the Security Coordinate ID. It is not used and should be set to zero.

Returns

Encrypted data as VARCHAR(34) FOR BIT DATA.

NULL: When user has no access to database.

Exception

If set in policy and user does not have access, then the UDF terminates.

Example

```
SELECT pty.upd_enc_smallint(0, 'AES128', CAST(1234 AS SMALLINT), 10) FROM
SYSIBM.SYSDUMMY1;
```

4.1.13.3 pty.sel_dec_smallint

This function decrypts the SMALL INTEGER data with a data element.

pty.sel_dec_smallint(comm_id INTEGER, dataelement VARCHAR, input_data VARCHAR, scid INTEGER)

Parameters

Name	Type	Description
comm_id	INTEGER	Specifies the location where the UDF will find the policy. Must be the same as configured in the *pepserver.cfg* file.
dataelement	VARCHAR(64)	Specifies the name of the data element.
input_data	VARCHAR(34) FOR BIT DATA	Specifies the input data for the UDF.
scid	INTEGER	Specifies the Security Coordinate ID. It is not used and should be set to zero.

Returns

Decrypted data as SMALLINT.

NULL: When user has no access to database.

Exception

If set in policy and user does not have access, then the UDF terminates.

Example

```
SELECT pty.sel_dec_smallint(0, 'AES128', pty.ins_enc_smallint(0, 'AES128', CAST(1234 AS
SMALLINT), 10), 10) FROM SYSIBM.SYSDUMMY1;
```

4.1.13.4 pty.ins_smallint

This function protects the SMALL INTEGER data with the data elements, such as, Tokenization and No Encryption method.

pty.ins_smallint(comm_id INTEGER, dataelement VARCHAR, input_data SMALLINT, scid INTEGER)

Parameters

Name	Type	Description
comm_id	INTEGER	Specifies the location where the UDF will find the policy. Must be the same as configured in the *pepserver.cfg* file.
dataelement	VARCHAR(64)	Specifies the name of the data element.
input_data	SMALLINT	Specifies the input data for the UDF.
scid	INTEGER	Specifies the Security Coordinate ID. It is not used and should be set to zero.

Returns

Data as SMALLINT.

NULL: When user has no access to database.

Exception

If set in policy and user does not have access, then the UDF terminates.

Example for Tokenization

```
SELECT pty.ins_smallint(0, 'TE_INT_SLT13_2', 1234, 10) FROM SYSIBM.SYSDUMMY1;
```

Example for No Encryption

```
SELECT pty.ins_smallint(0, 'NoEncryption', CAST(1234 AS SMALLINT), 10) FROM
SYSIBM.SYSDUMMY1;
```

4.1.13.5 pty.upd_smallint

This function re-protects the SMALL INTEGER data with the data elements, such as, Tokenization and No Encryption method.

pty.upd_smallint(comm_id INTEGER, dataelement VARCHAR, input_data SMALLINT, scid INTEGER)

Parameters

Name	Type	Description
comm_id	INTEGER	Specifies the location where the UDF will find the policy. Must be the same as configured in the *pepserver.cfg* file.
dataelement	VARCHAR(64)	Specifies the name of the data element.
input_data	SMALLINT	Specifies the input data for the UDF.
scid	INTEGER	Specifies the Security Coordinate ID. It is not used and should be set to zero.

Returns

Data as SMALLINT.

NULL: When user has no access to database.

Exception

If set in policy and user does not have access, then the UDF terminates.

Example for Tokenization

```
SELECT pty.upd_smallint(0, 'TE_INT_SLT13_2', CAST(1234 AS SMALLINT), 10) FROM
SYSIBM.SYSDUMMY1;
```

Example for No Encryption

```
SELECT pty.upd_smallint(0, 'NoEncryption', CAST(1234 AS SMALLINT), 10) FROM
SYSIBM.SYSDUMMY1;
```

4.1.13.6 pty.sel_smallint

This function unprotects the SMALL INTEGER data with the data elements, such as, Tokenization and No Encryption method.

pty.sel_smallint(comm_id INTEGER, dataelement VARCHAR, input_data SMALLINT, scid INTEGER)

Parameters

Name	Type	Description
comm_id	INTEGER	Specifies the location where the UDF will find the policy. Must be the same as configured in the *pepserver.cfg* file.
dataelement	VARCHAR(64)	Specifies the name of the data element.
input_data	SMALLINT	Specifies the input data for the UDF.
scid	INTEGER	Specifies the Security Coordinate ID. It is not used and should be set to zero.

Returns

Data as SMALLINT.

NULL: When user has no access to database.

Exception

If set in policy and user does not have access, then the UDF terminates.

Example for Tokenization

```
SELECT pty.sel_smallint(0, 'TE_INT_SLT13_2', pty.upd_smallint(0, 'TE_INT_SLT13_2',
CAST(1234 AS SMALLINT), 10), 10) FROM SYSIBM.SYSDUMMY1;
```

Example for No Encryption

```
SELECT pty.sel_smallint(0, 'NoEncryption', pty.upd_smallint(0, 'NoEncryption',
CAST(1234 AS SMALLINT), 10), 10) FROM SYSIBM.SYSDUMMY1;
```

4.1.14 BIGINT UDFs

These UDFs are used to encrypt and decrypt BIGINT data.

4.1.14.1 pty.ins_enc_bigint

This function encrypts BIG INTEGER data with a data element for encryption.

pty.ins_enc_bigint(comm_id INTEGER, dataelement VARCHAR, input_data BIGINT, scid INTEGER)

Parameters

Name	Type	Description
comm_id	INTEGER	Specifies the location where the UDF will find the policy. Must be the same as configured in the *pepserver.cfg* file.
dataelement	VARCHAR(64)	Specifies the name of the data element.
input_data	BIGINT	Specifies the input data for the UDF.
scid	INTEGER	Specifies the Security Coordinate ID. It is not used and should be set to zero.

Returns

Encrypted data as VARCHAR(34) FOR BIT DATA.

NULL: When user has no access to database.

Exception

If set in policy and user does not have access, then the UDF terminates.

Example

```
SELECT pty.ins_enc_bigint(0, 'AES128', CAST(12345678 AS BIGINT), 10) FROM
SYSIBM.SYSDUMMY1;
```

4.1.14.2 pty.upd_enc_bigint

This re-encrypts the BIG INTEGER data that has been updated with a data element for encryption.

pty.upd_enc_bigint(comm_id INTEGER, dataelement VARCHAR, input_data BIGINT, scid INTEGER)

Parameters

Name	Type	Description
comm_id	INTEGER	Specifies the location where the UDF will find the policy. Must be the same as configured in the *pepserver.cfg* file.
dataelement	VARCHAR(64)	Specifies the name of the data element.
input_data	BIGINT	Specifies the input data for the UDF.
scid	INTEGER	Specifies the Security Coordinate ID. It is not used and should be set to zero.

Returns

Encrypted data as VARCHAR(34) FOR BIT DATA.

NULL: When user has no access to database.

Exception

If set in policy and user does not have access, then the UDF terminates.

Example

```
SELECT pty.upd_enc_bigint(0, 'AES128', CAST(12345678 AS BIGINT), 10) FROM
SYSIBM.SYSDUMMY1;
```

4.1.14.3 pty.sel_dec_bigint

This function decrypts the BIG INTEGER data with a data element.

pty.sel_dec_bigint(comm_id INTEGER, dataelement VARCHAR, input_data VARCHAR, scid INTEGER)

Parameters

Name	Type	Description
comm_id	INTEGER	Specifies the location where the UDF will find the policy. Must be the same as configured in the *pepserver.cfg* file.
dataelement	VARCHAR(64)	Specifies the name of the data element.
input_data	VARCHAR(34) FOR BIT DATA	Specifies the input data for the UDF.
scid	INTEGER	Specifies the Security Coordinate ID. It is not used and should be set to zero.

Returns

Decrypted data as BIGINT.

NULL: When user has no access to database.

Exception

If set in policy and user does not have access, then the UDF terminates.

Example

```
SELECT pty.sel_dec_bigint(0, 'AES128', pty.ins_enc_bigint(0, 'AES128', CAST(12345678 AS
BIGINT), 10), 10) FROM SYSIBM.SYSDUMMY1;
```

4.1.14.4 pty.ins_bigint

This function protects the BIG INTEGER data with the data elements, such as, Tokenization and No Encryption method.

pty.ins_bigint(comm_id INTEGER, dataelement VARCHAR, input_data BIGINT, scid INTEGER)

Parameters

Name	Type	Description
comm_id	INTEGER	Specifies the location where the UDF will find the policy. Must be the same as configured in the *pepserver.cfg* file.

Name	Type	Description
dataelement	VARCHAR(64)	Specifies the name of the data element.
input_data	BIGINT	Specifies the input data for the UDF.
scid	INTEGER	Specifies the Security Coordinate ID. It is not used and should be set to zero.

Returns

Data as BIGINT.

NULL: When user has no access to database.

Exception

If set in policy and user does not have access, then the UDF terminates.

Example for Tokenization

```
SELECT pty.ins_bigint(0, 'TE_INT_8_BIGINTINS', CAST(12345678 AS BIGINT), 10) FROM
SYSIBM.SYSDUMMY1;
```

Example for No Encryption

```
SELECT pty.ins_bigint(0, 'NoEncryption', CAST(12345678 AS BIGINT), 10) FROM
SYSIBM.SYSDUMMY1;
```

4.1.14.5 pty.upd_bigint

This function re-protects the BIG INTEGER data with the data elements, such as, Tokenization and No Encryption method.

pty.upd_bigint(comm_id INTEGER, dataelement VARCHAR, input_data BIGINT, scid INTEGER)

Parameters

Name	Type	Description
comm_id	INTEGER	Specifies the location where the UDF will find the policy. Must be the same as configured in the *pepserver.cfg* file.
dataelement	VARCHAR(64)	Specifies the name of the data element.
input_data	BIGINT	Specifies the input data for the UDF.
scid	INTEGER	Specifies the Security Coordinate ID. It is not used and should be set to zero.

Returns

Data as BIGINT.

NULL: When user has no access to database.

Exception

If set in policy and user does not have access, then the UDF terminates.

Example for Tokenization

```
SELECT pty.upd_bigint(0, 'TE_INT_8_BIGINT', CAST(12345678 AS BIGINT), 10) FROM
SYSIBM.SYSDUMMY1;
```

Example for No Encryption

```
SELECT pty.upd_bigint(0, 'NoEncryption', CAST(12345678 AS BIGINT), 10) FROM
SYSIBM.SYSDUMMY1;
```

4.1.14.6 pty.sel_bigint

This function unprotects the BIG INTEGER data with the data elements, such as, Tokenization and No Encryption method.

pty.sel_bigint(comm_id INTEGER, dataelement VARCHAR, input_data BIGINT, scid INTEGER)

Parameters

Name	Type	Description
comm_id	INTEGER	Specifies the location where the UDF will find the policy. Must be the same as configured in the *pepserver.cfg* file.
dataelement	VARCHAR(64)	Specifies the name of the data element.
input_data	BIGINT	Specifies the input data for the UDF.
scid	INTEGER	Specifies the Security Coordinate ID. It is not used and should be set to zero.

Returns

Data as BIGINT.

NULL: When user has no access to database.

Exception

If set in policy and user does not have access, then the UDF terminates.

Example for Tokenization

```
SELECT pty.sel_bigint(0, 'TE_INT_8_BIGINT', pty.ins_bigint(0, 'TE_INT_8_BIGINT',
CAST(12345678 AS BIGINT), 10), 10) FROM SYSIBM.SYSDUMMY1;
```

Example for No Encryption

```
SELECT pty.sel_bigint(0, 'NoEncryption', pty.ins_bigint(0, 'NoEncryption',
CAST(12345678 AS BIGINT), 10), 10) FROM SYSIBM.SYSDUMMY1;
```

4.1.15 REAL UDFs

These UDFs are used to encrypt and decrypt REAL data.

4.1.15.1 pty.ins_enc_real

This function encrypts REAL data with a data element for encryption.

pty.ins_enc_real(comm_id INTEGER, dataelement VARCHAR, input_data REAL, scid INTEGER)

Parameters

Name	Type	Description
comm_id	INTEGER	Specifies the location where the UDF will find the policy. Must be the same as configured in the *pepserver.cfg* file.
dataelement	VARCHAR(64)	Specifies the name of the data element.
input_data	REAL	Specifies the input data for the UDF.
scid	INTEGER	Specifies the Security Coordinate ID. It is not used and should be set to zero.

Returns

Encrypted data as VARCHAR(34) FOR BIT DATA.

NULL: When user has no access to database.

Exception

If set in policy and user does not have access, then the UDF terminates.

Example

```
SELECT pty.ins_enc_real(0, 'AES128', CAST(123456 AS REAL), 10) FROM SYSIBM.SYSDUMMY1;
```

4.1.15.2 pty.upd_enc_real

This re-encrypts the REAL data that has been updated with a data element for encryption.

pty.upd_enc_real(comm_id INTEGER, dataelement VARCHAR, input_data REAL, scid INTEGER)

Parameters

Name	Type	Description
comm_id	INTEGER	Specifies the location where the UDF will find the policy. Must be the same as configured in the *pepserver.cfg* file.
dataelement	VARCHAR(64)	Specifies the name of the data element.
input_data	REAL	Specifies the input data for the UDF.
scid	INTEGER	Specifies the Security Coordinate ID. It is not used and should be set to zero.

Returns

Encrypted data as VARCHAR(34) FOR BIT DATA.

NULL: When user has no access to database.

Exception

If set in policy and user does not have access, then the UDF terminates.

Example

```
SELECT pty.upd_enc_real(0, 'AES128', CAST(123456 AS REAL), 10) FROM SYSIBM.SYSDUMMY1;
```

4.1.15.3 pty.sel_dec_real

This function decrypts the REAL data with a data element.

pty.sel_dec_real(comm_id INTEGER, dataelement VARCHAR, input_data VARCHAR, scid INTEGER)

Parameters

Name	Type	Description
comm_id	INTEGER	Specifies the location where the UDF will find the policy. Must be the same as configured in the *pepserver.cfg* file.
dataelement	VARCHAR(64)	Specifies the name of the data element.
input_data	VARCHAR(34) FOR BIT DATA	Specifies the input data for the UDF.
scid	INTEGER	Specifies the Security Coordinate ID. It is not used and should be set to zero.

Returns

Decrypted data as REAL.

NULL: When user has no access to database.

Exception

If set in policy and user does not have access, then the UDF terminates.

Example

```
SELECT pty.sel_dec_real(0, 'AES128', pty.ins_enc_real(0, 'AES128', CAST(123456 AS
REAL), 10), 10) FROM SYSIBM.SYSDUMMY1
```

4.1.15.4 pty.ins_real

This function protects the REAL data with the No Encryption data element.

pty.ins_real(comm_id INTEGER, dataelement VARCHAR, input_data REAL, scid INTEGER)

Parameters

Name	Type	Description
comm_id	INTEGER	Specifies the location where the UDF will find the policy. Must be the same as configured in the *pepserver.cfg* file.
dataelement	VARCHAR(64)	Specifies the name of the data element.
input_data	REAL	Specifies the input data for the UDF.
scid	INTEGER	Specifies the Security Coordinate ID. It is not used and should be set to zero.

Returns

Data as REAL.

NULL: When user has no access to database.

Exception

If set in policy and user does not have access, then the UDF terminates.

Example

```
SELECT pty.ins_real(0, 'NoEncryption', CAST(123456 AS REAL), 10) FROM SYSIBM.SYSDUMMY1;
```

4.1.15.5 pty.upd_real

This function re-protects the REAL data with the No Encryption data element.

pty.upd_real(comm_id INTEGER, dataelement VARCHAR, input_data REAL, scid INTEGER)

Parameters

Name	Type	Description
comm_id	INTEGER	Specifies the location where the UDF will find the policy. Must be the same as configured in the *pepserver.cfg* file.
dataelement	VARCHAR(64)	Specifies the name of the data element.
input_data	REAL	Specifies the input data for the UDF.
scid	INTEGER	Specifies the Security Coordinate ID. It is not used and should be set to zero.

Returns

Data as REAL.

NULL: When user has no access to database.

Exception

If set in policy and user does not have access, then the UDF terminates.

Example

```
SELECT pty.upd_real(0, 'NoEncryption', CAST(123456 AS REAL), 10) FROM SYSIBM.SYSDUMMY1;
```

4.1.15.6 pty.sel_real

This function unprotects the REAL data with the No Encryption data element.

pty.sel_real(comm_id INTEGER, dataelement VARCHAR, input_data REAL, scid INTEGER)

Parameters

Name	Type	Description
comm_id	INTEGER	Specifies the location where the UDF will find the policy. Must be the same as configured in the *pepserver.cfg* file.
dataelement	VARCHAR(64)	Specifies the name of the data element.

Name	Type	Description
input_data	REAL	Specifies the input data for the UDF.
scid	INTEGER	Specifies the Security Coordinate ID. It is not used and should be set to zero.

Returns

Data as REAL.

NULL: When user has no access to database.

Exception

If set in policy and user does not have access, then the UDF terminates.

Example

```
SELECT pty.sel_real(0, 'NoEncryption', pty.ins_real(0, 'NoEncryption', CAST(123456 AS
REAL), 10), 10 ) FROM SYSIBM.SYSDUMMY1;
```

4.1.16 DOUBLE UDFs

These UDFs are used to encrypt and decrypt DOUBLE data.

4.1.16.1 pty.ins_enc_double

This function encrypts the DOUBLE data with a data element for encryption.

pty.ins_enc_double(comm_id INTEGER, dataelement VARCHAR, input_data DOUBLE, scid INTEGER)

Parameters

Name	Type	Description
comm_id	INTEGER	Specifies the location where the UDF will find the policy. Must be the same as configured in the *pepserver.cfg* file.
dataelement	VARCHAR(64)	Specifies the name of the data element.
input_data	DOUBLE	Specifies the input data for the UDF.
scid	INTEGER	Specifies the Security Coordinate ID. It is not used and should be set to zero.

Returns

Encrypted data as VARCHAR(34) FOR BIT DATA.

NULL: When user has no access to database.

Exception

If set in policy and user does not have access, then the UDF terminates.

Example

```
SELECT pty.ins_enc_double(0, 'AES128', CAST(123456 AS DOUBLE), 10) FROM
SYSIBM.SYSDUMMY1;
```

4.1.16.2 pty.upd_enc_double

This re-encrypts the DOUBLE data that has been updated with a data element for encryption.

pty.upd_enc_double(comm_id INTEGER, dataelement VARCHAR,

Parameters

Name	Type	Description
comm_id	INTEGER	Specifies the location where the UDF will find the policy. Must be the same as configured in the *pepserver.cfg* file.
dataelement	VARCHAR(64)	Specifies the name of the data element.
input_data	DOUBLE	Specifies the input data for the UDF.
scid	INTEGER	Specifies the Security Coordinate ID. It is not used and should be set to zero.

Returns

Encrypted data as VARCHAR(34) FOR BIT DATA.

NULL: When user has no access to database.

Exception

If set in policy and user does not have access, then the UDF terminates.

Example

```
SELECT pty.upd_enc_double(0, 'AES128', CAST(123456 AS DOUBLE), 10) FROM
SYSIBM.SYSDUMMY1;
```

4.1.16.3 pty.sel_dec_double

This function decrypts the DOUBLE data with a data element.

pty.sel_dec_double(comm_id INTEGER, dataelement VARCHAR, input_data VARCHAR, scid INTEGER)

Parameters

Name	Type	Description
comm_id	INTEGER	Specifies the location where the UDF will find the policy. Must be the same as configured in the *pepserver.cfg* file.
dataelement	VARCHAR(64)	Specifies the name of the data element.
input_data	VARCHAR(34) FOR BIT DATA	Specifies the input data for the UDF.
scid	INTEGER	Specifies the Security Coordinate ID. It is not used and should be set to zero.

Returns

Decrypted data as DOUBLE.

NULL: When user has no access to database.

Exception

If set in policy and user does not have access, then the UDF terminates.

Example

```
SELECT pty.sel_dec_double(0, 'AES128', pty.ins_enc_double(0, 'AES128', CAST(123456 AS
DOUBLE), 10), 10) FROM SYSIBM.SYSDUMMY1;
```

4.1.16.4 pty.ins_double

This function protects the DOUBLE data with the No Encryption data element.

pty.ins_double(comm_id INTEGER, dataelement VARCHAR, input_data DOUBLE, scid INTEGER)

Parameters

Name	Type	Description
comm_id	INTEGER	Specifies the location where the UDF will find the policy. Must be the same as configured in the *pepserver.cfg* file.
dataelement	VARCHAR(64)	Specifies the name of the data element.
input_data	DOUBLE	Specifies the input data for the UDF.
scid	INTEGER	Specifies the Security Coordinate ID. It is not used and should be set to zero.

Returns

Data as DOUBLE.

NULL: When user has no access to database.

Exception

If set in policy and user does not have access, then the UDF terminates.

Example

```
SELECT pty.ins_double(0, 'NoEncryption', CAST(123456 AS DOUBLE), 10) FROM
SYSIBM.SYSDUMMY1;
```

4.1.16.5 pty.upd_double

This function re-protects the DOUBLE data with the No Encryption data element.

pty.upd_double(comm_id INTEGER, dataelement VARCHAR, input_data DOUBLE, scid INTEGER)

Parameters

Name	Type	Description
comm_id	INTEGER	Specifies the location where the UDF will find the policy. Must be the same as configured in the *pepserver.cfg* file.
dataelement	VARCHAR(64)	Specifies the name of the data element.

Name	Type	Description
input_data	DOUBLE	Specifies the input data for the UDF.
scid	INTEGER	Specifies the Security Coordinate ID. It is not used and should be set to zero.

Returns

Data as DOUBLE.

NULL: When user has no access to database.

Exception

If set in policy and user does not have access, then the UDF terminates.

Example

```
SELECT pty.upd_double(0, 'NoEncryption', CAST(123456 AS DOUBLE), 10) FROM
SYSIBM.SYSDUMMY1;
```

4.1.16.6 pty.sel_double

This function unprotects the DOUBLE data with the No Encryption data element.

pty.sel_double(comm_id INTEGER, dataelement VARCHAR, input_data DOUBLE, scid INTEGER)

Parameters

Name	Type	Description
comm_id	INTEGER	Specifies the location where the UDF will find the policy. Must be the same as configured in *pepserver.cfg*
dataelement	VARCHAR(64)	Specifies the name of the data element.
input_data	DOUBLE	Specifies the input data for the UDF.
scid	INTEGER	Specifies the Security Coordinate ID. It is not used and should be set to zero.

Returns

Data as DOUBLE.

NULL: When user has no access to database.

Exception

If set in policy and user does not have access, then the UDF terminates.

Example

```
SELECT pty.sel_double(0, 'NoEncryption', pty.ins_double(0, 'NoEncryption', CAST(123456
AS DOUBLE), 10), 10) FROM SYSIBM.SYSDUMMY1;
```

4.1.17 BLOB UDFs

These UDFs are used to encrypt and decrypt BLOB data.

Note:

- The BLOB UDFs supports a maximum input data of 64 KB. If the input data exceeds the maximum limit, then, an error message "Integrity check failed" appears.
- The BLOB UDFs are incompatible with videographic file formats.

4.1.17.1 pty.ins_enc_blob

This function encrypts the BLOB value with a data element for encryption.

pty.ins_enc_blob(comm_id INTEGER, dataelement VARCHAR, input_data BLOB, scid INTEGER)

Parameters

Name	Type	Description
comm_id	INTEGER	Specifies the location where the UDF will find the policy. Must be the same as configured in the *pepserver.cfg* file.
dataelement	VARCHAR(64)	Specifies the name of the data element.
input_data	BLOB(100 KB)	Specifies the input data for the UDF.
scid	INTEGER	Specifies the Security Coordinate ID. It is not used and should be set to zero.

Returns

Encrypted data as BLOB(102434 bytes).

NULL: When user has no access to database.

Exception

If set in policy and user does not have access, then the UDF terminates.

Example

```
SELECT pty.ins_enc_blob(0, 'AES128', BLOB('blob_db2.jpg'), 10) FROM SYSIBM.SYSDUMMY1;
```

4.1.17.2 pty.upd_enc_blob

This re-encrypts the BLOB value that has been updated with a data element for encryption.

pty.upd_enc_blob(comm_id INTEGER, dataelement VARCHAR, input_data BLOB, scid INTEGER)

Parameters

Name	Type	Description
comm_id	INTEGER	Specifies the location where the UDF will find the policy. Must be the same as configured in the *pepserver.cfg* file.
dataelement	VARCHAR(64)	Specifies the name of the data element.
input_data	BLOB(100 KB)	Specifies the input data for the UDF.
scid	INTEGER	Specifies the Security Coordinate ID. It is not used and should be set to zero.

Returns

Encrypted data as BLOB(102434 bytes).

NULL: When user has no access to database.

Exception

If set in policy and user does not have access, then the UDF terminates.

Example

```
SELECT pty.upd_enc_blob(0, 'AES128', BLOB('blob_db2.jpg'), 10) FROM SYSIBM.SYSDUMMY1;
```

4.1.17.3 pty.sel_dec_blob

This function decrypts the BLOB value with a data element.

pty.sel_dec_blob(comm_id INTEGER, dataelement VARCHAR, input_data BLOB, scid INTEGER)

Parameters

Name	Type	Description
comm_id	INTEGER	Specifies the location where the UDF will find the policy. Must be the same as configured in the *pepserver.cfg* file.
dataelement	VARCHAR(64)	Specifies the name of the data element.
input_data	BLOB(102434 bytes)	Specifies the input data for the UDF.
scid	INTEGER	Specifies the Security Coordinate ID. It is not used and should be set to zero.

Returns

Decrypted data as BLOB(100 KB).

NULL: When user has no access to database.

Exception

If set in policy and user does not have access, then the UDF terminates.

Example

```
SELECT pty.sel_dec_blob(0, 'AES128', pty.ins_enc_blob(0, 'AES128',
BLOB('blob_db2.jpg'), 10), 10) FROM SYSIBM.SYSDUMMY1;
```

4.1.17.4 pty.ins_blob

This function protects the BLOB value with the No Encryption data element.

pty.ins_blob(comm_id INTEGER, dataelement VARCHAR, input_data BLOB, scid INTEGER)

Parameters

Name	Type	Description
comm_id	INTEGER	Specifies the location where the UDF will find the policy. Must be the same as configured in the *pepserver.cfg* file.
dataelement	VARCHAR(64)	Specifies the name of the data element.
input_data	BLOB(100 KB)	Specifies the input data for the UDF.
scid	INTEGER	Specifies the Security Coordinate ID. It is not used and should be set to zero.

Returns

Data as BLOB(100 KB).

NULL: When user has no access to database.

Exception

If set in policy and user does not have access, then the UDF terminates.

Example

```
SELECT pty.ins_blob(0, 'NoEncryption', BLOB('blob_db2.jpg'), 10) FROM SYSIBM.SYSDUMMY1;
```

4.1.17.5 pty.upd_blob

This function re-protects the BLOB value with the No Encryption data element.

pty.upd_blob(comm_id INTEGER, dataelement VARCHAR, input_data BLOB, scid INTEGER)

Parameters

Name	Type	Description
comm_id	INTEGER	Specifies the location where the UDF will find the policy. Must be the same as configured in the *pepserver.cfg* file.
dataelement	VARCHAR(64)	Specifies the name of the data element.
input_data	BLOB(100 KB)	Specifies the input data for the UDF.
scid	INTEGER	Specifies the Security Coordinate ID. It is not used and should be set to zero.

Returns

Data as BLOB(100 KB).

NULL: When user has no access to database.

Exception

If set in policy and user does not have access, then the UDF terminates.

Example

```
SELECT pty.upd_blob(0, 'NoEncryption', BLOB('blob_db2.jpg'), 10) FROM SYSIBM.SYSDUMMY1;
```

4.1.17.6 pty.sel_blob

This function unprotects the BLOB value with the No Encryption data element.

pty.sel_blob(comm_id INTEGER, dataelement VARCHAR, input_data BLOB, scid INTEGER)

Parameters

Name	Type	Description
comm_id	INTEGER	Specifies the location where the UDF will find the policy. Must be the same as configured in the *pepserver.cfg* file.
dataelement	VARCHAR(64)	Specifies the name of the data element.
input_data	BLOB(100 KB)	Specifies the input data for the UDF.
scid	INTEGER	Specifies the Security Coordinate ID. It is not used and should be set to zero.

Returns

Data as BLOB(100 KB).

NULL: When user has no access to database.

Exception

If set in policy and user does not have access, then the UDF terminates.

Example

```
SELECT pty.sel_blob(0, 'NoEncryption', pty.ins_blob(0, 'NoEncryption',
BLOB('blob_db2.jpg'), 10), 10 ) FROM SYSIBM.SYSDUMMY1;
```

4.1.18 CLOB UDFs

These UDFs are used to encrypt and decrypt CLOB data.

> **Note:** The CLOB UDFs supports a maximum input data of 5 KB.

4.1.18.1 pty.ins_enc_clob

This function encrypts the CLOB value with a data element for encryption.

pty.ins_enc_clob(comm_id INTEGER, dataelement VARCHAR, input_data CLOB, scid INTEGER)

Parameters

Name	Type	Description
comm_id	INTEGER	Specifies the location where the UDF will find the policy. Must be the same as configured in the *pepserver.cfg* file.
dataelement	VARCHAR(64)	Specifies the name of the data element.
input_data	CLOB(5 KB)	Specifies the input data for the UDF.
scid	INTEGER	Specifies the Security Coordinate ID. It is not used and should be set to zero.

Returns

Encrypted data as BLOB(5154 bytes).

NULL: When user has no access to database.

Exception

If set in policy and user does not have access, then the UDF terminates.

Example

```
SELECT pty.ins_enc_clob(0, 'AES128', CLOB('clob.txt'), 10) FROM SYSIBM.SYSDUMMY1;
```

4.1.18.2 pty.upd_enc_clob

This re-encrypts the CLOB value that has been updated with a data element for encryption.

pty.upd_enc_clob(comm_id INTEGER, dataelement VARCHAR, input_data CLOB, scid INTEGER)

Parameters

Name	Type	Description
comm_id	INTEGER	Specifies the location where the UDF will find the policy. Must be the same as configured in the *pepserver.cfg* file.
dataelement	VARCHAR(64)	Specifies the name of the data element.
input_data	CLOB(5 KB)	Specifies the input data for the UDF.
scid	INTEGER	Specifies the Security Coordinate ID. It is not used and should be set to zero.

Returns

Encrypted data as BLOB(5154 bytes).

NULL: When user has no access to database.

Exception

If set in policy and user does not have access, then the UDF terminates.

Example

```
SELECT pty.upd_enc_clob(0, 'AES128', CLOB('clob.txt'), 10) FROM SYSIBM.SYSDUMMY1;
```

4.1.18.3 pty.sel_dec_clob

This function decrypts the CLOB value with a data element.

pty.sel_dec_clob(comm_id INTEGER, dataelement VARCHAR, input_data CLOB, scid INTEGER)

Parameters

Name	Type	Description
comm_id	INTEGER	Specifies the location where the UDF will find the policy. Must be the same as configured in the *pepserver.cfg* file.
dataelement	VARCHAR(64)	Specifies the name of the data element.
input_data	BLOB(5154 bytes)	Specifies the input data for the UDF.
scid	INTEGER	Specifies the Security Coordinate ID. It is not used and should be set to zero.

Returns

Decrypted data as CLOB(5 KB).

NULL: When user has no access to database.

Exception

If set in policy and user does not have access, then the UDF terminates.

Example

```
SELECT pty.sel_dec_clob(0, 'AES128', pty.ins_enc_clob(0, 'AES128', CLOB('clob.txt'),
10), 10) FROM SYSIBM.SYSDUMMY1;
```

4.1.18.4 pty.ins_clob

This function protects the CLOB value with the No Encryption data element.

pty.ins_clob(comm_id INTEGER, dataelement VARCHAR, input_data CLOB, scid INTEGER)

Parameters

Name	Type	Description
comm_id	INTEGER	Specifies the location where the UDF will find the policy. Must be the same as configured in the *pepserver.cfg* file.
dataelement	VARCHAR(64)	Specifies the name of the data element.
input_data	CLOB(5 KB)	Specifies the input data for the UDF.
scid	INTEGER	Specifies the Security Coordinate ID. It is not used and should be set to zero.

Returns

Clear data as CLOB(5376 bytes).

NULL: When user has no access to database.

Exception

If set in policy and user does not have access, then the UDF terminates.

Example

```
SELECT pty.ins_clob(0, 'NoEncryption', CLOB('clob.txt'), 10) FROM SYSIBM.SYSDUMMY1;
```

4.1.18.5 pty.upd_clob

This function re-protects the CLOB value with the No Encryption data element.

pty.upd_clob(comm_id INTEGER, dataelement VARCHAR, input_data CLOB, scid INTEGER)

Parameters

Name	Type	Description
comm_id	INTEGER	Specifies the location where the UDF will find the policy. Must be the same as configured in the *pepserver.cfg* file.
dataelement	VARCHAR(64)	Specifies the name of the data element.
input_data	CLOB(5 KB)	Specifies the input data for the UDF.
scid	INTEGER	Specifies the Security Coordinate ID. It is not used and should be set to zero.

Returns

Clear data as CLOB(5376 bytes).

NULL: When user has no access to database.

Exception

If set in policy and user does not have access, then the UDF terminates.

Example

```
SELECT pty.upd_clob(0, 'NoEncryption', CLOB('clob.txt'), 10) FROM SYSIBM.SYSDUMMY1;
```

4.1.18.6 pty.sel_clob

This function unprotects the CLOB value with the No Encryption data element.

pty.sel_clob(comm_id INTEGER, dataelement VARCHAR, input_data CLOB, scid INTEGER)

Parameters

Name	Type	Description
comm_id	INTEGER	Specifies the location where the UDF will find the policy. Must be the same as configured in the *pepserver.cfg* file.
dataelement	VARCHAR(64)	Specifies the name of the data element.

Name	Type	Description
input_data	CLOB(5376 bytes)	Specifies the input data for the UDF.
scid	INTEGER	Specifies the Security Coordinate ID. It is not used and should be set to zero.

Returns

Clear data as CLOB(5 KB).

NULL: When user has no access to database.

Exception

If set in policy and user does not have access, then the UDF terminates.

Example

```
SELECT pty.sel_clob(0, 'NoEncryption', pty.ins_clob(0, 'NoEncryption',
CLOB('clob.txt'), 10), 10) FROM SYSIBM.SYSDUMMY1;
```

4.2 Greenplum DB Protector UDFs

This section provides a detailed list of UDFs (User Defined Functions) for general information, and protection and unprotection of different data types.

> **Note:** In Greenplum, data of printable data type displays a blank or binary characters when detokenized. To view the correct value, convert the detokenized value to hex and compare with hex value of the clear data before checking the value in the original printable data type.

The following table provides the list of parameters that have been used in the examples provided with the UDFs.

Parameter Name	Description
TESTDB	Name of database
USER1	Database user in database and policy
Data elements in policy	
AES128	Encryption with AES128
AES128_IV_CRC_KID	Encryption with AES128, IV, CRC, and KID
NoEncryption	No encryption
TE_A_N_S23_L2R2_Y	Tokenization with Alphanumeric
TE_INT_4	Tokenization with Integer (4 byte), SLT 1-3
TE_Date_YMD_S26	Tokenization with Date YMD format, SLT 2-6
HMAC_SHA1	Hashing

4.2.1 General UDFs

4.2.1.1 pty_whoami

This UDF returns the name of the user currently logged in.

pty_whoami()

Parameters

None

Returns

Name of user logged on to the database as STRING

Exception

If set in policy and user does not have access, then the UDF terminates

Example

```
select pty_whoami();
```

4.2.1.2 pty_getversion

This UDF returns the version of the product.

pty_getversion()

Parameters

None

Returns

Product version as a string

Exception

If set in policy and user does not have access, then the UDF terminates

Example

```
select pty_getversion();
```

4.2.1.3 pty_getcurrentkeyid

This UDF returns the current key ID for a data element. It is typically used together with **pty_getkeyid** to determine if some data is protected with the most recent key for a given data element. User must have access rights for protection to run this UDF successfully.

pty_getcurrentkeyid(dataelement VARCHAR)

Parameters

Name	Type	Description
dataelement	VARCHAR	Name of data element. The example is for AES128 encryption with

Returns

Current key ID as INTEGER

Exception

If set in policy and user does not have access, then the UDF terminates

Example

```
select pty_getcurrentkeyid('AES128_IV_CRC_KID');
```

4.2.1.4 pty_getkeyid

This UDF returns the key ID that was used to protect an item of data. It is typically used together with **pty_getcurrentkeyid** to determine if some data is protected with the most recent key for a given data element.

You should have access rights to protect data for using this UDF.

pty_getkeyid(dataelement VARCHAR, data VARCHAR)

Parameters

Name	Type	Description
dataelement	VARCHAR	Name of data element
data	VARCHAR	Data that has been protected with encryption and is using key ID

Returns

Key ID as INTEGER

Exception

If set in policy and user does not have access, then the UDF terminates

Example

```
select pty_getkeyid('AES128_IV_CRC_KID', pty_varcharenc('ProtegrityProtegrity',
'AES128_IV_CRC_KID') );
```

4.2.2 VARCHAR UDFs

4.2.2.1 pty_varcharenc

This UDF protects VARCHAR data using encryption data element.

pty_varcharenc(data)

Parameters

Name	Type	Description
data	VARCHAR	Data to protect
dataelement	VARCHAR	Name of data element

Returns

Protected value as BYTEA

Exception

If set in policy and user does not have access, then the UDF terminates

Example

```
select pty_varcharenc('ProtegrityProtegrity', 'AES128_IV_CRC_KID');
```

4.2.2.2 pty_varchardec

This UDF decrypts the encrypted data that was encrypted using the *pty_varcharenc* UDF.

pty_varchardec(data BYTEA, dataelement VARCHAR)

Parameters

Name	Type	Description
data	BYTEA	Data to unprotect
dataelement	VARCHAR	Name of data element

Returns

Clear or unprotected VARCHAR value, if user has access to data

NULL, if user does not have access to data

Exception

If set in policy and user does not have access, then the UDF terminates

Example

```
select pty_varchardec(pty_varcharenc('ProtegrityProtegrity', 'AES128'), 'AES128') );
```

4.2.2.3 pty_varcharins

This UDF protects VARCHAR data with tokenization or no-encryption.

pty_varcharins(data VARCHAR, dataelement VARCHAR)

Parameters

Name	Type	Description
data	VARCHAR	Data to protect
dataelement	VARCHAR	Name of data element

Returns

Protected VARCHAR value, if user has access to data

Exception

If set in policy and user does not have access, then the UDF terminates

Example

```
select pty_varcharins('ProtegrityProtegrity', 'TE_A_N_S23_L2R2_Y');
```

4.2.2.4 pty_varcharsel

This UDF unprotects the protected data.

pty_varcharsel(data VARCHAR, dataelement VARCHAR)

Parameters

Name	Type	Description
data	VARCHAR	Data to unprotect
dataelement	VARCHAR	Name of data element

Returns

Clear or unprotected VARCHAR value, if user has access to data

NULL, if user does not have access to data

Exception

If set in policy and user does not have access, then the UDF terminates

Example

```
select pty_varcharsel(pty_varcharins('ProtegrityProtegrity', 'TE_A_N_S23_L2R2_Y'),
'TE_A_N_S23_L2R2_Y');
```

4.2.2.5 pty_varcharhash

This UDF hashes and protects the VARCHAR value. This is a one-way function and the protected data cannot be unprotected.

pty_varcharhash(col VARCHAR, dataelement VARCHAR)

Parameters

Name	Type	Description
col	VARCHAR	Data to protect
dataelement	VARCHAR	Name of data element

Returns

Hash value as BYTEA.

Exception

If set in policy and user does not have access, then the UDF terminates

Example

```
select pty_varcharhash('ProtegrityProtegrity', 'HMAC_SHA1');
```

4.2.2.6 pty_fpeunicodevarcharins

This UDF protects data with a data element for Format Preserving Encryption (FPE) with any plaintext encoding type.

pty_fpeunicodevarcharins(col VARCHAR, dataelement VARCHAR)

Parameters

Name	Type	Description
col	VARCHAR	Data to protect
dataelement	VARCHAR	Name of data element

Returns

Protected VARCHAR value, if user has access to data.

The maximum input length supported is 2752 bytes. If the length of the protected value exceeds the maximum output buffer limit, which is 5504 bytes, then the following error `Output buffer is too small` is returned.

Exception

If set in policy and user does not have access, then the UDF terminates.

> **Note:**
> Ensure that you use the supported data element only. Using an unsupported data element might result in successful protection without returning any error, but corruption of data can occur.

Example

```
select pty_fpeunicodevarcharins('Protegrity','FPE_UNICODE_BAS_LAT_LAT1_SUPP_A_UTF8');
```

4.2.2.7 pty_fpeunicodevarcharsel

This UDF unprotects the protected data.

pty_fpeunicodevarcharsel(col VARCHAR, dataelement VARCHAR)

Parameters

Name	Type	Description
col	VARCHAR	Data to unprotect
dataelement	VARCHAR	Name of data element

Returns

Clear or unprotected VARCHAR value, if user has access to data.

NULL, if user does not have access to data.

Exception

If set in policy and user does not have access, then the UDF terminates

> **Note:**
> Ensure that you use the supported data element only. Using an unsupported data element might result in successful protection without returning any error, but corruption of data can occur.

.

Example

```
select
pty_fpeunicodevarcharsel(pty_fpeunicodevarcharins('Protegrity','FPE_UNICODE_BAS_LAT_LAT1
_SUPP_A_UTF8'),'FPE_UNICODE_BAS_LAT_LAT1_SUPP_A_UTF8');
```

4.2.3 INTEGER UDFs

4.2.3.1 pty_integerenc

This UDF protects INTEGER data using encryption data element.

pty_integerenc(data INTEGER, dataelement VARCHAR)

Parameters

Name	Type	Description
data	INTEGER	Data to protect
dataelement	VARCHAR	Name of data element

Returns

Protected value as BYTEA

Exception

If set in policy and user does not have access, then the UDF terminates

Example

```
select pty_integerenc(123456, ‘AES128’), ‘AES128’);
```

4.2.3.2 pty_integerdec

This UDF unprotects the protected data.

pty_integerdec(data BYTEA, dataelement VARCHAR)

Parameters

Name	Type	Description
data	BYTEA	Data to unprotect
dataelement	VARCHAR	Name of data element

Returns

Clear or unprotected INTEGER value, if user has access to data

NULL, if user does not have access to data

Exception

If set in policy and user does not have access, then the UDF terminates

Example

```
select pty_integerdec(pty_integerenc(123456, ‘AES128’), ‘AES128’);
```

4.2.3.3 pty_integerins

This UDF protects INTEGER data with tokenization, type preserving encryption, or no-encryption.

pty_integerins(data INTEGER, dataelement VARCHAR))

Parameters

Name	Type	Description
data	INTEGER	Data to protect
dataelement	VARCHAR	Name of data element

Returns

Protected value as INTEGER

Exception

If set in policy and user does not have access, then the UDF terminates

Example

```
select pty_integerins(123456, 'TE_INT_4');
```

4.2.3.4 pty_integersel

This UDF unprotects the protected data.

pty_integersel(data INTEGER, dataelement VARCHAR)

Parameters

Name	Type	Description
data	INTEGER	Data to unprotect
dataelement	VARCHAR	Name of data element

Returns

Clear or unprotected INTEGER value, if user has access to data

NULL, if user does not have access to data

Exception

If set in policy and user does not have access, then the UDF terminates

Example

```
select pty_integersel(pty_integerins(123456, 'TE_INT_4'), 'TE_INT_4');
```

4.2.3.5 pty_integerhash

This UDF calculates a hash value for an INTEGER. This is a one-way function and data cannot be unprotected.

pty_integerhash(col INTEGER, dataelement VARCHAR)

Parameters

Name	Type	Description
col	INTEGER	Data to protect
dataelement	VARCHAR	Name of data element

Returns

Hash value as BYTEA.

Exception

If set in policy and user does not have access, then the UDF terminates

Example

```
select pty_integerhash(123456, 'HMAC_SHA1');
```

4.2.4 DATE UDFs

4.2.4.1 pty_dateenc

This UDF protects DATE data with strong encryption.

pty_dateenc(data DATE, dataelement VARCHAR)

Parameters

Name	Type	Description
data	DATE	Data to protect
dataelement	VARCHAR	Name of data element

Returns

Protected value as BYTEA

Exception

If set in policy and user does not have access, then the UDF terminates

Example

```
select pty_dateenc(to_date('22-09-1990', 'DD-MM-YYYY'), 'AES128');
```

4.2.4.2 pty_datedec

This UDF unprotects protected data.

pty_datedec(data BYTEA, dataelement VARCHAR)

Parameters

Name	Type	Description
data	BYTEA	Data to unprotect
dataelement	VARCHAR	Name of data element

Returns

Clear or unprotected DATE value, if user has access to data

NULL, if user does not have access to data

Exception

If set in policy and user does not have access, then the UDF terminates

Example

```
select pty_datedec(pty_dateenc(to_date('22-09-1990', 'DD-MM-YYYY'), 'AES128'),
'AES128');
```

4.2.4.3 pty_dateins

This UDF protects DATE data with tokenization, type preserving encryption or no-encryption.

pty_dateins(data DATE, dataelement VARCHAR)

Parameters

Name	Type	Description
data	DATE	Data to protect
dataelement	VARCHAR	Name of data element

Returns

Protected value as DATE

Exception

If set in policy and user does not have access, then the UDF terminates

Example

```
select pty_dateins(to_date('22-09-1990', 'DD-MM-YYYY'), 'TE_Date_YMD_S26’),
'TE_Date_YMD_S26');
```

4.2.4.4 pty_datesel

This UDF unprotects protected data.

pty_datesel(data DATE, dataelement VARCHAR)

Parameters

Name	Type	Description
data	DATE	Data to protect
dataelement	VARCHAR	Name of data element

Returns

Clear or unprotected DATE value, if user has access to data

NULL, if user does not have access to data

Exception

If set in policy and user does not have access, then the UDF terminates

Example

```
select pty_datesel(pty_dateins(to_date('22-09-1990', 'DD-MM-YYYY'), 'TE_Date_YMD_S26’),
'TE_Date_YMD_S26');
```

4.2.4.5 pty_datehash

This UDF calculates the hash value for DATE. This is a one-way function and data cannot be unprotected.

pty_datehash(col DATE, dataelement VARCHAR)

Parameters

Name	Type	Description
col	DATE	Data to protect

Name	Type	Description
dataelement	VARCHAR	Name of data element

Returns

Hash value as BYTEA.

Exception

If set in policy and user does not have access, then the UDF terminates

Example

```
select pty_datehash(to_date('11-11-1990', 'DD-MM-YYYY'), 'HMAC_SHA1');
```

4.2.5 REAL UDFs

4.2.5.1 pty_realenc

This UDF protects REAL data with strong encryption.

pty_realenc(data REAL, dataelement VARCHAR)

Parameters

Name	Type	Description
data	REAL	Data to protect
dataelement	VARCHAR	Name of data element

Returns

Protected value as BYTEA

Exception

If set in policy and user does not have access, then the UDF terminates

Example

```
select pty_realenc(‘12345e+12’, 'AES128');
```

4.2.5.2 pty_realdec

This UDF unprotects protected data.

pty_realdec(data BYTEA, dataelement VARCHAR)

Parameters

Name	Type	Description
data	BYTEA	Data to unprotect
dataelement	VARCHAR	Name of data element

Returns

Clear or unprotected REAL value, if user has access to data

NULL, if user does not have access to data

Exception

If set in policy and user does not have access, then the UDF terminates

Example

```
select pty_realdec(pty_realenc('12345e+12', 'AES128'), 'AES128');
```

4.2.5.3 pty_realins

This function is used for no encryption with a data element.

pty_realins(data REAL, dataelement VARCHAR)

Parameters

Name	Type	Description
data	REAL	Data to unprotect
dataelement	VARCHAR	Name of data element

Returns

Protected value as REAL

Exception

If set in policy and user does not have access, then the UDF terminates

Example

```
select pty_realins('12345e+12', 'NoEncryption');
```

4.2.5.4 pty_realsel

This UDF unprotects protected data.

pty_realsel(data REAL, dataelement VARCHAR)

Parameters

Name	Type	Description
data	REAL	Data to unprotect
dataelement	VARCHAR	Name of data element

Returns

Clear or unprotected REAL value, if user has access to data

NULL, if user does not have access to data

Exception

If set in policy and user does not have access, then the UDF terminates

Example

```
select pty_realsel(pty_realins('12345e+12', 'NoEncryption'), 'NoEncryption');
```

4.2.5.5 pty_realhash

This UDF calculates hash value for input REAL data. This is a one-way function and data cannot be unprotected.

pty_realhash(col REAL, dataelement VARCHAR)

Parameters

Name	Type	Description
col	REAL	Data to unprotect
dataelement	VARCHAR	Name of data element

Returns

Hash value as BYTEA.

Exception

If set in policy and user does not have access, then the UDF terminates

Example

```
select pty_realhash('12345e+12', 'HMAC_SHA1');
```

4.3 MS SQL DB Protector Functions

This section provides a detailed list of functions and extended stored procedures for general functions, and protection and unprotection of different data types.

> **Note:**
> In the case of MS SQL DB Protector, if the data element which is greater than 55 characters long is passed to the UDF, then the UDF terminates with the following error message.
>
> *No data element specified*

4.3.1 GENERAL Functions

4.3.1.1 pty_getVersion

This function returns the version of the installed UDFs.

pty_getVersion()

Parameters

None

Returns

Version number as NVARCHAR.

Example

```
DECLARE
   @data NVARCHAR(64)
SELECT @data = master.dbo.pty_getVersion()
PRINT @data;
```

4.3.1.2 pty_whoAmI

This function returns the name of the user currently logged in and running this function.

pty_whoAmI()

Parameters

None

Returns

Version number as NVARCHAR.

Example

```
DECLARE
   @data VARCHAR(128)
SELECT @data = master.dbo.pty_whoAmI()
PRINT @data
```

4.3.2 ACCESS CHECK Procedures

These functions check access permissions allowed to the user for protecting or unprotecting the data. Depending on availability of access, a bit value 0 (zero) or 1 is returned.

4.3.2.1 xp_pty_select_check

This function determines if the user has Select access to the data element.

xp_pty_select_check(dataelement VARCHAR)

Parameters

Name	Type	Description
dataelement	VARCHAR(64)	Name of data element. In the Example, value='AES256'.

Returns

- 0 – if user has Select access
- 1 – if user does not have the access

Example

```
DECLARE
   @result BIT
SELECT @result = master.dbo.xp_pty_select_check ('AES256')
PRINT @result
```

4.3.2.2 xp_pty_update_check

This function determines if the user has Update access to the data element.

xp_pty_update_check (dataelement VARCHAR)

Parameters

Name	Type	Description
dataelement	VARCHAR(64)	Name of data element.

Name	Type	Description
		In the **Example**, value='AES256'.

Returns

- 0 – if user has Update access
- 1 – if user does not have the access

Example

```
DECLARE
   @result BIT
SELECT @result = master.dbo.xp_pty_update_check('AES256')
PRINT @result
```

4.3.2.3 xp_pty_insert_check

This function checks if the user has Insert access to the data element.

xp_pty_insert_check(datalement VARCHAR)

Parameters

Name	*Type*	*Description*
dataelement	VARCHAR(64)	Name of data element. In the **Example**, value='AES256'.

Returns

- 0 – if user has Insert access
- 1 – if user does not have the access

Example

```
DECLARE
   @result BIT
SELECT @result = master.dbo.xp_pty_insert_check('AES256')
PRINT @result
```

4.3.2.4 xp_pty_delete_check

This function determines if the user has Delete access to the data element.

xp_pty_delete_check(dataelement VARCHAR, scid INT)

Parameters

Name	*Type*	*Description*
dataelement	VARCHAR(64)	Name of data element. In the Example, value='AES256'.
scid	INT	Security Coordinate ID. It is not used and should be set to zero.

Returns

- 0 – if user has Delete access
- 1 – if user does not have the access

Example

```
DECLARE
   @result BIT
SELECT @result = master.dbo.xp_pty_delete_check('AES256', 0)
PRINT @result
```

4.3.3 SELECT Functions and Procedures

These functions and stored procedures unprotect data and return the unprotected value.

4.3.3.1 pty_select

This function unprotects data that is protected by an encrypting data element.

pty_select (data VARBINARY, dataelement VARCHAR, def VARCHAR, scid INT)

Parameters

Name	*Type*	*Description*
data	VARBINARY(8000)	Data to unprotect
dataelement	VARCHAR(64)	Name of data element. In the Example, value='AES256'.
def	VARCHAR(8000)	Default value returned if user does not have permission to unprotect.
scid	INT	Security Coordinate ID. It is not used and should be set to zero.

Returns

Unprotected value as VARCHAR.

NULL: When user has no access to the data in the policy.

Exception

If configured in policy and user does not have access, the function terminates with an error message explaining what went wrong.

> **Note:**
> Ensure that you use the supported data element only. Using an unsupported data element might result in successful unprotection without returning any error, but corruption of data can occur.

Example

```
DECLARE
   @result INT,
   @encrypt VARBINARY64),
   @data VARCHAR64)
SET @data='232432423432'
EXEC @result= master.dbo.xp_pty_insert @encrypt output,@data, 'AES256',0
IF @result=0
   PRINT 'OK'
```

```
ELSE
   PRINT 'ERROR'
PRINT @encrypt
SELECT @data = master.dbo.pty_select (@encrypt, 'AES256',null,0)
PRINT @data
```

4.3.3.2 pty_selectunicode

This function unprotects data that is protected by Unicode Base64, Unicode Gen2, and FPE Unicode data elements.

> **Note:** This UDF does not support masking.

pty_selectunicode (data NVARCHAR, dataelement VARCHAR, def INT, scid INT)

Parameters

Name	*Type*	*Description*
Data	NVARCHAR(4000)	Data to unprotect
dataelement	VARCHAR(64)	Name of the data element
def	INT	The default value to return in case the user does not have permission to unprotect.
Scid	INT	Security Coordinate ID. It is not used and should be set to zero.

Returns

Unprotected value as NVARCHAR.

NULL: When user has no access to the data in the policy.

Exception

If configured in policy and user does not have access, the function terminates with an error message that explains what went wrong.

Example for Unicode Base64

```
DECLARE
   @result INT,
   @encrypt NVARCHAR(4000),
   @data NVARCHAR(4000)
SET @data= 'Protegrityl23'
EXEC @result= master.dbo.xp_pty_tpe_unicode_insert @encrypt output,@data,
'TE_UNICODE_BASE64_SLT23_ASTYES',0
IF @result=0
   PRINT 'OK'
ELSE
   PRINT 'ERROR'
PRINT @encrypt
SELECT @data = master.dbo.pty_selectunicode (@encrypt,
'TE_UNICODE_BASE64_SLT23_ASTYES',null,0)
PRINT @data
```

Example for Unicode Gen2

```
DECLARE
   @result INT,
   @encrypt NVARCHAR(4000),
   @data NVARCHAR(4000)
SET @data= N'Â¢Â€ÃÃ,ÃƒÃ„Ã…ÃŠÃ‹ÃŒÃÃŽÃÃÃ‘Ã’Ã“ÃÃ'
```

```
EXEC @result= master.dbo.xp_pty_tpe_unicode_insert @encrypt output,@data,
'TE_UG2_SLT13_L2R2_UTF16LE_Latin1_Supplement_ASTYES',0
IF @result=0
   PRINT 'OK'
ELSE
   PRINT 'ERROR'
PRINT @encrypt
SELECT @data = master.dbo.pty_selectunicode (@encrypt,
'TE_UG2_SLT13_L2R2_UTF16LE_Latin1_Supplement_ASTYES',null,0)
PRINT @data
```

Example for FPE Unicode

```
DECLARE
   @result INT,
   @encrypt NVARCHAR(4000),
   @data NVARCHAR(4000)
SET @data= N'232432423432'
EXEC @result= master.dbo.xp_pty_tpe_unicode_insert @encrypt output,@data,
'fpe_unicode',0
IF @result=0
   PRINT 'OK'
ELSE
   PRINT 'ERROR'
PRINT @encrypt
SELECT @data = master.dbo.pty_selectunicode (@encrypt, 'fpe_unicode',null,0)
PRINT @data
```

4.3.3.3 pty_select2

This function is used to unprotect data protected by a type preserving data element such as Tokens, DTP2 and No Encryption for access control.

> **Note:**
> Starting from the Version 7.1, Maintenance Release 1 (MR1), the DTP2 protection method is deprecated.
>
> For assistance in switching to a different protection method, contact Protegrity.

pty_select2(data VARCHAR, dataelement VARCHAR, def VARCHAR, scid INT))

Parameters

Name	*Type*	*Description*
data	VARCHAR(8000)	Data to unprotect.
dataelement	VARCHAR(64)	Name of data element. In the Example, value=‘TE_N_S16_L0R0_Y’.
def	VARCHAR(8000)	The default value to return in case the user does not have permission to unprotect.
scid	INT	Security Coordinate ID. It is not used and should be set to zero.

Returns

Unprotected data as VARCHAR.

Protected data, if the option is configured in policy and user does not have access to data.

NULL: When user has no access to the data in the policy.

Exception

If configured in policy and user does not have access, the function terminates with an error message explaining what went wrong.

> **Note:**
> Ensure that you use the supported data element only. If an unsupported data element is passed, error is returned.

Example

```
DECLARE
  @result INT,
  @encrypt VARCHAR64,
  @data VARCHAR64
SET @data='232432423432'
EXEC @result= master.dbo.xp_pty_tpe_insert @encrypt output,@data, 'TE_N_S16_L0R0_Y',0
IF @result=0
  PRINT 'OK'
ELSE
  PRINT 'ERROR'
PRINT @encrypt
SELECT @data = master.dbo.pty_select2(@encrypt,'TE_N_S16_L0R0_Y',null,0)
PRINT @data
```

4.3.3.4 pty_selectint

This function unprotects data protected by an integer tokenizing data element.

pty_selectint(data INT, dataelement VARCHAR, def INT, scid INT)

Parameters

Name	*Type*	*Description*
data	INT	Data to unprotect
dataelement	VARCHAR(64)	Name of data element
def	INT	The default value to return in case the user does not have permission to unprotect.
scid	INT	Security Coordinate ID. It is not used and should be set to zero.

Returns

Unprotected value as INT.

NULL: When user has no access to the data in the policy.

Exception

If configured in policy and user does not have access, the function terminates with an error message explaining what went wrong.

Example

```
DECLARE
            @result INT,
            @encrypt VARCHAR(64),
            @data INT
```

```
SET @data= 2324
EXEC @result= master.dbo.xp_pty_tpe_insert @encrypt output, @data, 'TE_INT_4', 0
IF @result = 0
     PRINT 'OK'
ELSE
     PRINT 'ERROR'
PRINT @encrypt
SELECT @data = master.dbo.pty_selectint(@encrypt,'TE_INT_4',null,0)
PRINT @data
```

4.3.3.5 xp_pty_select

This function unprotects data protected by an encrypting data element and can also be used when the Security Coordinate ID is not defined.

xp_pty_select(data VARBINARY, dataelement VARCHAR, def VARCHAR)

Parameters

Name	*Type*	*Description*
data	VARBINARY(8000)	Data to unprotect
dataelement	VARCHAR(64)	Name of data element
def	VARCHAR(8000)	The default value to return in case the user does not have permission to unprotect.

Returns

Returns a VARCHAR that represents the unprotected data.

NULL: When user has no access to the data in the policy.

Exception

If configured in policy and user does not have access, the function terminates with an error message explaining what went wrong.

Example

```
DECLARE
@result INT,
@encrypt VARBINARY(64)
@data VARCHAR(64),
SET data= '232432423432'
EXEC @result = master.dbo.xp_pty_insert @encrypt output,@data, 'AES256',0
IF @result=0
  PRINT 'OK'
ELSE
  PRINT 'ERROR'
PRINT @encrypt
SELECT @data = master.dbo.xp_pty_select(@encrypt,'AES256',null)
PRINT @data
```

4.3.4 INSERT Procedures

These extended procedures are used when protecting data in Insert queries.

4.3.4.1 xp_pty_insert

This stored procedure protects data with an encrypting data element.

xp_pty_insert(encrypt VARBINARY OUTPUT, data VARCHAR,dataelement VARCHAR, scid INT)

Parameters

Name	*Type*	*Description*
encrypt	VARBINARY(8000)	Result of the protect operation
data	VARCHAR(8000)	Data to protect
dataelement	VARCHAR(64)	Name of data element
scid	INT	Security Coordinate ID, which is not used; value should be set to zero.

Returns

Protected data in output parameter *Crypt.*

- 0 – if user has Insert access
- 1 – if user does not have the access or for input error

Exception

If the user does not have protect access rights in the policy, the procedure terminates with an error message explaining what went wrong.

> **Note:**
> Ensure that you use the supported data element only. Using an unsupported data element might result in successful protection without returning any error, but corruption of data can occur.

Example

```
DECLARE
@result INT,
     @encrypt VARBINARY(64),
     @data VARCHAR(64)
SET @data='232432423432'
EXEC @result= master.dbo.xp_pty_insert @encrypt output,@data, 'AES256', 0
IF @result = 0
       PRINT 'OK'
ELSE
       PRINT 'ERROR'
PRINT @encrypt
```

4.3.4.2 xp_pty_tpe_unicode_insert

This stored procedure protects data with Unicode Base 64, Unicode Gen2, and FPE Unicode data elements.

> **Note:** This UDF does not support masking.

xp_pty_tpe_unicode_insert(encrypt NVARCHAR OUTPUT, data NVARCHAR, dataelement VARCHAR, scid INT)

Parameters

Name	*Type*	*Description*
encrypt	NVARCHAR(4000)	Result of the protect operation

Name	*Type*	*Description*
data	NVARCHAR(4000)	Data to protect
dataelement	VARCHAR(64)	Name of the data element
scid	INT	Security Coordinate ID, which is not used; value should be set to zero.

Returns

Protected data in output parameter *encrypt.*

Exception

If the user does not have protect access rights in the policy, the procedure terminates with an error message that explains what went wrong.

Example for Unicode Base64

```
DECLARE
     @result INT,
     @encrypt NVARCHAR(4000),
     @data NVARCHAR(4000)
SET @data= 'Protegrityl23'
EXEC @result= master.dbo.xp_pty_tpe_unicode_insert @encrypt output,@data,
''TE_UNICODE_BASE64_SLT23_ASTYES', 0
IF @result = 0
       PRINT 'OK'
ELSE
       PRINT 'ERROR'
PRINT @encrypt
```

Example for Unicode Gen2

```
DECLARE
     @result INT,
     @encrypt NVARCHAR(4000),
     @data NVARCHAR(4000)
SET @data= N'Â¢Â€ÃÃ,ÃƒÃ„Ã…ÃŠÃ‹ÃŒÃÃŽÃÃÃ‘Ã’Ã“ÃÃ'
EXEC @result= master.dbo.xp_pty_tpe_unicode_insert @encrypt output,@data,
'TE_UG2_SLT13_L2R2_UTF16LE_Latin1_Supplement_ASTYES', 0
IF @result = 0
       PRINT 'OK'
ELSE
       PRINT 'ERROR'
PRINT @encrypt
```

Example for FPE Unicode

```
DECLARE
     @result INT,
     @encrypt NVARCHAR(4000),
     @data NVARCHAR(4000)
SET @data= N'232432423432'
EXEC @result= master.dbo.xp_pty_tpe_unicode_insert @encrypt output,@data,
'fpe_unicode', 0
IF @result = 0
       PRINT 'OK'
ELSE
       PRINT 'ERROR'
PRINT @encrypt
```

4.3.4.3 xp_pty_tpe_insert

This stored procedure protects data with a type-preserving data element, such as Tokens, DTP2, and No Encryption for access control.

> **Note:**
> Starting from the Version 7.1, Maintenance Release 1 (MR1), the DTP2 protection method is deprecated.
>
> For assistance in switching to a different protection method, contact Protegrity.

xp_pty_tpe_insert(encrypt VARCHAR OUTPUT, data VARCHAR, dataelement VARCHAR, scid INT)

Parameters

Name	*Type*	*Description*
encrypt	VARCHAR(8000)	Result of the protect operation
data	VARCHAR(64)	Data to protect
dataelement	VARCHAR(64)	Name of data element
scid	INT	Security Coordinate ID, which is not used; value should be set to zero.

Returns

Protected data in output parameter *encrypt.*

Exception

If the user does not have protect access rights in the policy, the procedure terminates with an error message explaining what went wrong.

> **Note:**
> Ensure that you use the supported data element only. If an unsupported data element is passed, error is returned.

Example

```
DECLARE
  @result INT,
  @encrypt VARCHAR(64),
  @data VARCHAR(64)
SET @data='How are you'
EXEC @result= master.dbo.xp_pty_tpe_insert @encrypt output,@data, 'TE_A_S13_L0R0_Y', 0
IF @result = 0
 PRINT 'OK'
ELSE
 PRINT 'ERROR'
PRINT @encrypt
```

4.3.4.4 xp_pty_tpe_int_insert

This stored procedure protects integer data by an integer tokenizing data element.

xp_pty_tpe_int_insert(encrypt INT OUTPUT, data INT, dataelement VARCHAR, scid INT)

> **Note:**
> This UDF does not support *NOENC* integer tokenizing data element.

Parameters

Name	*Type*	*Description*
encrypt	INT	Result of the protect operation
data	INT	Data to protect
dataelement	VARCHAR(64)	Name of data element
scid	INT	Security Coordinate ID, which is not used; value should be set to zero.

Returns

Protected data in output parameter *encrypt.*

- 0 – if user has Insert access
- 1 – if user does not have the access or for input error

Exception

If the user does not have protect access rights in the policy, the procedure terminates with an error message explaining what went wrong.

Example

```
DECLARE
     @result INT,
     @encrypt INT,
     @data INT
SET @data='1234'
EXEC @result= master.dbo.xp_pty_tpe_int_insert @encrypt output,@data, 'Integer', 0
IF @result=0
     PRINT 'OK'
ELSE
     PRINT 'ERROR'
PRINT @encrypt
```

4.3.4.5 xp_pty_insert_hash

This stored procedure calculates the hash of integer data with an HMAC-SHA1 data element.

xp_pty_insert_hash(hash VARBINARY OUTPUT, data VARCHAR, dataelement VARCHAR, scid INT)

Parameters

Name	*Type*	*Description*
hash	VARBINARY(8000)	Protected data
data	VARCHAR(8000)	Data to protect

Name	*Type*	*Description*
dataelement	VARCHAR(64)	Name of data element
scid	INT	Security Coordinate ID, which is not used; value should be set to zero.

Returns

The result of the hash operation returned in output parameter *encrypt.*

- 0 – if user has hashing access
- 1 – if user does not have the access or for input error

Exception

If user does not have protect access rights in the policy, the procedure terminates with an error message explaining what went wrong.

Example

```
DECLARE
   @result INT,
   @hash VARBINARY(64),
   @data VARCHAR(64)
SET @data='232432423432'
EXEC @result= master.dbo.xp_pty_insert_hash @hash output, @data, 'HMAC_SHA1', 0
IF @result=0
   PRINT 'OK'
ELSE
   PRINT 'ERROR'
PRINT @hash
```

4.3.5 UPDATE Extended Stored Procedures

These stored stored procedures are used when protecting data in update statements.

4.3.5.1 xp_pty_update

This stored procedure protects data with an encrypting data element.

xp_pty_update(encrypt VARBINARY OUTPUT, data VARCHAR, dataelement VARCHAR, status CHAR, scid INT)

Parameters

Name	*Type*	*Description*
encrypt	INT	Result of the protect operation
data	VARCHAR(8000)	Data to protect
dataelement	VARCHAR(64)	Name of data element
status	CHAR(1)	Status value set to ‘T’
scid	INT	Security Coordinate ID, which is not used; value should be set to zero.

Returns

Protected data in output parameter *encrypt.*

- 0 – if user has Update access
- 1 – if user does not have the access or for other input error

Exception

If user does not have reprotect access rights in policy, the procedure terminates with an error message explaining what went wrong.

> **Note:**
> Ensure that you use the supported data element only. Using an unsupported data element might result in successful reprotection without returning any error, but corruption of data can occur.

Example

```
DECLARE
     @result INT,
     @encrypt VARBINARY(64),
     @data VARCHAR(64)
SET @data = ‘232432423432’
EXEC @result = master.dbo.xp_pty_update @encrypt output, @data, ‘AES256’, ‘T’, 0
IF @result = 0
     PRINT ‘OK’
ELSE
     PRINT ‘ERROR’
PRINT @encrypt
```

4.3.5.2 xp_pty_tpe_unicode_update

This stored procedure protects data with an FPE Unicode data element only.

> **Note:** This UDF does not support masking.

xp_pty_tpe_unicode_update(encrypt NVARCHAR OUTPUT, data NVARCHAR, dataelement VARCHAR, status CHAR, scid INT)

Parameters

Name	*Type*	*Description*
encrypt	NVARCHAR(4000)	Result of the protect operation
data	NVARCHAR(4000)	Data to protect
dataelement	VARCHAR(64)	Name of data element
status	CHAR(1)	Status value set to ‘T’
scid	INT	Security Coordinate ID, which is not used; value should be set to zero.

Returns

Protected data in output parameter *encrypt.*

Exception

If user does not have reprotect access rights in policy, the procedure terminates with an error message that explains what went wrong.

Example

```
DECLARE
  @result INT,
  @encrypt NVARCHAR(4000),
  @data NVARCHAR(4000)
SET @data = N'232432423432'
EXEC @result = master.dbo.xp_pty_tpe_unicode_update @encrypt output, @data,
'fpe_unicode', 'T', 0
IF @result = 0
 PRINT 'OK'
ELSE
 PRINT 'ERROR'
 PRINT @encrypt
```

4.3.5.3 xp_pty_tpe_update

This stored procedure protects data with a type preserving data element, such as tokens, DTP2, and No Encryption for access control.

> **Note:**
> Starting from the Version 7.1, Maintenance Release 1 (MR1), the DTP2 protection method is deprecated.
>
> For assistance in switching to a different protection method, contact Protegrity.

xp_pty_tpe_update(encrypt VARCHAR OUTPUT, data VARCHAR, dataelement VARCHAR, status CHAR, scid INT)

Parameters

Name	*Type*	*Description*
encrypt	VARCHAR(8000)	Result of the protect operation
data	VARCHAR(8000)	Data to protect
dataelement	VARCHAR(64)	Name of data element
status	CHAR(1)	Status value set to 'T'
scid	INT	Security Coordinate ID, which is not used; value should be set to zero.

Returns

Protected data in output parameter *encrypt.*

- 0 – if user has Update access
- 1 – if user does not have the access or for other input error

Exception

If user does not have reprotect access rights in policy, procedure terminates with an error message explaining what went wrong.

> **Note:**
> Ensure that you use the supported data element only. If an unsupported data element is passed, error is returned.

Example

```
DECLARE
     @result INT,
     @encrypt VARCHAR(64),
     @data VARCHAR(64)
SET @data = ‘How are you’
EXEC @result = master.dbo.xp_pty_tpe_update @encrypt output, @data, ‘TE_A_S13_L0R0_Y’,
‘T’, 0
IF @result = 0
     PRINT ‘OK’
ELSE
     PRINT ‘ERROR’
PRINT @encrypt
```

4.3.5.4 xp_pty_tpe_int_update

This stored procedure protects integer data by an integer tokenizing data element.

xp_pty_tpe_int_update(encrypt VARBINARY OUTPUT, data INT, dataelement VARCHAR, scid INT)

Parameters

Name	*Type*	*Description*
encrypt	VARBINARY(8000)	Result of the protect operation
data	INT	Data to protect
dataelement	VARCHAR(64)	Name of data element
scid	INT	Security Coordinate ID, which is not used; value should be set to zero.

Returns

Protected data in output parameter *encrypt.*

- 0 – if user has Update access
- 1 – if user does not have the access or for other input error

Exception

If the user does not have reprotect access rights in the policy, the procedure terminates with an error message explaining what went wrong.

Example

```
DECLARE
     @result INT,
     @encrypt INT,
     @data INT
SET @data ='1234'
EXEC @result= master.dbo.xp_pty_tpe_int_insert @encrypt output, @data, 'Integer', 0
IF @result = 0
     PRINT 'OK'
ELSE
     PRINT 'ERROR'
PRINT @encrypt
```

4.3.5.5 xp_pty_update_hash

This stored procedure calculates the hash value of INT data with an HMAC-SHA1 data element.

xp_pty_update_hash(hash VARBINARY OUTPUT, data VARCHAR, dataelement VARCHAR, scid INT)

Parameters

Name	Type	Description
hash	VARBINARY(8000)	Result of the hash operation
data	VARCHAR(8000)	Data to protect
dataelement	VARCHAR(64)	Name of data element
scid	INT	Security Coordinate ID, which is not used; value should be set to zero.

Returns

The result of the hash operation in VARBINARY.

- 0 – if user has Update access
- 1 – if user does not have the access, or for other input error

Exception

If the user does not have reprotect access rights in the policy, UDF terminates with an error message explaining what went wrong.

Example

```
DECLARE
     @result INT,
     @hash VARBINARY(64),
     @data VARCHAR(64)
SET @data='232432423432'
EXEC @result= master.dbo.xp_pty_update_hash @hash output,@data, 'HMAC_SHA1', 0
IF @result=0
     PRINT 'OK'
ELSE
     PRINT 'ERROR'
PRINT @hash
```

4.3.6 KEY ID Functions

These functions help to obtain information about key IDs. For example, the current key ID can be used to automate key rotation of data to create a new key ID for a data element.

4.3.6.1 pty_getCurrentKeyID

This function determines the new key ID, created using key rotation, that would be used for the protect operation.

pty_getCurrentKeyID(dataelement NVARCHAR)

Parameters

Name	*Type*	*Description*
dataelement	NVARCHAR(64)	Name of data element

Returns

Key ID as INT.

Exception

If the data element is set in the policy or if the data element does not contain the key ID, the function terminates with an error message explaining what went wrong.

Example

```
DECLARE
   @dataelement VARCHAR(64)
SET @dataelement = ‘AES256_IV_CRC_KID’
PRINT @dataelement
SELECT master.dbo.PTY_GetCurrentKeyID(@dataelement)
```

4.3.6.2 pty_GetKeyID

This function determines the key ID that has been used to protect data.

pty_getKeyID(dataelement NVARCHAR, data VARBINARY)

Parameters

Name	*Type*	*Description*
dataelement	NVARCHAR(64)	Name of data element
data	VARBINARY(8000)	Data from which key ID is retrieved

Returns

Key ID as INT that was used to protect *data*.

Exception

If the data element is set in the policy or if the data element does not contain the key ID, the function terminates with an error message explaining what went wrong.

Example

```
DECLARE
   @result INT,
   @encrypt VARBINARY(64),
   @data VARCHAR(64)
SET @data = 'John Doe'
EXEC @result= MASTER.DBO.XP_PTY_INSERT @encrypt OUTPUT,@data, 'AES256_IV_CRC_KID',0
PRINT @encrypt
SELECT @result = MASTER.DBO.PTY_GetKeyID('AES256_IV_CRC_KID',@encrypt)
PRINT @result
```

4.3.7 VARCHAR UDFs

This section provides a list of VARCHAR UDFs for protect and unprotect operations.

4.3.7.1 pty_varcharins

This UDF protects VARCHAR data with Tokenization or the No Encryption method.

pty_varcharins(data VARCHAR(8000), dataelement VARCHAR(64))

Parameters

Name	Type	Description
data	VARCHAR(8000)	Data to protect

Name	Type	Description
dataelement	VARCHAR(64)	Name of data element

Results

Returns protected value as VARCHAR, if the user has protect access to data.

Returns NULL, if NULL data is provided as input.

Exception

If set in policy and user does not have protect access to the data, then the UDF terminates with an error message explaining what went wrong.

> **Note:** Ensure that you use the supported data element only. Using an unsupported data element might result in successful protection without returning any error, but corruption of data can occur.

Example

```
select master.dbo.pty_varcharins('Protegrity123','TE_AN_L0R0_Y');
```

4.3.7.2 pty_varcharsel

This UDF unprotects the protected VARCHAR data with tokenization or the No Encryption method.

pty_varcharsel(data VARCHAR(8000), dataelement VARCHAR(64))

Parameters

Name	Type	Description
data	VARCHAR(8000)	Data to unprotect
dataelement	VARCHAR(64)	Name of data element

Results

Unprotected VARCHAR value, if the user has unprotect access to data.

Returns NULL, if the user does not have access to data.

Exception

If set in policy and the user does not have unprotect access to the data, then the UDF terminates with an error message explaining what went wrong.

> **Note:**
> Ensure that you use the supported data element only. Using an unsupported data element might result in successful unprotection without returning any error, but corruption of data can occur.

Example

```
select master.dbo.pty_varcharsel('m6g2ZC6qb0xSAY','TE_AN_L0R0_Y');
```

4.3.7.3 pty_hash_varchar

This UDF hashes and protects the VARCHAR value. This function is irreversible, and the protected data cannot be unprotected.

pty_hash_varchar(data VARCHAR(8000), dataelement VARCHAR(64))

Parameters

Name	Type	Description
data	VARCHAR(8000)	Data to protect
dataelement	VARCHAR(64)	Name of data element

Results

Returns protected value as VARBINARY, if the user has protect access to data.

Returns NULL, if NULL data is provided as Input.

Exception

If set in policy and user does not have protect access to the data, then the UDF terminates with an error message explaining what went wrong.

> **Note:**
> Ensure that you use the supported data element only. Using an unsupported data element might result in successful protection without returning any error, but corruption of data can occur.

Example

```
select master.dbo.pty_hash_varchar('Protegrityl23','HMAC_SHA1');
```

4.3.7.4 pty_varcharenc

This UDF encrypts VARCHAR data using the encryption data element.

pty_varcharenc(data VARCHAR(8000), dataelement VARCHAR(64))

Parameters

Name	Type	Description
data	VARCHAR(8000)	Data to protect
dataelement	VARCHAR(64)	Name of data element

Results

Encrypted value as VARBINARY, if the user has protect access to data.

Returns NULL, if NULL data is provided as input.

Exception

If set in policy and user does not have protect access, then the UDF terminates with an error message explaining what went wrong.

Example

```
select master.dbo.pty_varcharenc('Protegrityl23','AES256');
```

4.3.7.5 pty_varchardec

This UDF decrypts the encrypted data that was encrypted using the *pty_varcharenc* UDF.

pty_varchardec(data VARBINARY(8000), dataelement VARCHAR(64))

Parameters

Name	Type	Description
data	VARBINARY(8000)	Data to unprotect
dataelement	VARCHAR(64)	Name of data element

Returns

Clear or decrypted VARCHAR value, if the user has unprotect access to data.

Returns NULL, if the user does not have access to data.

Exception

If set in policy and user does not have unprotect access, then the UDF terminates with an error message explaining what went wrong.

Example

```
select
master.dbo.pty_varchardec(master.dbo.pty_varcharenc('Protegrityl23','AES256'),'AES256');
```

4.3.8 NVARCHAR UDFs

This section provides a list of NVARCHAR UDFs for protect and unprotect operations.

4.3.8.1 pty_unicodevarcharins

This UDF protects the NVARCHAR data with Unicode Base64, Unicode Gen2, and FPE data elements.

> **Note:** This UDF does not support masking.

pty_unicodevarcharins(data NVARCHAR(4000), dataelement VARCHAR(64))

Parameters

Name	Type	Description
data	NVARCHAR(4000)	Data to protect
dataelement	VARCHAR(64)	Name of data element

Results

Returns protected value as NVARCHAR, if the user has protect access to data.

Returns NULL, if NULL data is provided as Input.

Exception

If set in policy and user does not have protect access to the data, then the UDF terminates with an error message explaining what went wrong.

> **Note:** Ensure that you use the supported data element only. Using an unsupported data element might result in successful protection without returning any error, but corruption of data can occur.

Example for Unicode Base64

```
select
master.dbo.pty_unicodevarcharins('Protegrityl23','TE_UNICODE_BASE64_SLT13_ASTYES');
```

Example for Unicode Gen2

> **Note:** Unicode Gen2 data elements supports newly introduced SLT_X_1 tokenizer from 9.1 version onwards along with existing SLT_1_3 tokenizer. For more information, refer to section *3.4.15 Unicode Gen2* in the *Protection Methods Reference Guide 9.1.0.0*.

```
select master.dbo.pty_unicodevarcharins(N'Â¢Â€ÃÃ,ÃƒÃ„Ã…
ÃŠÃ‹ÃŒÃÃŽÃÃÃ‘Ã’Ã“ÃÃ','TE_UG2_SLT13_L2R2_UTF16LE_Latin1_Supplement_ASTYES');
```

```
select
master.dbo.pty_unicodevarcharins(N'▒','TE_UG2_SLTX1_L2R2_N_IPA_Greek_Coptic_UTF16LE');
```

4.3.8.2 pty_unicodevarcharsel

This UDF unprotects the protected NVARCHAR data with Unicode Base64, Unicode Gen2, and FPE data elements.

> **Note:** This UDF does not support masking.

pty_unicodevarcharsel (data NVARCHAR(4000), DataElement VARCHAR(64))

Parameters

Name	Type	Description
data	NVARCHAR(4000)	Data to unprotect
dataelement	VARCHAR(64)	Name of data element

Results

Unprotected NVARCHAR value, if the user has unprotect access to data.

Returns NULL, if the user does not have access to data.

Exception

If set in policy and user does not have unprotect access to the data, then the UDF terminates with an error message explaining what went wrong.

> **Note:** Ensure that you use the supported data element only. Using an unsupported data element might result in successful unprotection without returning any error, but corruption of data can occur.

Example for Unicode Base64

```
select
master.dbo.pty_unicodevarcharsel(master.dbo.pty_unicodevarcharins('Protegrityl23','TE_UN
ICODE_BASE64_SLT13_ASTYES'),'TE_UNICODE_BASE64_SLT13_ASTYES');
```

Example for Unicode Gen2

> **Note:** Unicode Gen2 data elements supports newly introduced SLT_X_1 tokenizer from 9.1 version onwards along with existing SLT_1_3 tokenizer. For more information, refer to section *3.4.15 Unicode Gen2* in the *Protection Methods Reference Guide 9.1.0.0*.

```
select
master.dbo.pty_unicodevarcharsel(master.dbo.pty_unicodevarcharins(N'Â¢Â€ÃÃ,ÃƒÃ„Ã…
ÃŠÃ‹ÃŒÃÃŽÃÃ'Ã'Ã“ÃÃ,'TE_UG2_SLT13_L2R2_UTF16LE_Latin1_Supplement_ASTYES'),'TE_UG2_SLT13_
L2R2_UTF16LE_Latin1_Supplement_ASTYES');
```

```
select
master.dbo.pty_unicodevarcharsel(master.dbo.pty_unicodevarcharins(N'
▒,'TE_UG2_SLTX1_L2R2_N_IPA_Greek_Coptic_UTF16LE'),'TE_UG2_SLTX1_L2R2_N_IPA_Greek_Coptic_
UTF16LE');
```

4.3.9 INTEGER UDFs

This section provides a list of Integer UDFs for protect and unprotect operations.

4.3.9.1 pty_integerins

This UDF protects the Integer data with tokenization method.

pty_integerins(data Integer, dataelement VARCHAR(64))

Parameters

Name	Type	Description
data	Integer	Data to protect
dataelement	VARCHAR(64)	Name of data element

Results

Returns protected value as Integer, if the user has protect access to data.

Returns NULL, if NULL data is provided as Input.

Exception

If set in policy and user does not have protect access to the data, then the UDF terminates with an error message explaining what went wrong.

> **Note:** Ensure that you use the supported data element only. Using an unsupported data element might result in successful protection without returning any error, but corruption of data can occur.

Example

```
select master.dbo.pty_integerins(123456,'TE_INT_4');
```

4.3.9.2 pty_integersel

This UDF unprotects the protected Integer data with tokenization method.

pty_integersel(data Integer, dataelement VARCHAR(64))

Parameters

Name	Type	Description
data	Integer	Data to unprotect
dataelement	VARCHAR(64)	Name of data element

Results

Unprotected Integer value, if the user has unprotect access to data.

Returns NULL, if the user does not have access to data.

Exception

If set in policy and user does not have unprotect access to the data, then the UDF terminates with an error message explaining what went wrong.

> **Note:** Ensure that you use the supported data element only. Using an unsupported data element might result in successful unprotection without returning any error, but corruption of data can occur.

Example

```
select master.dbo.pty_integersel(master.dbo.pty_integerins(123456, 'TE_INT_4'),
'TE_INT_4');
```

4.3.9.3 pty_integerenc

This UDF encrypts Integer data using an encryption data element.

pty_integerenc(data Integer, dataelement VARCHAR(64))

Parameters

Name	Type	Description
data	Integer	Data to protect
dataelement	VARCHAR(64)	Name of data element

Returns

Encrypted value as VARBINARY, if user has protect access to data.

Returns NULL, if NULL data is provided as Input.

Exception

If set in policy and user does not have protect access to the data, then the UDF terminates.

Example

```
select master.dbo.pty_integerenc(1234,'AES256');
```

4.3.9.4 pty_integerdec

This UDF decrypts the encrypted data that was encrypted using the *pty_integerenc* UDF.

pty_integerdec(data VARCHAR(8000), dataelement VARCHAR(64))

Parameters

Name	Type	Description
data	VARBINARY(8000)	Data to unprotect
dataelement	VARCHAR(64)	Name of data element

Returns

Clear or unprotected Integer value, if user has unprotect access to data.

Returns NULL, if the user does not have access to data.

Exception

If set in policy and user does not have unprotect access to the data, then the UDF terminates.

Example

```
select master.dbo.pty_integerdec(master.dbo.pty_integerenc(1234,'AES256'),'AES256');
```

4.3.10 BLOB UDFs

These UDFs can be used to encrypt and decrypt the data stored as BLOB.

4.3.10.1 pty_blobenc

This function is used to encrypt data stored as VARBINARY(max) using any encryption data element.

pty_blobenc(data VARBINARY(max), dataelement VARCHAR(64))

Parameters

Name	Type	Description
data	VARBINARY(max*)	Data to protect
dataelement	VARCHAR(64)	Name of data element

Warning:
This function supports encryption for data up to 1 GB. Exceeding this limit will result in memory issues.

Results

Returns protected value as VARBINARY(max*) if the user has protect access to data.

Returns NULL, if NULL data is provided as input.

Note: *The max parameter specifies the maximum length of input and output data, which depends on the specified maximum storage limitation of the VARBINARY data type supported by the MS SQL database.

Exception

If the user does not have protect privileges in the policy, then the UDF terminates with an error message explaining what went wrong.

> **Note:** Ensure that you use the supported data element only. Using an unsupported data element might result in successful protection without returning any error, but corruption of data can occur.

Example

```
select master.dbo.pty_blobenc(cast('Protegrity' as varbinary(max)),'AES256');
```

4.3.10.2 pty_blobdec

This function is used to decrypt encrypted data stored as VARBINARY(max*) using any encryption data element.

pty_blobdec(data VARBINARY(max), dataelement VARCHAR(64))

Parameters

Name	Type	Description
data	VARBINARY(max*)	Data to protect
dataelement	VARCHAR(64)	Name of data element

Results

Returns unprotected value as VARBINARY(max*) if the user has unprotect access to data.

Returns NULL, if the user does not have access to data.

> **Note:** *The max parameter specifies the maximum length of input and output data, which depends on the specified maximum storage limitation of the VARBINARY data type supported by the MS SQL database.

Exception

If the user does not have unprotect privileges in the policy, then the UDF terminates with an error message explaining what went wrong.

> **Note:** Ensure that you use the supported data element only. Using an unsupported data element might result in successful unprotection without returning any error, but corruption of data can occur.

Example

```
select master.dbo.pty_blobdec(master.dbo.pty_blobenc(cast('Protegrity' as
varbinary(max)),'AES256'),'AES256')
```

4.3.11 CLOB UDFs

These UDFs can be used to encrypt and decrypt the data stored in CLOB.

4.3.11.1 pty_clobenc

This function is used to encrypt data stored as VARCHAR(max*) using any encryption data element.

pty_clobenc(data VARCHAR(max), dataelement VARCHAR(64))

Parameters

Name	Type	Description
data	VARCHAR(max*)	Data to protect
dataelement	VARCHAR(64)	Name of data element

Warning:
This function supports encryption for data up to 1.5 GB. Exceeding this limit will result in memory issues.

Results

Returns protected value as VARBINARY(max*) if the user has protect access to data.

Returns NULL, if NULL data is provided as Input.

Note: *The max parameter specifies the maximum length of input and output data, which depends on the specified maximum storage limitation of the VARCHAR and VARBINARY data types supported by the MS SQL database.

Exception

If the user does not have protect privileges in the policy, then the UDF terminates with an error message explaining what went wrong.

Note: Ensure that you use the supported data element only. Using an unsupported data element might result in successful protection without returning any error, but corruption of data can occur.

Example

```
select master.dbo.pty_clobenc('Protegrity','AES256');
```

4.3.11.2 pty_clobdec

This function is used to decrypt encrypted data stored as VARBINARY(max*) using any encryption data element.

pty_clobdec(data VARBINARY(max), dataelement VARCHAR(64))

Parameters

Name	Type	Description
data	VARBINARY(max*)	Data to protect
dataelement	VARCHAR(64)	Name of data element

Results

Returns unprotected value as VARCHAR(max*) if the user has unprotect access to data.

Returns NULL, if the user does not have access to data.

Note: *The max parameter specifies the maximum length of input and output data, which depends on the specified maximum storage limitation of the VARCHAR and VARBINARY data types supported by the MS SQL database.

Exception

If the user does not have unprotect privileges in the policy, then the UDF terminates with an error message explaining what went wrong.

> **Note:** Ensure that you use the supported data element only. Using an unsupported data element might result in successful unprotection without returning any error, but corruption of data can occur.

Example

```
select master.dbo.pty_clobdec(dbo.pty_clobenc('Protegrity','AES256'),'AES256');
```

4.4 Netezza DB Protector UDFs

This section provides a detailed list of UDFs (User Defined Functions) for general information, and protection and unprotection of different data types.

> **Note:**
> The execution mode for Netezza UDFs is fenced, by default. Fenced mode defines if the function is executed in a separate process, in a protected address space. Thus, the fenced UDFs protect the database from any malicious activity such as accessing database resources like files or shared memory. Also, since the fenced UDFs are executed in a separate process, any failure in the execution doesn't impact the database, and any other concurrent processing continues unaffected. Unfenced UDFs access the database directly and any interruption or issue with the process may cause the database to crash.
>
> The execution of UDFs in unfenced mode is not supported. Running the UDF in unfenced mode returns the following error message.
>
> *Unable to access policy in shared memory*

4.4.1 General UDFs

4.4.1.1 PTY_WHOAMI

This UDF returns the name of the user who is currently logged in.

PTY_WHOAMI()

Parameters

None

Returns

Name of user logged on to the database.

Example

```
SELECT pty_whoami();
```

4.4.1.2 PTY_GETVERSION

This UDF returns the version of the PEP Server.

PTY_GETVERSION()

Parameters

None

Returns

Product version as STRING

Example

```
SELECT pty_getversion();
```

4.4.1.3 PTY_GETCURRENTKEYID

This UDF returns the current key ID for a data element. It is typically used together with PTY_GETKEYID to determine if some data is protected with the most recent key for a given data element. If not, then the data can be reprotected with the most recent key.

PTY_GETCURRENTKEYID(dataelement VARCHAR)

Parameters

Name	Type	Description
dataelement	VARCHAR	Name of data element

Returns

Current key ID as INTEGER

Exception

If set in policy and user does not have access, then the UDF terminates with an error message explaining what went wrong.

Example

```
select PTY_GETCURRENTKEYID('AES256_IV_CRC_KID');
```

4.4.1.4 PTY_GETKEYID

This UDF returns the key ID that was used to protect an item of data. It is typically used together with PTY_GETCURRENTKEYID to determine if some data is protected with the most recent key for a given data element.

PTY_GETKEYID(data VARCHAR, dataelement VARCHAR)

Parameters

Name	Type	Description
data	VARCHAR	Data that has been protected with encryption and is using key ID
dataelement	VARCHAR	Name of data element

Returns

Key ID as INTEGER

Exception

If set in policy and user does not have access, then the UDF terminates with an error message explaining what went wrong.

Example

```
select PTY_GETKEYID(PTY_VARCHARENC(‘ProtegrityProtegrity’,
'AES128_IV_CRC_KID'),'AES128_IV_CRC_KID’);
```

4.4.2 VARCHAR UDFs

4.4.2.1 PTY_VARCHARENC

This UDF protects VARCHAR data with strong encryption.

PTY_VARCHARENC(data VARCHAR, dataelement VARCHAR)

Parameters

Name	Type	Description
data	VARCHAR	Data to protect
dataelement	VARCHAR	Name of data element

Returns

Protected value as VARCHAR

Exception

If set in policy and user does not have access, then the UDF terminates with an error message explaining what went wrong.

Example

```
SELECT pty_varcharenc('9876987698769876','DE_AES256');
```

4.4.2.2 PTY_VARCHARDEC

This UDF unprotects the protected data.

PTY_VARCHARDEC(data VARCHAR, dataelement VARCHAR)

Parameters

Name	Type	Description
data	VARCHAR	Data to unprotect
dataelement	VARCHAR	Name of data element

Returns

Unprotected value as VARCHAR

NULL: When user has no access to database

Exception

If set in policy and user does not have access, then the UDF terminates with an error message explaining what went wrong.

Example

```
select pty_varchardec(pty_varcharenc('1234123412341234', 'DE_AES256'), 'DE_AES256');
```

4.4.2.3 PTY_VARCHARINS

This UDF protects VARCHAR data with tokenization, type preserving encryption, or no-encryption.

PTY_VARCHARINS(data VARCHAR, dataelement VARCHAR)

Parameters

Name	Type	Description
data	VARCHAR	Data to protect
dataelement	VARCHAR	Name of data element

Returns

Protected value as VARCHAR

NULL: When user has no access to database

Exception

If set in policy and user does not have access, then the UDF terminates with an error message explaining what went wrong.

Example

```
SELECT pty_varcharins ('ProtegrityProtegrity', ‘TE_A_N_S23_L2R2_Y’);
```

4.4.2.4 PTY_VARCHARSEL

This UDF unprotects protected data.

PTY_VARCHARSEL(data VARCHAR, dataelement VARCHAR)

Parameters

Name	Type	Description
data	VARCHAR	Data to unprotect
dataelement	VARCHAR	Name of data element

Returns

Unprotected value as VARCHAR

NULL: When user has no access to database

Exception

If set in policy and user does not have access, then the UDF terminates with an error message explaining what went wrong.

Example

```
select pty_varcharsel
(pty_varcharins('ProtegrityProtegrity','TE_A_N_S23_L2R2_Y'),'TE_A_N_S23_L2R2_Y');
```

4.4.3 INTEGER UDFs

4.4.3.1 PTY_INTEGERENC

This UDF protects INTEGER data with strong encryption.

PTY_INTEGERENC(data INTEGER, dataelement VARCHAR)

Parameters

Name	Type	Description
data	INTEGER	Data to protect
dataelement	VARCHAR	Name of data element

Returns

Protected value as VARCHAR

NULL: When user has no access to database

Exception

If set in policy and user does not have access, then the UDF terminates with an error message explaining what went wrong.

Example

```
select pty_integerenc(123456, 'AES128');
```

4.4.3.2 PTY_INTEGERDEC

This UDF unprotects the protected data.

PTY_INTEGERDEC(data VARCHAR, dataelement VARCHAR)

Parameters

Name	Type	Description
data	VARCHAR	Data to unprotect
dataelement	VARCHAR	Name of data element

Returns

Unprotected value as INTEGER

NULL: When user has no access to database

Exception

If set in policy and user does not have access, then the UDF terminates with an error message explaining what went wrong.

Example

```
select pty_integerdec(pty_integerenc(123456, 'AES128'), 'AES128');
```

4.4.3.3 PTY_INTEGERINS

This UDF protects INTEGER data with tokenization, type preserving encryption, or no-encryption.

PTY_INTEGERINS(data INTEGER, dataelement VARCHAR)

Parameters

Name	Type	Description
data	INTEGER	Data to unprotect
dataelement	VARCHAR	Name of data element

Returns

Protected value as INTEGER

NULL: When user has no access to database

Exception

If set in policy and user does not have access, then the UDF terminates with an error message explaining what went wrong.

Example

```
select pty_integerins(123456, 'TE_INT_4');
```

4.4.3.4 PTY_INTEGERSEL

This UDF unprotects the protected data.

PTY_INTEGERSEL(data INTEGER, dataelement VARCHAR)

Parameters

Name	Type	Description
data	INTEGER	Data to unprotect
dataelement	VARCHAR	Name of data element

Returns

Unprotected value as INTEGER

NULL: When user has no access to database

Exception

If set in policy and user does not have access, then the UDF terminates with an error message explaining what went wrong.

Example

```
select pty_integersel(pty_integerins(123456, 'TE_INT_4'), 'TE_INT_4');
```

4.4.4 DATE UDFs

4.4.4.1 PTY_DATEENC

This UDF protects DATE data with strong encryption.

PTY_DATEENC(data DATE, dataelement VARCHAR)

Parameters

Name	Type	Description
data	DATE	Data to protect
dataelement	VARCHAR	Name of data element

Returns

Protected value as VARCHAR

NULL: When user has no access to database

Exception

If set in policy and user does not have access, then the UDF terminates with an error message explaining what went wrong.

Example

```
select pty_dateenc(to_date('22-09-1990', 'DD-MM-YYYY'), 'AES128');
```

4.4.4.2 PTY_DATEDEC

This UDF unprotects protected data.

PTY_DATEDEC(data VARCHAR, dataelement VARCHAR)

Parameters

Name	Type	Description
data	VARCHAR	Data to protect
dataelement	VARCHAR	Name of data element

Returns

Unprotected value as DATE

NULL: When user has no access to database

Exception

If set in policy and user does not have access, then the UDF terminates with an error message explaining what went wrong.

Example

```
select pty_datedec(pty_dateenc(to_date('22-09-1990', 'DD-MM-YYYY'), 'AES128'),
'AES128');
```

4.4.4.3 PTY_DATEINS

This UDF protects DATE data with tokenization, type preserving encryption, or no-encryption.

PTY_DATEINS(data DATE, dataelement VARCHAR)

Parameters

Name	Type	Description
data	DATE	Data to protect
dataelement	VARCHAR	Name of data element

Returns

Protected value as DATE

NULL: When user has no access to database

Exception

If set in policy and user does not have access, then the UDF terminates with an error message explaining what went wrong.

Example

```
select pty_dateins(to_date('22-09-1990', 'DD-MM-YYYY'), 'TE_Date_YMD_S26');
```

4.4.4.4 PTY_DATESEL

This UDF unprotects protected data.

PTY_DATESEL(data DATE, dataelement VARCHAR)

Parameters

Name	Type	Description
data	DATE	Data to unprotect
dataelement	VARCHAR	Name of data element

Returns

Unprotected value as DATE

NULL: When user has no access to database

Exception

If set in policy and user does not have access, then the UDF terminates with an error message explaining what went wrong.

Example

```
select pty_datesel(pty_dateins(to_date('22-09-1990', 'DD-MM-YYYY'), 'TE_Date_YMD_S26'),
'TE_Date_YMD_S26');
```

4.4.5 REAL UDFs

4.4.5.1 PTY_REALENC

This UDF protects REAL data with encryption.

PTY_REALENC(data REAL, dataelement VARCHAR)

Parameters

Name	Type	Description
data	REAL	Data to protect
dataelement	VARCHAR	Name of data element

Returns

Protected value as VARCHAR

NULL: When user has no access to database

Exception

If set in policy and user does not have access, then the UDF terminates with an error message explaining what went wrong.

Example

```
select pty_realenc('12345e+12', 'AES128');
```

4.4.5.2 PTY_REALDEC

This UDF unprotects protected data.

PTY_REALDEC(data VARCHAR, dataelement VARCHAR)

Parameters

Name	Type	Description
data	VARCHAR	Data to unprotect
dataelement	VARCHAR	Name of data element

Returns

Unprotected value as REAL

NULL: When user has no access to database

Exception

If set in policy and user does not have access, then the UDF terminates with an error message explaining what went wrong.

Example

```
select pty_realdec(pty_realenc('12345e+12', 'AES128'),'AES128');
```

4.4.5.3 PTY_REALINS

This UDF protects REAL data with tokenization, type preserving encryption, or no-encryption.

PTY_REALINS(data REAL, dataelement VARCHAR)

Parameters

Name	Type	Description
data	REAL	Data to unprotect
dataelement	VARCHAR	Name of data element

Returns

Protected value as REAL

NULL: When user has no access to database

Exception

If set in policy and user does not have access, then the UDF terminates with an error message explaining what went wrong.

Example

```
select pty_realins('12345e+12', 'NoEncryption');
```

4.4.5.4 PTY_REALSEL

This UDF unprotects protected data.

PTY_REALSEL(data REAL, dataelement VARCHAR)

Parameters

Name	Type	Description
data	REAL	Data to unprotect
dataelement	VARCHAR	Name of data element

Returns

Unprotected value as REAL datatype

NULL: When user has no access to database

Exception

If set in policy and user does not have access, then the UDF terminates with an error message explaining what went wrong.

Example

```
select pty_realsel(pty_realins('12345e+12', 'NoEncryption'), 'NoEncryption');
```

4.5 Oracle User Defined Functions and Procedures

This section provides a detailed list of Oracle User Defined Functions and Procedures (UDFs) that are used to obtain general information, and protect and unprotect data for different data types. The protection method can be either encryption or tokenization.

Note:

- All UDFs are preceded by the container name pty.
- Data security operations, such as, protect, unprotect, and reprotect will fail if the user name contains Unicode characters.

4.5.1 General UDFs

4.5.1.1 pty.whoami

This UDF returns the name of the user who is currently logged in to the database.

pty.whoami()

Parameters

None

Returns

This UDF returns the name of the user as the VARCHAR2 string.

Exception

None

Example

```
select pty.whoami() ”Test of WhoAmI” from dual;
Test of WhoAmI
---
USER1
```

4.5.1.2 pty.getversion

This UDF returns the version of the protector.

pty.getversion()

Parameters

None

Returns

This UDF returns the version of the protector as the VARCHAR2 string.

Example

```
select pty.getversion() ”Test of GetVersion” from dual;

Test of GetVersion
---
x.x.x.x
```

4.5.1.3 pty.getcurrentkeyid

This UDF returns the current key ID for a data element. It is typically used together with **getkeyid** to determine if data is protected with the most recent key for a given data element.

pty.getcurrentkeyid(dataelement VARCHAR)

Parameters

Name	Type	Description
dataelement	VARCHAR	Specifies the name of data element.

Returns

This UDF returns the current key ID as the BINARY INTEGER.

Exception

If the data element is missing in the policy or if the data element does not contain a key ID, the function terminates with an error message explaining what went wrong.

Example

```
select pty.getCurrentKeyID('DE_AES256_IV_CRC_KID') ”Test of getCurrentKeyID” from dual;

Test of getCurrentKeyID
---
4
```

4.5.1.4 pty.getkeyid

This UDF returns the key ID of the data element key that was used to protect a value of data. It is typically used together with **getcurrentkeyid** to determine if data is protected with the most recent key for a given encryption data element.

pty.getkeyid(dataelement VARCHAR, data RAW)

Parameters

Name	Type	Description
dataelement	VARCHAR	Specifies the name of data element.
data	RAW	Specifies the data that has been protected with encryption and is using key ID.

Returns

This UDF returns the key ID as the BINARY INTEGER.

Exception

None

Example

```
select pty.getKeyID('AES256_IV_CRC_KID',PTY.ins_encrypt('AES256_IV_CRC_KID',
  'Original data', 0)) “Test of getKeyID” from dual;

Test of getKeyID
-------
4
```

4.5.2 Access Check Procedures

These procedures checks whether user is granted access permissions to the data element. The procedures will pass if user has access; otherwise, it casts an exception with the reason for failure.

> **Note:** The permissions for protect, unprotect, and reprotect are defined based on the roles assigned to the user. For more information about how to grant these permissions and assign roles, refer to the *Policy Management Guide*.

4.5.2.1 pty.sel_check

This procedure is used to determine if the user has **select (unprotect)** access to a data element.

pty.sel_check(dataelement VARCHAR)

Parameters

Name	Type	Description
dataelement	VARCHAR	Specifies the name of data element.

Returns

This UDF returns the value as Success, if user has access.

Example

```
declare
begin
  dbms_output.put_line('Test of SELECT check procedure');
  dbms_output.put_line('-----------------------------');
  pty.sel_check('DE_AES256');
end;
```

4.5.2.2 pty.upd_check

This procedure determines if the user has **update (reprotect)** access to a data element.

pty.upd_check(dataelement VARCHAR)

Parameters

Name	Type	Description
dataelement	VARCHAR	Specifies the name of data element.

Returns

This UDF returns the value as Success, if user can update.

Example

```
declare
begin
  dbms_output.put_line('Test of UPDATE check procedure');
  dbms_output.put_line('-----------------------------');
  pty.upd_check('DE_AES256');
end;
```

4.5.2.3 pty.ins_check

This procedure determines if the user has **insert (protect)** access to a data element.

pty.ins_check(dataelement VARCHAR)

Parameters

Name	Type	Description
dataelement	VARCHAR	Specifies the name of data element.

Returns

This UDF returns the value as Success, if user can insert data.

Example

```
declare
begin
  dbms_output.put_line('Test of INSERT check procedure');
  dbms_output.put_line('-----------------------------');
```

```
  pty.ins_check('DE_AES256');
end;
```

4.5.2.4 pty.del_check

This procedure determines if the user has *delete* access to a data element. The **pty.del_check** is added to a delete trigger that is associated with a table. When any delete operation is initiated on the table, the **pty.del_check** UDF checks if the user is granted delete access on the table. If the user has delete access, the delete operation succeeds.

> **Note:** The delete access must be assigned to a role from the ESA. As a default, the delete access is not visible on the UI. You must enable delete access setting in the UI.
>
> For more information, refer to section *Enabling the Delete Permission* in the *Policy Management Guide*.

pty.del_check(dataelement VARCHAR, scid BINARY_INTEGER)

Parameters

Name	Type	Description
dataelement	VARCHAR	Specifies the name of data element.
scid	BINARY_INTEGER	Specifies the security co-ordinate ID, which is not used. The value should be set to zero

Returns

This UDF returns the value as Success, if user can delete.

Example

```
declare
begin
  dbms_output.put_line('Test of DELETE check procedure');
  dbms_output.put_line('-----------------------------');
  pty.del_check('DE_AES256', 0);
end;
```

4.5.3 MULTIPLE INSERT ENCRYPTION Procedures

These procedures encrypt one to four values of data with one procedure call. The user must be granted *Insert* access for the data element that will be used to execute these procedures. You can use the **ins_check** UDF to check if the user has insert access.

4.5.3.1 pty.encInsert

This procedure encrypts one value of VARCHAR2 data with one data element for encryption.

pty.encInsert(dataelement VARCHAR2, cdata VARCHAR2, rdata RAW, scid BINARY_INTEGER)

Parameters

Name	Type	Description
dataelement	VARCHAR2	Specifies the name of data element.
cdata	VARCHAR2	Specifies the input data

Name	Type	Description
rdata	RAW	Specifies the encrypted output data
scid	BINARY_INTEGER	Specifies the security co-ordinate ID, which is not used. The value should be set to zero.

Returns

This UDF returns the encrypted value as RAW data.

Exception

If the user does not have protect access rights in the policy, then the procedure terminates with an error message explaining what went wrong.

Example

```
declare
  raw_out raw(2000);
begin
  dbms_output.put_line('Test of INSERT multi encryption procedure for 1
    COLUMN');
  dbms_output.put_line('-----------------------------------------------');
  pty.encInsert('DE_AES256', 'ASFGFGghg5577fFFyu', raw_out, 0);
  DBMS_OUTPUT.PUT_LINE('Encrypted data: ' || raw_out);
end;
```

4.5.3.2 pty.ins_encryptx2

This procedure encrypts two values of VARCHAR2 data with two data elements for encryption.

pty.ins_encryptx2(dataelement1 VARCHAR2, cdata1 VARCHAR2, rdata1 RAW, scid1 BINARY_INTEGER, dataelement2 VARCHAR2, cdata2 VARCHAR2, rdata2 RAW, scid2 BINARY_INTEGER)

Parameters

Name	Type	Description
dataelement1	VARCHAR2	Speicifies the name of data element
cdata1	VARCHAR2	Specifies the input data
rdata1	RAW	Specifies the encrypted output data
scid1	BINARY_INTEGER	Specifies the security co-ordinate ID, which is not used. The value should be set to zero.
dataelement2	VARCHAR2	Speicifies the name of data element
cdata2	VARCHAR2	Specifies the input data
rdata2	RAW	Specifies the encrypted output data
scid2	BINARY_INTEGER	Specifies the security co-ordinate ID, which is not used. The value should be set to zero.

Returns

This UDF returns the encrypted values as VARCHAR2.

Exception

If the user does not have protect access rights in the policy, then the procedure terminates with an error message explaining what went wrong.

Example

```
Encrypted values are the output parameters
declare
  raw_out1 raw(2000);
  raw_out2 raw(2000);
begin
  dbms_output.put_line('Test of INSERT multi encryption procedure for 2
    COLUMNS');
  dbms_output.put_line('-------------------------------------------');
  pty.ins_encryptx2('DE_AES256', 'ASFGFGghg5577fFFyu', raw_out1, 0,
    'DE_AES256', 'IyutGGg76hg8h1', raw_out2, 0);
  DBMS_OUTPUT.PUT_LINE('Encrypted data1: ' || raw_out1);
  DBMS_OUTPUT.PUT_LINE('Encrypted data2: ' || raw_out2);
end;
```

4.5.3.3 pty.ins_encryptx3

This procedure encrypts three values of VARCHAR2 data with three data elements for encryption.

pty.ins_encryptx3(dataelement1 VARCHAR2, cdata1 VARCHAR2, rdata1 RAW, scid1 BINARY_INTEGER, dataelement2 VARCHAR2, cdata2 VARCHAR2, rdata2 RAW, scid2 BINARY_INTEGER, dataelement3 VARCHAR2, cdata3 VARCHAR2, rdata3 RAW, scid3 BINARY_INTEGER)

Parameters

Name	Type	Description
dataelement1	VARCHAR2	Specifies the name of data element
cdata1	VARCHAR2	Specifies the input data
rdata1	RAW	Specifies the encrypted output data
scid1	BINARY_INTEGER	Specifies the security co-ordinate ID, which is not used. The value should be set to zero.
dataelement2	VARCHAR2	Specifies the name of data element
cdata2	VARCHAR2	Specifies the input data
rdata2	RAW	Specifies the encrypted output data
scid2	BINARY_INTEGER	Specifies the security co-ordinate ID, which is not used. The value should be set to zero.
dataelement3	VARCHAR3	Specifies the name of data element
cdata3	VARCHAR3	Specifies the input data
rdata3	RAW	Specifies the encrypted output data

Name	Type	Description
scid3	BINARY_INTEGER	Specifies the security co-ordinate ID, which is not used. The value should be set to zero.

Returns

This UDF returns the encrypted values as VARCHAR2.

Exception

If the user does not have protect access rights in the policy, then the procedure terminates with an error message explaining what went wrong.

Example

```
declare
  raw_out1 raw(2000);
  raw_out2 raw(2000);
  raw_out3 raw(2000);
begin
  dbms_output.put_line('Test of INSERT multi encryption procedure for 3
    COLUMNS');
  dbms_output.put_line('-------------------------------------------');
  pty.ins_encryptx3('DE_AES256', 'ASFGFGghg5577fFFyu', raw_out1, 0,
    'DE_AES256', 'IyutGGg76hg8h1', raw_out2, 0, 'DE_AES256', 'AAaazzZZ1199',
    raw_out3, 0);
  DBMS_OUTPUT.PUT_LINE('Encrypted data1: ' || raw_out1);
  DBMS_OUTPUT.PUT_LINE('Encrypted data2: ' || raw_out2);
  DBMS_OUTPUT.PUT_LINE('Encrypted data3: ' || raw_out3);
end;
```

4.5.3.4 pty.ins_encryptx4

This procedure encrypts four values of VARCHAR2 data with four data elements for encryption.

pty.ins_encryptx4(dataelement1 VARCHAR2, cdata1 VARCHAR2, rdata1 RAW, scid1 BINARY_INTEGER, dataelement2 VARCHAR2, cdata2 VARCHAR2, rdata2 RAW, scid2 BINARY_INTEGER, dataelement3 VARCHAR2, cdata3 VARCHAR2, rdata3 RAW, scid3 BINARY_INTEGER, dataelement4 VARCHAR2, cdata4 VARCHAR2, rdata4 RAW, scid4 BINARY_INTEGER)

Parameters

Name	Type	Description
dataelement1	VARCHAR2	Specifies the name of data element
cdata1	VARCHAR2	Specifies the input data
rdata1	RAW	Specifies the encrypted output data
scid1	BINARY_INTEGER	Specifies the security co-ordinate ID, which is not used. The value should be set to zero.
dataelement2	VARCHAR2	Specifies the name of data element
cdata2	VARCHAR2	Specifies the input data
rdata2	RAW	Encrypted output data

Name	Type	Description
scid2	BINARY_INTEGER	Specifies the security co-ordinate ID, which is not used. The value should be set to zero.
dataelement3	VARCHAR3	Specifies the name of data element
cdata3	VARCHAR3	Specifies the input data
rdata3	RAW	Specifies the encrypted output data
scid3	BINARY_INTEGER	Specifies the security co-ordinate ID, which is not used. The value should be set to zero.
dataelement4	VARCHAR2	Specifies the name of data element
cdata4	VARCHAR2	Specifies the input data
rdata4	RAW	Specifies the encrypted output data
scid4	BINARY_INTEGER	Specifies the security co-ordinate ID, which is not used. The value should be set to zero.

Returns

This UDF returns the encrypted value as RAW data.

Exception

If the user does not have protect access rights in the policy, then the procedure terminates with an error message explaining what went wrong.

Example

```
begin
  dbms_output.put_line('Test of UPDATE multi encryption procedure for 4
    COLUMNS');
  dbms_output.put_line('-----------------------------------------------
    -------');
  pty.upd_encryptx4('DE_AES256', 'ASFGFGghg5577fFFyu', raw_out1, 0,
    'DE_AES256', 'IyutGGg76hg8h1', raw_out2, 0, 'DE_AES256', 'AAaazzZZ1199',
    raw_out3, 0 , 'DE_AES256', ' ASFGFGghg5577fFFyu; AblnQEWsw0129NGku;
    BINKUcrc8749lLLnx; CAESYwiw0098mMMns; FEORLkjk2323kKKmn;
    LAENILmcm6677kBBop; MOIRNAzlz9876lMMyu;  MUBMIARAR6087kUUmn;
   NIASAlziz2398hTTuv; PATRHXuru9898hFFns; ROYNESgog7802gMMus;
   SIRSHAuna9049kKKjn; TOTALSlol7843mWWqa; TUSFAVopo8080tTTnx;
   TUHSRAknk8108mKKdw; VAENSAJJBJ6712fFFGH; VEPSIMdsd9898kSDnm;
   URDPLAghg7676LLyu; UNBAKERkik2233lLLmu; YANMRAlsl9090fFFyu;
   YASTURhom0123hHHmn; XAOILDghg0987fFFmn; ZABCDEmom5577bHHyy;
  ZOHRASghg5297nNNcd ', raw_out4, 0);
  DBMS_OUTPUT.PUT_LINE('Encrypted data1: ' || raw_out1);
  DBMS_OUTPUT.PUT_LINE('Encrypted data2: ' || raw_out2);
  DBMS_OUTPUT.PUT_LINE('Encrypted data3: ' || raw_out3);
  DBMS_OUTPUT.PUT_LINE('Encrypted data4: ' || raw_out4);
end;
```

4.5.4 MULTIPLE UPDATE ENCRYPTION PROCEDURES

These procedures updates one to four values of data with one procedure call. The user must be granted *Update* access to use these procedures.

4.5.4.1 pty.encUpdate

This procedure updates and encrypts one value of the *VARCHAR2* data with one data element for encryption.

pty.encUpdate(dataelement VARCHAR2, cdata VARCHAR2, rdata RAW, scid BINARY_INTEGER)

Parameters

Name	Type	Description
dataelement	VARCHAR2	Specifies the name of data element
cdata	VARCHAR2	Specifies the input data
rdata	RAW	Specifies the encrypted output data
scid	BINARY_INTEGER	Specifies the security co-ordinate ID, which is not used. The value should be set to zero.

Returns

This UDF returns the encrypted value as RAW data.

Exception

If the user does not have reprotect access rights in the policy, then the procedure terminates with an error message explaining what went wrong.

Example

```
declare
  raw_out raw(2000);
begin
  dbms_output.put_line('Test of UPDATE multi encryption procedure for 1
    COLUMN');
  dbms_output.put_line('-----------------------------------------------
    ------');
  pty.encUpdate('DE_AES256', 'ASFGFGghg5577fFFyu', raw_out, 0);
  DBMS_OUTPUT.PUT_LINE('Encrypted data: ' || raw_out);
end;
```

4.5.4.2 pty.upd_encryptx2

This procedure updates and encrypts two values of VARCHAR2 data with two data elements for encryption.

pty.upd_encryptx2(dataelement1 VARCHAR2, cdata1 VARCHAR2, rdata1 RAW, scid1 BINARY_INTEGER, dataelement2 VARCHAR2, cdata2 VARCHAR2, rdata2 RAW, scid2 BINARY_INTEGER)

Parameters

Name	Type	Description
dataelement1	VARCHAR2	Specifies the name of data element
cdata1	VARCHAR2	Specifies the input data
rdata1	RAW	Specifies the encrypted output data
scid1	BINARY_INTEGER	Specifies the security co-ordinate ID, which is not used. The value should be set to zero.

Name	Type	Description
dataelement2	VARCHAR2	Specifies the name of data element
cdata2	VARCHAR2	Specifies the input data
rdata2	RAW	Specifies the encrypted output data
scid2	BINARY_INTEGER	Specifies the security co-ordinate ID, which is not used. The value should be set to zero.

Returns

This UDF returns the encrypted value as RAW data.

Exception

If the user does not have reprotect access rights in the policy, then the procedure terminates with an error message explaining what went wrong.

Example

```
begin
  dbms_output.put_line('Test of UPDATE multi encryption procedure for 2
    COLUMNS');
  dbms_output.put_line('-----------------------------------------------
    -------');
  pty.upd_encryptx2('DE_AES256', 'ASFGFGghg5577fFFyu', raw_out1, 0,
    'DE_AES256', 'IyutGGg76hg8h1', raw_out2, 0);
  DBMS_OUTPUT.PUT_LINE('Encrypted data1: ' || raw_out1);
  DBMS_OUTPUT.PUT_LINE('Encrypted data2: ' || raw_out2);
end;
```

4.5.4.3 pty.upd_encryptx3

This procedure updates and encrypts three values of VARCHAR2 data with three data elements for encryption.

pty.upd_encryptx3(dataelement1 VARCHAR2, cdata1 VARCHAR2, rdata1 RAW, scid1 BINARY_INTEGER, dataelement2 VARCHAR2, cdata2 VARCHAR2, rdata2 RAW, scid2 BINARY_INTEGER, dataelement3 VARCHAR2, cdata3 VARCHAR2, rdata3 RAW, scid3 BINARY_INTEGER)

Parameters

Name	Type	Description
dataelement1	VARCHAR2	Specifies the name of data element
cdata1	VARCHAR2	Specifies the input data
rdata1	RAW	Specifies the encrypted output data
scid1	BINARY_INTEGER	Specifies the security co-ordinate ID, which is not used. The value should be set to zero.
dataelement2	VARCHAR2	Specifies the name of data element
cdata2	VARCHAR2	Specifies the input data

Name	Type	Description
rdata2	RAW	Specifies the encrypted output data
scid2	BINARY_INTEGER	Security Coordinate ID, which is not used. The value should be set to zero.
dataelement3	VARCHAR2	Specifies the name of data element
cdata3	VARCHAR2	Specifies the input data
rdata3	RAW	Specifies the encrypted output data
scid3	BINARY_INTEGER	Specifies the security co-ordinate ID, which is not used. The value should be set to zero.

Returns

This UDF returns the encrypted value as RAW data.

Exception

If the user does not have protect access rights in the policy, then the procedure terminates with an error message explaining what went wrong.

Example

```
begin
  dbms_output.put_line('Test of UPDATE multi encryption procedure for 3
    COLUMNS');
  dbms_output.put_line('-----------------------------------------------
    --------');
  pty.upd_encryptx3('DE_AES256', 'ASFGFGghg5577fFFyu', raw_out1, 0,
    'DE_AES256', 'IyutGGg76hg8h1', raw_out2, 0, 'DE_AES256', 'AAaazzZZ1199',
    raw_out3, 0);
  DBMS_OUTPUT.PUT_LINE('Encrypted data1: ' || raw_out1);
  DBMS_OUTPUT.PUT_LINE('Encrypted data2: ' || raw_out2);
  DBMS_OUTPUT.PUT_LINE('Encrypted data3: ' || raw_out3);
end;
```

4.5.4.4 pty.upd_encryptx4

This procedure updates and encrypts four values of VARCHAR2 data with four data elements for encryption.

pty.upd_encryptx4(dataelement1 VARCHAR2, cdata1 VARCHAR2, rdata1 RAW, scid1 BINARY_INTEGER, dataelement2 VARCHAR2, cdata2 VARCHAR2, rdata2 RAW, scid2 BINARY_INTEGER, dataelement3 VARCHAR2, cdata3 VARCHAR2, rdata3 RAW, scid3 BINARY_INTEGER, dataelement4 VARCHAR2, cdata4 VARCHAR2, rdata4 RAW, scid4 BINARY_INTEGER)

Parameters

Name	Type	Description
dataelement1	VARCHAR2	Specifies the name of data element
cdata1	VARCHAR2	Specifies the input data
rdata1	RAW	Specifies the encrypted output data

Name	Type	Description
scid1	BINARY_INTEGER	Specifies the security co-ordinate ID, which is not used. The value should be set to zero.
dataelement2	VARCHAR2	Specifies the name of data element
cdata2	VARCHAR2	Specifies the input data
rdata2	RAW	Specifies the encrypted output data
scid2	BINARY_INTEGER	Specifies the security co-ordinate ID, which is not used. The value should be set to zero.
dataelement3	VARCHAR2	Specifies the name of data element
cdata3	VARCHAR2	Specifies the input data
rdata3	RAW	Encrypted output data
scid3	BINARY_INTEGER	Specifies the security co-ordinate ID, which is not used. The value should be set to zero.
dataelement4	VARCHAR2	Specifies the name of data element
cdata4	VARCHAR2	Specifies the input data
rdata4	RAW	Specifies the encrypted output data
scid4	BINARY_INTEGER	Specifies the security co-ordinate ID, which is not used. The value should be set to zero.

Returns

This UDF returns the encrypted value as RAW data.

Exception

If the user does not have protect access rights in the policy, then the procedure terminates with an error message explaining what went wrong.

Example

```
begin
  dbms_output.put_line('Test of UPDATE multi encryption procedure for 4
    COLUMNS');
  dbms_output.put_line('-----------------------------------------------
    -------');
  pty.upd_encryptx4('DE_AES256', 'ASFGFGghg5577fFFyu', raw_out1, 0,
    'DE_AES256', 'IyutGGg76hg8h1', raw_out2, 0, 'DE_AES256', 'AAaazzZZ1199',
    raw_out3, 0 , 'DE_AES256', ' ASFGFGghg5577fFFyu; AblnQEWsw0129NGku;
    BINKUcrc8749lLLnx; CAESYwiw0098mMMns; FEORLkjk2323kKKmn;
    LAENILmcm6677kBBop; MOIRNAzlz9876lMMyu;  MUBMIARAR6087kUUmn;
   NIASAlziz2398hTTuv; PATRHXuru9898hFFns; ROYNESgog7802gMMus;
   SIRSHAuna9049kKKjn; TOTALSlol7843mWWqa; TUSFAVopo8080tTTnx;
   TUHSRAknk8108mKKdw; VAENSAJJBJ6712fFFGH; VEPSIMdsd9898kSDnm;
   URDPLAghg7676LLyu; UNBAKERkik2233lLLmu; YANMRAlsl9090fFFyu;
   YASTURhom0123hHHmn; XAOILDghg0987fFFmn; ZABCDEmom5577bHHyy;
```

```
  ZOHRASghg5297nNNcd ', raw_out4, 0);
  DBMS_OUTPUT.PUT_LINE('Encrypted data1: ' || raw_out1);
  DBMS_OUTPUT.PUT_LINE('Encrypted data2: ' || raw_out2);
  DBMS_OUTPUT.PUT_LINE('Encrypted data3: ' || raw_out3);
  DBMS_OUTPUT.PUT_LINE('Encrypted data4: ' || raw_out4);
end;
```

4.5.5 INSERT ENCRYPTION UDFs

These UDFs encrypt data. The **Insert** access is required to use these functions.

> **Note:** The permissions for protect, unprotect, and reprotect are defined based on the roles assigned to the user. For more information about how to grant these permissions and assign roles, refer to the *Policy Management Guide*.

4.5.5.1 pty.ins_encrypt

This UDF encrypts data with a data element for encryption.

pty.ins_encrypt(dataelement CHAR, inval CHAR, scid BINARY_INTEGER)

Parameters

Name	Type	Description
dataelement	CHAR	Specifies the name of data element
inval	CHAR	Specifies the input data
scid	BINARY_INTEGER	Specifies the security co-ordinate ID, which is not used. The value should be set to zero.

Returns

This UDF returns the encrypted value as RAW data.

Exception

If the user does not have protect access rights in the policy, then the UDF terminates with an error message explaining what went wrong.

Example

```
select pty.ins_encrypt('DE_AES256', 'Original data', 0) "Test of INSERT encrypt func"
from dual;
```

4.5.5.2 pty.ins_encrypt_char

This UDF encrypts the *CHAR* data with a data element for encryption.

pty.ins_encrypt_char(dataelement CHAR, inval CHAR, scid BINARY_INTEGER)

Parameters

Name	Type	Description
dataelement	CHAR	Specifies the name of data element
inval	CHAR	Specifies the input data

Name	Type	Description
scid	BINARY_INTEGER	Specifies the security co-ordinate ID, which is not used. The value should be set to zero.

Returns

This UDF returns the encrypted value as RAW data.

Exception

If the user does not have protect access rights in the policy, then the UDF terminates with an error message explaining what went wrong.

Example

```
select pty.ins_encrypt_char('DE_AES256', 'Original data', 0) "Test of INSERT enc CHAR
func" from dual;
```

4.5.5.3 pty.ins_encrypt_varchar2

This UDF encrypts the *VARCHAR2* data with a data element for encryption.

pty.ins_encrypt_varchar2(dataelement CHAR, inval VARCHAR2, scid1 BINARY_INTEGER)

Parameters

Name	Type	Description
dataelement	CHAR	Specifies the name of data element
inval	VARCHAR2	Specifies the input data
scid	BINARY_INTEGER	Specifies the security co-ordinate ID, which is not used. The value should be set to zero.

Returns

This UDF returns the encrypted values as the LONG RAW data.

Exception

If the user does not have protect access rights in the policy, then the UDF terminates with an error message explaining what went wrong.

Example

```
select pty.ins_encrypt_varchar2('DE_AES256', 'Original data', 0) "Test INSERT enc
VARCHAR2 func" from dual;
```

4.5.5.4 pty.ins_encrypt_date

This UDF encrypts the *DATE* data with a data element for encryption.

> **Note:** If you want to protect the Oracle input data type **DATE**, then you can use the UDFs as described in *Oracle Input Data Type to UDF Mapping* to identify the appropriate UDF as per your requirement.

pty.ins_encrypt_date(dataelement CHAR, inval DATE, scid BINARY_INTEGER)

Parameters

Name	Type	Description
dataelement	CHAR	Specifies the name of data element
inval	DATE	Specifies the input data
scid	BINARY_INTEGER	Specifies the security co-ordinate ID, which is not used. The value should be set to zero.

Returns

This UDF returns the encrypted values as the RAW data.

Exception

If the user does not have protect access rights in the policy, then the UDF terminates with an error message explaining what went wrong.

Example

```
select pty.ins_encrypt_date('DE_AES256', '23-OCT-14', 0) "Test of INSERT enc DATE func"
from dual;
```

4.5.5.5 pty.ins_encrypt_integer

This UDF encrypts the **INTEGER** data with a data element for encryption.

pty.ins_encrypt_integer(dataelement CHAR, inval INTEGER, scid BINARY_INTEGER)

Parameters

Name	Type	Description
dataelement	CHAR	Specifies the name of data element
inval	INTEGER	Specifies the input data
scid	BINARY_INTEGER	Specifies the security co-ordinate ID, which is not used; value should be set to zero.

Returns

This UDF returns the encrypted values as the RAW data.

Exception

If the user does not have protect access rights in the policy, then the UDF terminates with an error message explaining what went wrong.

Example

```
select PTY.ins_encrypt_integer('DE_AES256', 12345, 0) "Test of INSERT enc INT func"
from dual;
```

4.5.5.6 pty.ins_encrypt_real

This UDF encrypts the **real** data with a data element for encryption.

pty.ins_encrypt_real(dataelement CHAR, inval REAL, scid BINARY_INTEGER)

Parameters

Name	Type	Description
dataelement	CHAR	Specifies the name of data element
inval	REAL	Specifies the input data
scid	BINARY_INTEGER	Specifies the security co-ordinate ID, which is not used. The value should be set to zero.

Returns

This UDF returns the encrypted values as the RAW data.

Exception

If the user does not have protect access rights in the policy, then the UDF terminates with an error message explaining what went wrong.

Example

```
select pty.ins_encrypt_real('DE_AES256', 1234.1234, 0) "Test of INSERT enc REAL func"
from dual;
```

4.5.5.7 pty.ins_encrypt_float

This UDF encrypts the **FLOAT** data with a data element for encryption.

pty.ins_encrypt_float(dataelement CHAR, inval FLOAT, scid BINARY_INTEGER)

Parameters

Name	Type	Description
dataelement	CHAR	Specifies the name of data element
inval	FLOAT	Specifies the input data
scid	BINARY_INTEGER	Specifies the security co-ordinate ID, which is not used. The value should be set to zero.

Returns

This UDF returns the encrypted values as the RAW data.

Exception

If the user does not have protect access rights in the policy, then the UDF terminates with an error message explaining what went wrong.

Example

```
select PTY.ins_encrypt_float('DE_AES256', 1234.1234, 0) "Test of INSERT enc FLOAT func"
from dual;
```

4.5.5.8 pty.ins_encrypt_number

This UDF encrypts the **NUMBER** data with a data element for encryption.

pty.ins_encrypt_number(dataelement CHAR, inval NUMBER, scid BINARY_INTEGER)

Parameters

Name	Type	Description
dataelement	CHAR	Specifies the name of data element
inval	NUMBER	Specifies the input data
scid	BINARY_INTEGER	Specifies the security co-ordinate ID, which is not used. The value should be set to zero.

Returns

This UDF returns the encrypted values as the RAW data.

Exception

If the user does not have protect access rights in the policy, then the UDF terminates with an error message explaining what went wrong.

Example

```
select PTY.ins_encrypt_number('DE_AES256', 12345, 0) "Test of INSERT enc NUMBER func"
from dual;
```

4.5.5.9 pty.ins_encrypt_raw

This UDF encrypts the **RAW** data, which is variable length binary data of maximum size 2000 bytes, with a data element for encryption.

pty.ins_encrypt_raw(dataelement CHAR, inval RAW, scid BINARY_INTEGER)

Parameters

Name	Type	Description
dataelement	CHAR	Specifies the name of data element
inval	RAW	Specifies the input data
scid	BINARY_INTEGER	Specifies the security co-ordinate ID, which is not used. The value should be set to zero.

Returns

This UDF returns the encrypted values as the RAW data.

Exception

If the user does not have protect access rights in the policy, then the UDF terminates with an error message explaining what went wrong.

Example

```
select PTY.ins_encrypt_raw('DE_AES256', 'FFDD12345', 0) "Test of INSERT enc RAW func"
from dual;
```

4.5.6 INSERT NO-ENCRYPTION, TOKEN, FPE AND DTP2 UDFs

These UDFs check user access and create audit logs. The user must have **insert** access to use these functions. Some of these functions are also used for tokenization, Format Preserving Encryption (FPE), and DTP2.

> **Note:**

> Starting from the Version 7.1, Maintenance Release 1 (MR1), the DTP2 protection method is deprecated.
>
> For assistance in switching to a different protection method, contact Protegrity.

4.5.6.1 pty.ins_char

This UDF protects the **CHAR** data with data elements such as tokens, Format Preserving Encryption (FPE) data elements with ASCII as the plaintext encoding, DTP2 and No Encryption for access control.

> **Note:** This UDF supports masking.

> **Note:**
> Starting from the Version 7.1, Maintenance Release 1 (MR1), the DTP2 protection method is deprecated.
>
> For assistance in switching to a different protection method, contact Protegrity.

pty.ins_char(dataelement CHAR, inval CHAR, scid BINARY_INTEGER)

Parameters

Name	Type	Description
dataelement	CHAR	Specifies the name of data element
inval	CHAR	Specifies the input data
scid	BINARY_INTEGER	Specifies the security co-ordinate ID, which is not used. The value should be set to zero.

Returns

This UDF returns the protected value as the CHAR datatype

Exception

If the user does not have protect access rights in the policy, then the UDF terminates with an error message explaining what went wrong.

Example

```
select PTY.ins_char('DE_CHAR', 'Original data', 0) "Test of INSERT CHAR func" from dual;
```

4.5.6.2 pty.ins_varchar2

This UDF protects the **VARCHAR** data with data elements such as tokens, DTP2 and No Encryption for access control.

> **Note:** This UDF supports masking.

> **Note:**
> Starting from the Version 7.1, Maintenance Release 1 (MR1), the DTP2 protection method is deprecated.

For assistance in switching to a different protection method, contact Protegrity.

Caution:

For Date type of data elements, the *pty.ins_varchar2* UDF returns an invalid date format error if the input value falls between the non-existent date range from 05-OCT-1582 to 14-OCT-1582 of the Gregorian Calendar.

For more information about the tokenization and de-tokenization of the cutover dates of the Proleptic Gregorian Calendar, refer to the section *Date Tokenization for cutover Dates of the Proleptic Gregorian Calendar* in the *Protection Method Reference Guide 9.1.0.0.*.

pty.ins_varchar2 (dataelement CHAR, inval VARCHAR2, scid BINARY_INTEGER)

Parameters

Name	Type	Description
dataelement	CHAR	Specifies the name of data element
inval	VARCHAR2	Specifies the input data
scid	BINARY_INTEGER	Specifies the security co-ordinate ID, which is not used. The value should be set to zero.

Returns

This UDF returns the protected value as the VARCHAR2 datatype

Exception

If the user does not have protect access rights in the policy, then the UDF terminates with an error message explaining what went wrong.

Example

```
select PTY.ins_varchar2('DE_VARCHAR2', 'Original data', 0) "Test of INSERT VARCHAR2
func" from dual;
```

4.5.6.3 pty.ins_unicodenvarchar2

This UDF encrypts data with a data element for Format Preserving Encryption (FPE) with any plaintext encoding type.

Note: This UDF does not support masking.

pty.ins_unicodenvarchar2(dataelement CHAR, inval NVARCHAR2, scid BINARY_INTEGER)

Parameters

Name	*Type*	*Description*
dataelement	CHAR	Specifies the name of data element
inval	CHAR	Specifies the input data
scid	BINARY_INTEGER	Specifies the security co-ordinate ID, which is not used. The value should be set to zero.

Returns

This UDF returns the protected value as the NVARCHAR2 datatype

Exception

If the user does not have protect access rights in the policy, then the UDF terminates with an error message that explains what went wrong.

> **Note:**
> Ensure that you use the supported data element only. Using an unsupported data element might result in successful protection without returning any error, but corruption of data can occur.

Example

```
select pty.ins_unicodenvarchar2('fpe_unicode', 'Original data', 0) "Test of INSERT
encrypt func" from dual;
```

4.5.6.4 pty.ins_unicodevarchar2_tok

This UDF protects the **VARCHAR2** data with a Unicode Base64 and Unicode Gen2 data element.

> **Note:** This UDF does not support masking.

pty.ins_unicodevarchar2_tok(dataelement IN CHAR, inval IN VARCHAR2, SCID IN BINARY_INTEGER)

Parameters

Name	*Type*	*Description*
dataelement	CHAR	Specifies the name of data element
inval	VARCHAR2	Specifies the input data
scid	BINARY_INTEGER	Specifies the security co-ordinate ID, which is not used. The value should be set to zero.

Returns

This UDF returns the protected value as the VARCHAR2 datatype.

Exception

If the user does not have protect access rights in the policy, then the UDF terminates with an error message that explains what went wrong.

> **Note:**
> Ensure that you use the supported data element only. Using an unsupported data element might result in successful protection without returning any error, but corruption of data can occur.

Example for Unicode Base64

```
select pty.ins_unicodevarchar2_tok('TE_UNICODE_BASE64_SLT13_ASTYES', 'Protegrity123',0)
from dual;
```

Example for Unicode Gen2

> **Note:** Unicode Gen2 data elements supports newly introduced SLT_X_1 tokenizer from 9.1 version onwards along with existing SLT_1_3 tokenizer. For more information, refer to section *3.4.15 Unicode Gen2* in the *Protection Methods Reference Guide 9.1.0.0*.

```
select
pty.ins_unicodevarchar2_tok('TE_UG2_UTF16LE_LL1AN_SLT13_L2R0_ASTYES',N'xyzÀÁÂÃÄÅÆÇÈÉÊ',0
) from dual;
```

```
select
pty.ins_unicodevarchar2_tok('TE_UG2_SLTX1_L2R2_N_IPA_Greek_Coptic_UTF16LE',N'▒,0) from
dual;
```

4.5.6.5 pty.ins_unicodenvarchar2_tok

This UDF protects the **NVARCHAR2** data with a Unicode Gen2 data element.

> **Note:** This UDF does not support masking.

pty.ins_unicodenvarchar2_tok(dataelement IN CHAR, inval IN NVARCHAR2, SCID IN BINARY_INTEGER)

Parameters

Name	*Type*	*Description*
dataelement	CHAR	Specifies the name of data element
inval	NVARCHAR2	Specifies the input data
scid	BINARY_INTEGER	Specifies the security co-ordinate ID, which is not used. The value should be set to zero.

Returns

This UDF returns the protected value as the NVARCHAR2 data type.

Exception

If the user does not have protect access rights in the policy, then the UDF terminates with an error message that explains what went wrong.

> **Note:**
> Ensure that you use the supported data element only. Using an unsupported data element might result in successful protection without returning any error, but corruption of data can occur.

Example for Unicode Gen2

> **Note:** Unicode Gen2 data elements supports newly introduced SLT_X_1 tokenizer from 9.1 version onwards along with existing SLT_1_3 tokenizer. For more information, refer to section *3.4.15 Unicode Gen2* in the *Protection Methods Reference Guide 9.1.0.0*.

```
select
pty.ins_unicodenvarchar2_tok('TE_UG2_UTF16LE_LL1AN_SLT13_L2R0_ASTYES',N'xyzÀÁÂÃÄÅÆÇÈÉÊ',
0) from dual;
```

```
select
pty.ins_unicodenvarchar2_tok('TE_UG2_SLTX1_L2R2_N_IPA_Greek_Coptic_UTF16LE',N'▒,0) from
dual;
```

4.5.6.6 pty.ins_date

This UDF protects the **DATE** data with a data element such as tokens, Format Preserving Encryption (FPE) data elements with ASCII as the plaintext encoding, DTP2 and No Encryption for access control.

> **Note:** The DATE UDFs can be used for tokenization if the data element date format and the NLS_DATE_FORMAT environment variable for an Oracle session is the same.
>
> For example, if you define a data element with MM-DD-YYYY date type, the data will be tokenized only if the NLS_DATE_FORMAT environment variable for an Oracle session is also set to MM-DD-YYYY date type.
>
> You can use the following query to change the date type.
>
> ```
> ALTER SESSION SET NLS_DATE_FORMAT = 'MM-DD-YYYY';
> ```

pty.ins_date(dataelement CHAR, inval DATE, scid BINARY_INTEGER)

Parameters

Name	Type	Description
dataelement	CHAR	Specifies the name of data element
inval	DATE	Specifies the input data
scid	BINARY_INTEGER	Specifies the security co-ordinate ID, which is not used. The value should be set to zero.

Returns

This UDF returns the unprotected DATE value, when No Encryption data element is used.

This UDF returns the protected DATE value, when tokenization data element is used and if the data element date format and the **NLS_DATE_FORMAT** environment variable for an Oracle session is the same as mentioned in the note above.

Exception

No Encryption Date Element: If the user does not have protect access rights in the policy, then the UDF terminates with an error message explaining what went wrong.

Tokenization Date Element: Tokenization fails and the UDF terminates with an error message explaining what went wrong.

Example: No Encryption

```
select PTY.ins_date('DE_NoEnc', '10-23-2014', 0) "Test of INSERT DATE func" from dual;
```

Example: Tokenization

```
select PTY.ins_date('DE_DATE', '10-23-2014', 0) "Test of INSERT DATE func" from dual;
```

4.5.6.7 pty.ins_integer

This UDF protects the **INTEGER** data with data elements such as tokens and No Encryption for access control.

pty.ins_integer(dataelement CHAR, inval INTEGER, scid BINARY_INTEGER)

Parameters

Name	Type	Description
dataelement	CHAR	Specifies the name of data element
inval	INTEGER	Specifies the input data
scid	BINARY_INTEGER	Specifies the security co-ordinate ID, which is not used. The value should be set to zero.

Returns

This UDF returns the protected value as the INTEGER datatype.

Exception

If the user does not have protect access rights in the policy, then the UDF terminates with an error message explaining what went wrong.

Example

```
select PTY.ins_integer('DE_Integer', 12345, 0) "Test of INSERT INT func" from dual;
```

4.5.6.8 pty.ins_real

This UDF protects the **REAL** data with a data element for No Encryption for access control.

pty.ins_real(dataelement CHAR, inval REAL, scid BINARY_INTEGER)

Parameters

Name	Type	Description
dataelement	CHAR	Specifies the name of data element
inval	REAL	Specifies the input data
scid	BINARY_INTEGER	Specifies the security co-ordinate ID, which is not used. The value should be set to zero.

Returns

This UDF returns the unprotected value as the REAL datatype.

Exception

If the user does not have protect access rights in the policy, then the UDF terminates with an error message explaining what went wrong.

> **Note:**
> Ensure that you use the supported data element only. If an unsupported data element is passed, the following error is returned: *character to number conversion error.*

Example

```
select PTY.ins_real('DE_NoEnc', 1234.1234, 0) "Test of INSERT REAL func" from dual;
```

4.5.6.9 pty.ins_float

This UDF protects the **FLOAT** data with a data element such as No Encryption for access control.

pty.ins_float(dataelement CHAR, inval FLOAT, scid BINARY_INTEGER)

Parameters

Name	Type	Description
dataelement	CHAR	Specifies the name of data element
inval	FLOAT	Specifies the input data
scid	BINARY_INTEGER	Specifies the security co-ordinate ID, which is not used. The value should be set to zero.

Returns

This UDF returns the unprotected value as the FLOAT datatype.

Exception

If the user does not have protect access rights in the policy, then the UDF terminates with an error message explaining what went wrong.

> **Note:**
> Ensure that you use the supported data element only. If an unsupported data element is passed, the following error is returned: *character to number conversion error.*

Example

```
select PTY.ins_float('DE_NoEnc', 1234.1234, 0) "Test of INSERT FLOAT func" from dual;
```

4.5.6.10 pty.ins_number

This UDF protects the **NUMBER** data with data element such as tokens and No Encryption for access control.

pty.ins_number(dataelement CHAR, inval NUMBER, scid BINARY_INTEGER)

Parameters

Name	Type	Description
dataelement	CHAR	Specifies the name of data element

Name	Type	Description
inval	NUMBER	Specifies the input data
scid	BINARY_INTEGER	Specifies the security co-ordinate ID, which is not used. The value should be set to zero.

Returns

This UDF returns the unprotected value as the NUMBER datatype.

Exception

If the user does not have protect access rights in the policy, then the UDF terminates with an error message explaining what went wrong.

> **Note:**
> Ensure that you use the supported data element only. If an unsupported data element is passed, the following error is returned: *character to number conversion error*.

Example

```
select PTY.ins_number('DE_Integer', 12345, 0) "Test of INSERT NUMBER func" from dual;
```

4.5.6.11 pty.ins_raw

This UDF protects the **RAW** data with a data element for No Encryption for access control.

pty.ins_raw(dataelement CHAR, inval RAW, scid BINARY_INTEGER)

Parameters

Name	Type	Description
dataelement	CHAR	Specifies the name of data element
inval	RAW	Specifies the input data
scid	BINARY_INTEGER	Specifies the security co-ordinate ID, which is not used. The value should be set to zero.

Returns

This UDF returns the unprotected value as RAW data.

Exception

If the user does not have protect access rights in the policy, then the UDF terminates with an error message explaining what went wrong.

> **Note:**
> Ensure that you use the supported data element only. If an unsupported data element is passed, the following error is returned: *character to number conversion error*.

Example

```
select PTY.ins_raw('DE_NoEnc', 'FFDD12345', 0) "Test of INSERT RAW func" from dual;
```

4.5.7 UPDATE ENCRYPTION UDFs

These UDFs encrypt data by first unprotecting it with the older data element, and then protecting the updated value with the new data element. **Update** access is required to use these functions.

> **Note:**
> Starting from the Version 7.1, Maintenance Release 1 (MR1), the DTP2 protection method is deprecated.
>
> For assistance in switching to a different protection method, contact Protegrity.

4.5.7.1 pty.encUpdate

This procedure updates and encrypts one value of the *VARCHAR2* data with one data element for encryption.

pty.encUpdate(dataelement VARCHAR2, cdata VARCHAR2, rdata RAW, scid BINARY_INTEGER)

Parameters

Name	Type	Description
dataelement	VARCHAR2	Specifies the name of data element
cdata	VARCHAR2	Specifies the input data
rdata	RAW	Specifies the encrypted output data
scid	BINARY_INTEGER	Specifies the security co-ordinate ID, which is not used. The value should be set to zero.

Returns

This UDF returns the encrypted value as RAW data.

Exception

If the user does not have reprotect access rights in the policy, then the procedure terminates with an error message explaining what went wrong.

Example

```
declare
  raw_out raw(2000);
begin
  dbms_output.put_line('Test of UPDATE multi encryption procedure for 1
    COLUMN');
  dbms_output.put_line('-----------------------------------------------
    ------');
  pty.encUpdate('DE_AES256', 'ASFGFGghg5577fFFyu', raw_out, 0);
  DBMS_OUTPUT.PUT_LINE('Encrypted data: ' || raw_out);
end;
```

4.5.7.2 pty.upd_encrypt_char

This UDF re-encrypts the **CHAR** protected data that has been updated, with a data element for encryption.

pty.upd_encrypt_char(dataelement CHAR, inval CHAR, scid BINARY_INTEGER)

Parameters

Name	Type	Description
dataelement	CHAR	Specifies the name of data element
inval	CHAR	Specifies the input data
scid	BINARY_INTEGER	Specifies the security co-ordinate ID, which is not used. The value should be set to zero.

Returns

This UDF returns the encrypted value as RAW data.

Exception

If the user does not have reprotect access rights in the policy, then the UDF terminates with an error message explaining what went wrong.

Example

```
select PTY.upd_encrypt_char('DE_AES256', 'Original data', 0) "Test of UPDATE enc CHAR
func" from dual;
```

4.5.7.3 pty.upd_encrypt_varchar2

This UDF re-encrypts the **VARCHAR2** data that has been updated, with a data element for encryption.

pty.upd_encrypt_varchar2(dataelement CHAR, inval VARCHAR2, scid BINARY_INTEGER)

Parameters

Name	Type	Description
dataelement	CHAR	Specifies the name of data element
inval	VARCHAR2	Specifies the input data
scid	BINARY_INTEGER	Specifies the security co-ordinate ID, which is not used. The value should be set to zero.

Returns

This UDF returns the encrypted value as RAW data.

Exception

If the user does not have reprotect access rights in the policy, then the UDF terminates with an error message explaining what went wrong.

Example

```
select PTY.upd_encrypt_varchar2('DE_AES256', 'Original data', 0) "Test of UPDATE enc
VARCHAR2 func" from dual;
```

4.5.7.4 pty.upd_encrypt_date

This UDF re-encrypts the **DATE** data that has been updated, with a data element for encryption.

> **Note:** When you use the **`pty.ins_encrypt_date`** UDF to protect date, the data is not protected. If you want to protect the Oracle input data type **`DATE`**, you must use the UDFs as described in *Oracle Input Data Type to UDF Mapping* to identify the appropriate UDF as per your requirement.

pty.upd_encrypt_date(dataelement CHAR, inval DATE, scid BINARY_INTEGER)

Parameters

Name	Type	Description
dataelement	CHAR	Specifies the name of data element
inval	DATE	Specifies the input data
scid	BINARY_INTEGER	Specifies the security co-ordinate ID, which is not used. The value should be set to zero.

Returns

This UDF returns the encrypted value as RAW data.

Exception

If the user does not have reprotect access rights in the policy, then the UDF terminates with an error message explaining what went wrong.

Example

```
select PTY.upd_encrypt_date('DE_AES256', '23-OCT-14', 0) "Test of UPDATE enc DATE func"
from dual;
```

4.5.7.5 pty.upd_encrypt_integer

This UDF re-encrypts the **INTEGER** data that has been updated, with a data element for encryption.

pty.upd_encrypt_integer(dataelement CHAR, inval INTEGER, scid BINARY_INTEGER)

Parameters

Name	Type	Description
dataelement	CHAR	Specifies the name of data element
inval	INTEGER	Specifies the input data
scid	BINARY_INTEGER	Specifies the security co-ordinate ID, which is not used. The value should be set to zero.

Returns

This UDF returns the encrypted value as RAW data.

Exception

If the user does not have reprotect access rights in the policy, then the UDF terminates with an error message explaining what went wrong.

Example

```
select PTY.upd_encrypt_integer('DE_AES256', 12345, 0) "Test of UPDATE enc INT func"
from dual;
```

4.5.7.6 pty.upd_encrypt_real

This UDF re-encrypts the **REAL** data that has been updated, with a data element for encryption.

pty.upd_encrypt_real(dataelement CHAR, inval REAL, scid BINARY_INTEGER)

Parameters

Name	Type	Description
dataelement	CHAR	Specifies the name of data element
inval	REAL	Specifies the input data
scid	BINARY_INTEGER	Specifies the security co-ordinate ID, which is not used. The value should be set to zero.

Returns

This UDF returns the encrypted value as RAW data.

Exception

If the user does not have reprotect access rights in the policy, then the UDF terminates with an error message explaining what went wrong.

Example

```
select PTY.upd_encrypt_real('DE_AES256', 1234.1234, 0) "Test of UPDATE enc REAL func"
from dual;
```

4.5.7.7 pty.upd_encrypt_float

This UDF re-encrypts the **FLOAT** data that has been updated, with a data element for encryption.

pty.upd_encrypt_float(dataelement CHAR, inval FLOAT, scid BINARY_INTEGER)

Parameters

Name	Type	Description
dataelement	CHAR	Specifies the name of data element
inval	FLOAT	Specifies the input data
scid	BINARY_INTEGER	Specifies the security co-ordinate ID, which is not used. The value should be set to zero.

Returns

This UDF returns the encrypted value as RAW data.

Exception

If the user does not have reprotect access rights in the policy, then the UDF terminates with an error message explaining what went wrong.

Example

```
select PTY.upd_encrypt_float('DE_AES256', 1234.1234, 0) "Test of UPDATE enc FLOAT func"
from dual;
```

4.5.7.8 pty.upd_encrypt_number

This UDF re-encrypts the **NUMBER** data that has been updated, with a data element in encryption.

pty.upd_encrypt_number(dataelement CHAR, inval NUMBER, scid BINARY_INTEGER)

Parameters

Name	Type	Description
dataelement	CHAR	Specifies the name of data element
inval	NUMBER	Specifies the input data
scid	BINARY_INTEGER	Specifies the security co-ordinate ID, which is not used. The value should be set to zero.

Returns

This UDF returns the encrypted value as RAW data.

Exception

If the user does not have reprotect access rights in the policy, then the UDF terminates with an error message explaining what went wrong.

Example

```
select PTY.upd_encrypt_number('DE_AES256', 12345, 0) "Test of UPDATE enc NUMBER func"
from dual;
```

4.5.7.9 pty.upd_encrypt_raw

This UDF re-encrypts the **RAW** data that has been updated, with a data element for encryption.

pty.upd_encrypt_RAW(dataelement CHAR, inval RAW, scid BINARY_INTEGER)

Parameters

Name	Type	Description
dataelement	CHAR	Specifies the name of data element
inval	RAW	Specifies the input data
scid	BINARY_INTEGER	Specifies the security co-ordinate ID, which is not used. The value should be set to zero.

Returns

This UDF returns the encrypted value as RAW data.

Exception

If the user does not have reprotect access rights in the policy, then the UDF terminates with an error message explaining what went wrong.

Example

```
select PTY.upd_encrypt_raw('DE_AES256', 'FFDD12345', 0) "Test of UPDATE enc RAW func"
from dual;
```

4.5.8 UPDATE NO-ENCRYPTION, TOKEN, FPE, AND DTP2 UDFs

These UDFs checks user access and gets audit logs while updating data using tokenization, Format Preserving Encryption (FPE), and DTP2. **Update** access is required to use these procedures.

> **Note:**
> Starting from the Version 7.1, Maintenance Release 1 (MR1), the DTP2 protection method is deprecated.
>
> For assistance in switching to a different protection method, contact Protegrity.

> **Note:** For reprotect operations, the Audit logs are generated as Protect Logs instead of Reprotect Logs.

4.5.8.1 pty.upd_char

This UDF re-protects the **CHAR** data with data elements such as tokens, DTP2, and No Encryption for access control.

> **Note:**
> Starting from the Version 7.1, Maintenance Release 1 (MR1), the DTP2 protection method is deprecated.
>
> For assistance in switching to a different protection method, contact Protegrity.

pty.upd_char(dataelement CHAR, inval CHAR, scid BINARY_INTEGER)

Parameters

Name	Type	Description
dataelement	CHAR	Specifies the name of data element
inval	CHAR	Specifies the input data
scid	BINARY_INTEGER	Specifies the security co-ordinate ID, which is not used. The value should be set to zero.

Returns

This UDF returns the output value as the CHAR datatype.

Exception

If the user does not have reprotect access rights in the policy, then the UDF terminates with an error message explaining what went wrong.

Example

```
select PTY.upd_char('DE_DTP2_AES256_AN', 'Original data', 0) "Test of UPDATE CHAR func" from
dual;
```

4.5.8.2 pty.upd_varchar2

This UDF reprotects the **VARCHAR2** data with data elements such as tokens, DTP2, and No Encryption for access control.

> **Note:**
> Starting from the Version 7.1, Maintenance Release 1 (MR1), the DTP2 protection method is deprecated.
>
> For assistance in switching to a different protection method, contact Protegrity.

pty.upd_varchar2(dataelement CHAR, inval VARCHAR2, scid BINARY_INTEGER)

Parameters

Name	Type	Description
dataelement	CHAR	Specifies the name of data element
inval	VARCHAR2	Specifies the input data
scid	BINARY_INTEGER	Specifies the security co-ordinate ID, which is not used. The value should be set to zero.

Returns

This UDF returns the output value as the VARCHAR2 datatype.

Exception

If the user does not have reprotect access rights in the policy, then the UDF terminates with an error message explaining what went wrong.

Example

```
select PTY.upd_varchar2('DE_DTP2_AES256_AN', 'Original data', 0) "Test of UPDATE VARCHAR2
func" from dual;
```

4.5.8.3 pty.upd_unicodenvarchar2

This UDF re-encrypts the **NVARCHAR2** data that has been updated, with a data element for FPE Unicode.

pty.upd_unicodenvarchar2(dataelement CHAR, inval NVARCHAR2, scid BINARY_INTEGER)

Parameters

Name	Type	Description
dataelement	CHAR	Specifies the name of data element
inval	NVARCHAR2	Specifies the input data
scid	BINARY_INTEGER	Specifies the security co-ordinate ID, which is not used. The value should be set to zero.

Returns

This UDF returns the encrypted value as the NVARCHAR2 data.

Exception

If the user does not have reprotect access rights in the policy, then the UDF terminates with an error message explaining what went wrong.

> **Note:**
> Ensure that you use the supported data element only. Using an unsupported data element might result in successful reprotection without returning any error, but corruption of data can occur.

Example

```
select PTY.upd_unicodenvarchar2('fpe_unicode', 'Original data', 0) "Test of UPDATE encrypt
NVARCHAR2 func" from dual;
```

4.5.8.4 pty.upd_unicodevarchar2_tok

This UDF re-encrypts **VARCHAR2** data that has been updated with a Unicode Base64 and Unicode Gen2 data element.

upd_unicodevarchar2_tok(dataelement IN CHAR, inval IN VARCHAR2, SCID IN BINARY_INTEGER)

Parameters

Name	Type	Description
dataelement	CHAR	Name of data element
inval	VARCHAR2	Input data
scid	BINARY_INTEGER	Security Coordinate ID, which is not used. The value should be set to zero.

Returns

Encrypted value as VARCHAR2 data.

Exception

If the user does not have reprotect access rights in the policy, then the UDF terminates with an error message explaining what went wrong.

> **Note:**
> Ensure that you use the supported data element only. Using an unsupported data element might result in successful reprotection without returning any error, but corruption of data can occur.

Example

```
select pty.upd_unicodevarchar2_tok('TE_UG2_S13_PL_N_BASCYR_AN_UTF8','▒▒▒',0) from dual;
```

4.5.8.5 pty.upd_unicodenvarchar2_tok

This UDF re-encrypts **NVARCHAR2** data that has been updated with a Unicode Base64 and Unicode Gen2 data element.

pty.upd_unicodenvarchar2_tok(dataelement IN CHAR, inval IN NVARCHAR2, SCID IN BINARY_INTEGER)

Parameters

Name	Type	Description
dataelement	CHAR	Name of data element
inval	NVARCHAR2	Input data
scid	BINARY_INTEGER	Security Coordinate ID, which is not used. The value should be set to zero.

Returns

Encrypted value as NVARCHAR2 data.

Exception

If the user does not have reprotect access rights in the policy, then the UDF terminates with an error message explaining what went wrong.

> **Note:**
> Ensure that you use the supported data element only. Using an unsupported data element might result in successful reprotection without returning any error, but corruption of data can occur.

Example

```
select pty.upd_unicodenvarchar2_tok('TE_UG2_S13_PL_N_BASCYR_AN_UTF8','▒▒▒',0) from dual;
```

4.5.8.6 pty.upd_date

This UDF reprotects the **DATE** data with a data element for No Encryption to do access control.

> **Note:** When you use the **pty.ins_encrypt_date** UDF to protect date, the data is not protected. If you want to protect the Oracle input data type **DATE**, you must use the UDFs as described in *Oracle Input Data Type to UDF Mapping* to identify the appropriate UDF as per your requirement.

> **Note:** The DATE UDFs can be used for tokenization if the data element date format and the NLS_DATE_FORMAT environment variable for an Oracle session is the same.
>
> For example, if you define a data element with MM-DD-YYYY date type, the data will be tokenized only if the NLS_DATE_FORMAT environment variable for an Oracle session is also set to MM-DD-YYYY date type.
>
> You can use the following query to change the date type.
>
> ```
> ALTER SESSION SET NLS_DATE_FORMAT = 'MM-DD-YYYY';
> ```

pty.upd_date(dataelement CHAR, inval DATE, scid BINARY_INTEGER)

Parameters

Name	Type	Description
dataelement	CHAR	Specifies the name of data element
inval	DATE	Specifies the input data
scid	BINARY_INTEGER	Specifies the security co-ordinate ID, which is not used. The value should be set to zero.

Returns

Original value as DATE.

Exception

If the user does not have reprotect access rights in the policy, then the UDF terminates with an error message explaining what went wrong.

Example

```
select PTY.upd_date('DE_NoEnc', '23-OCT-14', 0) "Test of UPDATE DATE func" from dual;
```

4.5.8.7 pty.upd_integer

This UDF re-protects the **INTEGER** data with data elements such as tokens and No Encryption for access control.

pty.upd_integer(dataelement CHAR, inval INTEGER, scid BINARY_INTEGER)

Parameters

Name	Type	Description
dataelement	CHAR	Specifies the name of data element
inval	INTEGER	Specifies the input data
scid	BINARY_INTEGER	Specifies the security co-ordinate ID, which is not used. The value should be set to zero.

Returns

This UDF returns the original value as the INTEGER datatype.

Exception

If the user does not have reprotect access rights in the policy, then the UDF terminates with an error message explaining what went wrong.

Example

```
select PTY.upd_integer('DE_Integer', 12345, 0) "Test of UPDATE INT func" from dual;
```

4.5.8.8 pty.upd_real

This UDF reprotects the **REAL** data with a data element for No Encryption to do access control.

pty.upd_real(dataelement CHAR, inval REAL, scid BINARY_INTEGER)

Parameters

Name	Type	Description
dataelement	CHAR	Specifies the name of data element
inval	REAL	Specifies the input data
scid	BINARY_INTEGER	Specifies the security co-ordinate ID, which is not used. The value should be set to zero.

Returns

This UDF returns the origianal value as the REAL datatype.

Exception

If the user does not have reprotect access rights in the policy, then the UDF terminates with an error message explaining what went wrong.

> **Note:**
> Ensure that you use the supported data element only. If an unsupported data element is passed, the following error is returned:
> *`character to number conversion error.`*

Example

```
select PTY.upd_real('DE_NoEnc', 1234.1234, 0) "Test of UPDATE REAL func" from dual;
```

4.5.8.9 pty.upd_float

This UDF reprotects the **FLOAT** data with a data element for No Encryption to do access control.

pty.upd_float(dataelement CHAR, inval FLOAT, scid BINARY_INTEGER)

Parameters

Name	Type	Description
dataelement	CHAR	Specifies the name of data element
inval	FLOAT	Specifies the input data
scid	BINARY_INTEGER	Specifies the security co-ordinate ID, which is not used. The value should be set to zero.

Returns

This UDF returns the original value as the FLOAT datatype.

Exception

If the user does not have reprotect access rights in the policy, then the UDF terminates with an error message explaining what went wrong.

> **Note:**
> Ensure that you use the supported data element only. If an unsupported data element is passed, the following error is returned:
> *`character to number conversion error.`*

Example

```
select PTY.upd_float('DE_NoEnc', 1234.1234, 0) "Test of UPDATE FLOAT func" from dual;
```

4.5.8.10 pty.upd_number

This UDF reprotects the **NUMBER** data with data elements such as tokens and No Encryption for access control.

pty.upd_number(dataelement CHAR, inval NUMBER, scid BINARY_INTEGER)

Parameters

Name	Type	Description
dataelement	CHAR	Specifies the name of data element
inval	NUMBER	Specifies the input data
scid	BINARY_INTEGER	Specifies the security co-ordinate ID, which is not used. The value should be set to zero.

Returns

This UDF returns the original value as the NUMBER datatype.

Exception

If the user does not have reprotect access rights in the policy, then the UDF terminates with an error message explaining what went wrong.

> **Note:**
> Ensure that you use the supported data element only. If an unsupported data element is passed, the following error is returned:
> `character to number conversion error`.

Example

```
select PTY.upd_number('DE_Integer', 12345, 0) "Test of UPDATE NUMBER func" from dual;
```

4.5.8.11 pty.upd_raw

This UDF re-protects the **RAW** data with a data element for No Encryption to do access control.

pty.upd_raw(dataelement CHAR, inval RAW, scid BINARY_INTEGER)

Parameters

Name	Type	Description
dataelement	CHAR	Specifies the name of data element
inval	RAW	Specifies the input data

Name	Type	Description
scid	BINARY_INTEGER	Specifies the security co-ordinate ID, which is not used. The value should be set to zero.

Returns

This UDF returns the original value as the RAW data.

Exception

If the user does not have reprotect access rights in the policy, then the UDF terminates with an error message explaining what went wrong.

> **Note:**
> Ensure that you use the supported data element only. If an unsupported data element is passed, the following error is returned:
> *character to number conversion error.*

Example

```
select PTY.upd_raw('DE_NoEnc', 'FFDD12345', 0) "Test of UPDATE RAW func" from dual;
```

4.5.9 SELECT DECRYPTION UDFs

These UDFs unprotects tokenized data items with every function call. **Select** access is required to use these procedures.

4.5.9.1 pty.sel_decrypt

This UDF decrypts the RAW data with a data element for encryption.

pty.sel_decrypt(dataelement CHAR, inval RAW, scid BINARY_INTEGER)

Parameters

Name	Type	Description
dataelement	CHAR	Specifies the name of data element
inval	RAW	Specifies the input data
scid	BINARY_INTEGER	Specifies the security co-ordinate ID, which is not used. The value should be set to zero.

Returns

This UDF returns the unprotected value as the CHAR2 datatype.

This UDF returns the unprotected value as the NULL, when the user has no access to data in the policy.

Exception

If configured in policy and user does not have access, then the UDF terminates with an error message explaining what went wrong.

Example

```
select PTY.sel_decrypt('DE_AES256', PTY.ins_encrypt('DE_AES256', 'Original data', 0),0)
"Test of SELECT dec func" from dual;
```

4.5.9.2 pty.sel_decrypt_char

This UDF decrypts the *CHAR* data with a data element for encryption.

pty.sel_decrypt_char(dataelement CHAR, inval RAW, scid BINARY_INTEGER)

Parameters

Name	Type	Description
dataelement	CHAR	Specifies the name of data element
inval	RAW	Specifies the input data
scid	BINARY_INTEGER	Specifies the security co-ordinate ID, which is not used. The value should be set to zero.

Returns

This UDF returns the unprotected value as the CHAR2 datatype.

This UDF returns the unprotected value as the NULL, when the user has no access to data in the policy.

Exception

If configured in policy and user does not have access, then the UDF terminates with an error message explaining what went wrong.

Example

```
select PTY.sel_decrypt_char('AES256', PTY.ins_encrypt_char('AES256', 'Original data',
0),0) "Test of SELECT dec CHAR func" from dual;
```

4.5.9.3 pty.sel_decrypt_varchar2

This UDF decrypts the *VARCHAR2* data with a data element for encryption.

pty.sel_decrypt_varchar2(dataelement CHAR, inval RAW, scid BINARY_INTEGER)

Parameters

Name	Type	Description
dataelement	CHAR	Specifies the name of data element
inval	RAW	Specifies the input data
scid	BINARY_INTEGER	Specifies the security co-ordinate ID, which is not used. The value should be set to zero.

Returns

This UDF returns the unprotected value as the CHAR2 datatype.

This UDF returns the unprotected value as the NULL, when the user has no access to data in the policy.

Exception

If configured in policy and user does not have access, then the UDF terminates with an error message explaining what went wrong.

Example

```
select PTY.sel_decrypt_varchar2('AES256', PTY.ins_encrypt_varchar2('AES256','Original
data', 0),0) "Test of SELECT dec VARCHAR2 func" from dual;
```

4.5.9.4 pty.sel_decrypt_date

This UDF decrypts the *DATE* data with a data element for encryption.

pty.sel_decrypt_date(dataelement CHAR, inval RAW, scid BINARY_INTEGER)

Parameters

Name	Type	Description
dataelement	CHAR	Specifies the name of data element
inval	RAW	Specifies the input data
scid	BINARY_INTEGER	Specifies the security co-ordinate ID, which is not used. The value should be set to zero.

Returns

Unprotected value as DATE. This UDF returns the unprotected value as the DATE datatype.

This UDF returns the unprotected value as the NULL, when the user has no access to data in the policy.

Exception

If configured in policy and user does not have access, then the UDF terminates with an error message explaining what went wrong.

Example

```
select PTY.sel_decrypt_date('DE_AES256', PTY.ins_encrypt_date('DE_AES256', '23-OCT-14',
0),0) "Test of SELECT dec DATE func" from dual;
```

4.5.9.5 pty.sel_decrypt_integer

This UDF decrypts the *INTEGER* data with a data element for encryption.

pty.sel_decrypt_integer(dataelement CHAR, inval RAW, scid BINARY_INTEGER)

Parameters

Name	Type	Description
dataelement	CHAR	Specifies the name of data element
inval	RAW	Specifies the input data
scid	BINARY_INTEGER	Specifies the security co-ordinate ID, which is not used. The value should be set to zero.

Returns

Unprotected value as INTEGER. This UDF returns the unprotected value as the INTEGER datatype.

This UDF returns the unprotected value as the NULL, when the user has no access to data in the policy.

Exception

If configured in policy and user does not have access, then the UDF terminates with an error message explaining what went wrong.

Example

```
select PTY.sel_decrypt_integer('DE_AES256', PTY.ins_encrypt_integer('DE_AES256', 12345,
0),0) "Test of SELECT dec INT func" from dual;
```

4.5.9.6 pty.sel_decrypt_real

This UDF decrypts the *REAL* data with a data element for encryption.

pty.sel_decrypt_real(dataelement CHAR, inval RAW, scid BINARY_INTEGER)

Parameters

Name	Type	Description
dataelement	CHAR	Specifies the name of data element
inval	RAW	Specifies the input data
scid	BINARY_INTEGER	Specifies the security co-ordinate ID, which is not used. The value should be set to zero.

Returns

Unprotected value as REAL.This UDF returns the unprotected value as the REAL datatype.

This UDF returns the unprotected value as the NULL, when the user has no access to data in the policy.

Exception

If configured in policy and user does not have access, then the UDF terminates with an error message explaining what went wrong.

Example

```
select PTY.sel_decrypt_real('AES256', PTY.ins_encrypt_real('AES256',1234.1234,0),0)
“Test of SELECT dec REAL func” from dual;
```

4.5.9.7 pty.sel_decrypt_float

This UDF decrypts the *FLOAT* data with a data element for encryption.

pty.sel_decrypt_float(dataelement CHAR, inval RAW, scid BINARY_INTEGER)

Parameters

Name	Type	Description
dataelement	CHAR	Specifies the name of data element
inval	RAW	Specifies the input data
scid	BINARY_INTEGER	Specifies the security co-ordinate ID, which is not used. The value should be set to zero.

Returns

This UDF returns the unprotected value as the FLOAT datatype.

This UDF returns the unprotected value as the NULL, when the user has no access to data in the policy.

Exception

If configured in policy and user does not have access, then the UDF terminates with an error message explaining what went wrong.

Example

```
select PTY.sel_decrypt_float('DE_AES256', PTY.ins_encrypt_float('DE_AES256', 1234.1234,
0),0) "Test of SELECT dec FLOAT func" from dual;
```

4.5.9.8 pty.sel_decrypt_number

This UDF decrypts the *NUMBER* data with a data element for encryption.

pty.sel_decrypt_number(dataelement CHAR, inval RAW, scid BINARY_INTEGER)

Parameters

Name	Type	Description
dataelement	CHAR	Specifies the name of data element
inval	RAW	Specifies the input data
scid	BINARY_INTEGER	Specifies the security co-ordinate ID, which is not used. The value should be set to zero.

Returns

This UDF returns the unprotected value as the NUMBER datatype.

This UDF returns the unprotected value as the NULL, when the user has no access to data in the policy.

Exception

If configured in policy and user does not have access, then the UDF terminates with an error message explaining what went wrong.

Example

```
select PTY.sel_decrypt_number('DE_AES256', PTY.ins_encrypt_number('DE_AES256', 12345,
0),0) "Test of SELECT dec NUMBER func" from dual;
```

4.5.9.9 pty.sel_decrypt_raw

This UDF decrypts the RAW data with a data element for encryption.

pty.sel_decrypt_raw(dataelement CHAR, inval RAW, scid BINARY_INTEGER)

Parameters

Name	Type	Description
dataelement	CHAR	Specifies the name of data element
inval	RAW	Specifies the input data
scid	BINARY_INTEGER	Specifies the security co-ordinate ID, which is not used. The value should be set to zero.

Returns

This UDF returns the unprotected value as the RAW data.

This UDF returns the unprotected value as the NULL, when the user has no access to data in the policy.

Exception

If configured in policy and user does not have access, then the UDF terminates with an error message explaining what went wrong.

Example

```
select PTY.sel_decrypt_raw('AES256', PTY.ins_encrypt_raw('AES256', 'FFDD12345', 0),0)
"Test of SELECT dec RAW func" from dual;
```

4.5.10 SELECT NO-ENCRYPTION, TOKEN, FPE AND DTP2 UDFs

These UDFs checks user access and gets audit logs while unprotecting data with Tokenization, DTP2 and Access Control. The **Select** access is needed to use these procedures.

> **Note:**
> Starting from the Version 7.1, Maintenance Release 1 (MR1), the DTP2 protection method is deprecated.
>
> For assistance in switching to a different protection method, contact Protegrity.

4.5.10.1 pty.sel_char

This UDF unprotects the *CHAR* data with data element such as tokens, Format Preserving Encryption (FPE) data elements with ASCII as the plaintext encoding, DTP2, and No Encryption for access control.

> **Note:** This UDF supports masking.

> **Note:**
> Starting from the Version 7.1, Maintenance Release 1 (MR1), the DTP2 protection method is deprecated.
>
> For assistance in switching to a different protection method, contact Protegrity.

pty.sel_char(dataelement CHAR, inval CHAR, scid BINARY_INTEGER)

Parameters

Name	Type	Description
dataelement	CHAR	Specifies the name of data element
inval	CHAR	Specifies the input data
scid	BINARY_INTEGER	Specifies the security co-ordinate ID, which is not used. The value should be set to zero.

Returns

This UDF returns the unprotected value as the CHAR datatype.

This UDF returns the protected value, if this option is configured in the policy and user does not have access to data.

This UDF returns the unprotected value as NULL, when the user has no access to data in the policy.

This UDF returns the unprotected value as NULL, when the user is not specified in the policy.

Exception

If configured in policy and user does not have unprotect access rights, then the UDF terminates with an error message explaining what went wrong.

Example

```
select PTY.sel_char('DE_DTP2_AES256_AN', PTY.ins_char('DE_DTP2_AES256_AN', 'Original
data', 0),0) "Test of SELECT CHAR func" from dual;
```

4.5.10.2 pty.sel_varchar2

This UDF unprotects the *VARCHAR2* data with data elements such as tokens, Format Preserving Encryption (FPE) data elements with ASCII as the plaintext encoding, DTP2, and No Encryption for access control.

> **Note:** This UDF supports masking.

> **Note:**
> Starting from the Version 7.1, Maintenance Release 1 (MR1), the DTP2 protection method is deprecated.
>
> For assistance in switching to a different protection method, contact Protegrity.

pty.sel_varchar2(dataelement CHAR, inval VARCHAR2, scid BINARY_INTEGER)

Parameters

Name	Type	Description
dataelement	CHAR	Specifies the name of data element
inval	VARCHAR2	Specifies the input data
scid	BINARY_INTEGER	Specifies the security co-ordinate ID, which is not used. The value should be set to zero.

Returns

This UDF returns the unprotected value as the VARCHAR2 datatype.

This UDF returns the protected value, if this option is configured in the policy and user does not have access to data.

This UDF returns the unprotected value as NULL, when the user has no access to data in the policy.

This UDF returns the unprotected value as NULL, when the user is not specified in the policy.

Exception

If configured in policy and user does not have unprotect access rights, then the UDF terminates with an error message explaining what went wrong.

Example

```
select PTY.sel_varchar2('DE_DTP2_AES256_AN', PTY.ins_varchar2('DE_DTP2_AES256_AN',
'Original data', 0),0) "Test of SELECT VARCHAR2 func" from dual;
```

4.5.10.3 pty.sel_unicodenvarchar2

This UDF unprotects the *NVARCHAR* data protected by a Format Preserving Encryption (FPE) data element with any plaintext encoding type.

> **Note:** This UDF does not support masking.

pty.sel_unicodenvarchar2(dataelement CHAR, inval NVARCHAR2, scid BINARY_INTEGER)

Parameters

Name	*Type*	*Description*
dataelement	CHAR	Specifies the name of data element
inval	NVARCHAR2	Specifies the input data
scid	BINARY_INTEGER	Specifies the security co-ordinate ID, which is not used. The value should be set to zero.

Returns

This UDF returns the unprotected value as the NVARCHAR2 datatype.

This UDF returns the protected value, if this option is configured in the policy and user does not have access to data.

This UDF returns the unprotected value as NULL, when the user has no access to data in the policy.

This UDF returns the unprotected value as NULL, when the user is not specified in the policy.

Exception

If configured in policy and user does not have unprotect access rights, then the UDF terminates with an error message explaining what went wrong.

> **Note:**
> Ensure that you use the supported data element only. Using an unsupported data element might result in successful unprotection without returning any error, but corruption of data can occur.

Example

```
select pty.sel_unicodenvarchar2('fpe_unicode', PTY.ins_unicodenvarchar2('fpe_unicode',
'Original data', 0),0) "Test of SELECT NVARCHAR2 func" from dual;
```

4.5.10.4 pty.sel_unicodevarchar2_tok

This UDF unprotects the **VARCHAR2** data protected by a Unicode Base64 and Unicode Gen2 data element.

> **Note:** This UDF does not support masking.

pty.sel_unicodevarchar2_tok(dataelement IN CHAR, inval IN VARCHAR2, SCID IN BINARY_INTEGER)

Parameters

Name	*Type*	*Description*
dataelement	CHAR	Specifies the name of data element
inval	VARCHAR2	Specifies the input data
scid	BINARY_INTEGER	Specifies the security co-ordinate ID, which is not used. The value should be set to zero.

Returns

This UDF returns the unprotected value as VARCHAR2.

Exception

If configured in policy and user does not have unprotect access rights, then the UDF terminates with an error message explaining what went wrong.

> **Note:**
> Ensure that you use the supported data element only. Using an unsupported data element might result in successful unprotection without returning any error, but corruption of data can occur.

Example for Unicode Base64

```
select pty.sel_unicodevarchar2_tok('TE_UNICODE_BASE64_SLT13_ASTYES',
pty.ins_unicodevarchar2_tok('TE_UNICODE_BASE64_SLT13_ASTYES', 'Protegrity123',0),0)
from dual;
```

Example for Unicode Gen2

> **Note:** Unicode Gen2 data elements supports newly introduced SLT_X_1 tokenizer from 9.1 version onwards along with existing SLT_1_3 tokenizer. For more information, refer to section *3.4.15 Unicode Gen2* in the *Protection Methods Reference Guide 9.1.0.0*.

```
select
pty.sel_unicodevarchar2_tok('TE_UG2_UTF16LE_LL1AN_SLT13_L2R0_ASTYES',pty.ins_unicodevarc
har2_tok('TE_UG2_UTF16LE_LL1AN_SLT13_L2R0_ASTYES',N'xyzÀÁÂÃÄÅÆÇÈÉÊ',0),0) from dual;
```

```
select
pty.sel_unicodevarchar2_tok('TE_UG2_SLTX1_L2R2_N_IPA_Greek_Coptic_UTF16LE',pty.ins_unico
devarchar2_tok('TE_UG2_SLTX1_L2R2_N_IPA_Greek_Coptic_UTF16LE',N'▒,0),0) from dual;
```

4.5.10.5 pty.sel_unicodenvarchar2_tok

This UDF unprotects **NVARCHAR2** data protected by a Unicode Gen2 data element.

> **Note:** This UDF does not support masking.

pty.sel_unicodenvarchar2_tok(dataelement IN CHAR, inval IN NVARCHAR2, SCID IN BINARY_INTEGER)

Parameters

Name	*Type*	*Description*
dataelement	CHAR	Specifies the name of data element
inval	NVARCHAR2	Specifies the input data
scid	BINARY_INTEGER	Specifies the security co-ordinate ID, which is not used. The value should be set to zero.

Returns

This UDF returns the unprotected value as NVARCHAR2.

Exception

If configured in policy and user does not have unprotect access rights, then the UDF terminates with an error message explaining what went wrong.

> **Note:**
> Ensure that you use the supported data element only. Using an unsupported data element might result in successful unprotection without returning any error, but corruption of data can occur.

Example for Unicode Gen2

> **Note:** Unicode Gen2 data elements supports newly introduced SLT_X_1 tokenizer from 9.1 version onwards along with existing SLT_1_3 tokenizer. For more information, refer to section *3.4.15 Unicode Gen2* in the *Protection Methods Reference Guide 9.1.0.0*.

```
select
pty.sel_unicodenvarchar2_tok('TE_UG2_UTF16LE_LL1AN_SLT13_L2R0_ASTYES',pty.ins_unicodenva
rchar2_tok('TE_UG2_UTF16LE_LL1AN_SLT13_L2R0_ASTYES',N'xyzÀÁÂÃÄÅÆÇÈÉÊ',0),0) from dual;
```

```
select
pty.sel_unicodenvarchar2_tok('TE_UG2_SLTX1_L2R2_N_IPA_Greek_Coptic_UTF16LE',pty.ins_unic
odenvarchar2_tok('TE_UG2_SLTX1_L2R2_N_IPA_Greek_Coptic_UTF16LE',N'▒',0),0) from dual;
```

4.5.10.6 pty.sel_date

This UDF unprotects the *DATE* data with a data element for No Encryption to do access control.

> **Note:** The DATE UDFs can be used for tokenization if the data element date format and the NLS_DATE_FORMAT environment variable for an Oracle session is the same.
>
> For example, if you define a data element with MM-DD-YYYY date type, the data will be tokenized only if the NLS_DATE_FORMAT environment variable for an Oracle session is also set to MM-DD-YYYY date type.
>
> You can use the following query to change the date type.
>
> ```
> ALTER SESSION SET NLS_DATE_FORMAT = 'MM-DD-YYYY';
> ```

pty.sel_date(dataelement CHAR, inval DATE, scid BINARY_INTEGER)

Parameters

Name	Type	Description
dataelement	CHAR	Specifies the name of data element
inval	DATE	Specifies the input data
scid	BINARY_INTEGER	Specifies the security co-ordinate ID, which is not used. The value should be set to zero.

Returns

This UDF returns the unprotected value as the DATE datatype.

This UDF returns the unprotected value as NULL, when the user has no access to data in the policy.

This UDF returns the unprotected value as NULL, when the user is not specified in the policy.

Exception

If configured in policy and user does not have unprotect access rights, then the UDF terminates with an error message explaining what went wrong.

Example

```
select PTY.sel_date('DE_NoEnc', PTY.ins_date('DE_NoEnc', '23-OCT-14', 0),0) "Test of
SELECT DATE func" from dual;
```

4.5.10.7 pty.sel_integer

This UDF unprotects the *INTEGER* data with data elements such as tokens and No Encryption for access control.

pty.sel_integer(dataelement CHAR, inval INTEGER, scid BINARY_INTEGER)

Parameters

Name	Type	Description
dataelement	CHAR	Specifies the name of data element
inval	INTEGER	Specifies the input data
scid	BINARY_INTEGER	Specifies the security co-ordinate ID, which is not used. The value should be set to zero.

Returns

This UDF returns the unprotected value as the INTEGER datatype.

This UDF returns the protected value, if this option is configured in the policy and user does not have access to data.

This UDF returns the unprotected value as NULL, when the user has no access to data in the policy.

This UDF returns the unprotected value as NULL, when the user is not specified in the policy.

Exception

If configured in policy and user does not have unprotect access rights, then the UDF terminates with an error message explaining what went wrong.

Example

```
select PTY.sel_integer('Integer4',PTY.ins_integer('integer',12344567,0),0) “Test of
SELECT INT func” from dual;
```

4.5.10.8 pty.sel_real

This UDF unprotects the *REAL* data with a data element for No Encryption for access control.

pty.sel_real(dataelement CHAR, inval REAL, scid BINARY_INTEGER)

Parameters

Name	Type	Description
dataelement	CHAR	Specifies the name of data element
inval	REAL	Specifies the input data
scid	BINARY_INTEGER	Specifies the security co-ordinate ID, which is not used. The value should be set to zero.

Returns

This UDF returns the unprotected value as the REAL datatype.

This UDF returns the unprotected value as NULL, when the user has no access to data in the policy.

This UDF returns the unprotected value as NULL, when the user is not specified in the policy.

Exception

If configured in policy and user does not have unprotect access rights, then the UDF terminates with an error message explaining what went wrong.

> **Note:**
> Ensure that you use the supported data element only. If an unsupported data element is passed, the following error is returned: *`character to number conversion error`*.

Example

```
select PTY.sel_real('DE_NoEnc', PTY.ins_real('DE_NoEnc', 1234.1234, 0),0) "Test of
SELECT REAL func" from dual;
```

4.5.10.9 pty.sel_float

This UDF unprotects the *FLOAT* data with a data element for No Encryption for access control.

pty.sel_float(dataelement CHAR, inval FLOAT, scid BINARY_INTEGER)

Parameters

Name	Type	Description
dataelement	CHAR	Specifies the name of data element
inval	FLOAT	Specifies the input data

Name	Type	Description
scid	BINARY_INTEGER	Specifies the security co-ordinate ID, which is not used. The value should be set to zero.

Returns

This UDF returns the unprotected value as the FLOAT datatype.

This UDF returns the unprotected value as NULL, when the user has no access to data in the policy.

This UDF returns the unprotected value as NULL, when the user is not specified in the policy.

Exception

If configured in policy and user does not have unprotect access rights, then the UDF terminates with an error message explaining what went wrong.

> **Note:**
> Ensure that you use the supported data element only. If an unsupported data element is passed, the following error is returned:
> *`character to number conversion error.`*

Example

```
select PTY.sel_float('DE_NoEnc', PTY.ins_float('DE_NoEnc', 1234.1234, 0),0) "Test of
SELECT FLOAT func" from dual;
```

4.5.10.10 pty.sel_number

This UDF unprotects the *NUMBER* data with data elements such as tokens and No Encryption for access control.

pty.sel_number(dataelement CHAR, inval NUMBER, scid BINARY_INTEGER)

Parameters

Name	Type	Description
dataelement	CHAR	Specifies the name of data element
inval	NUMBER	Specifies the input data
scid	BINARY_INTEGER	Specifies the security co-ordinate ID, which is not used. The value should be set to zero.

Returns

This UDF returns the unprotected value as the NUMBER datatype.

This UDF returns the protected value, if this option is configured in the policy and user does not have access to data.

This UDF returns the unprotected value as NULL, when the user has no access to data in the policy.

This UDF returns the unprotected value as NULL, when the user is not specified in the policy.

Exception

If configured in policy and user does not have unprotect access rights, then the UDF terminates with an error message explaining what went wrong.

> **Note:**
> Ensure that you use the supported data element only. If an unsupported data element is passed, the following error is returned: *`character to number conversion error.`*

Example

```
select PTY.sel_number('DE_Integer', PTY.ins_number('DE_Integer', 123455667, 0),0) "Test
of SELECT NUMBER func" from dual;
```

4.5.10.11 pty.sel_raw

This UDF unprotects the *RAW* data with a data element for No Encryption for access control.

pty.sel_raw(dataelement CHAR, inval RAW, scid BINARY_INTEGER)

Parameters

Name	Type	Description
dataelement	CHAR	Specifies the name of data element
inval	RAW	Specifies the input data
scid	BINARY_INTEGER	Specifies the security co-ordinate ID, which is not used. The value should be set to zero.

Returns

This UDF returns the unprotected value as the RAW data.

This UDF returns the unprotected value as NULL, when the user has no access to data in the policy.

This UDF returns the unprotected value as NULL, when the user is not specified in the policy.

Exception

If configured in policy and user does not have unprotect access rights, then the UDF terminates with an error message explaining what went wrong.

> **Note:**
> Ensure that you use the supported data element only. If an unsupported data element is passed, the following error is returned: *`character to number conversion error.`*

Example

```
select PTY.sel_raw('DE_NoEnc', PTY.ins_raw('DE_NoEnc', 'FFDD12345', 0),0) "Test of
SELECT RAW func" from dual;
```

4.5.11 HASH UDFs

These UDFs protect the data as a hash value.

4.5.11.1 pty.ins_hash_varchar2

This UDF uses the hash function to protect the *VARCHAR* data with a data element for hashing to return a protected value.

pty.ins_hash_varchar2(dataelement CHAR, cdata VARCHAR2, scid BINARY_INTEGER)

Parameters

Name	Type	Description
dataelement	CHAR	Specifies the name of data element
cdata	VARCHAR2	Specifies the input data
scid	BINARY_INTEGER	Specifies the security co-ordinate ID, which is not used. The value should be set to zero.

Returns

This UDF returns the Hash value as the RAW data.

This UDF returns the unprotected value as NULL, when the user has no access to data in the policy.

Exception

If configured in policy and user does not have unprotect access rights, then the UDF terminates with an error message explaining what went wrong.

Example

```
SELECT PTY.ins_hash_varchar2('DE_Hash', ' ASertcv2013; CUxdcs3675; ccNNddfF9084;
hjMjCS0123',0) "Test of INSERT HASH function" from dual;
```

4.5.11.2 pty.upd_hash_varchar2

This UDF uses the hash function to protect VARCHAR data with a data element for hashing to return a protected value.

pty.ins_hash_varchar2(dataelement CHAR, inval VARCHAR2, scid BINARY_INTEGER)

Parameters

Name	Type	Description
dataelement	CHAR	Specifies the name of data element
cdata	VARCHAR2	Specifies the input data
scid	BINARY_INTEGER	Specifies the security co-ordinate ID, which is not used. The value should be set to zero.

Returns

This UDF returns the Hash value as the RAW data.

This UDF returns the unprotected value as NULL, when the user has no access to data in the policy.

Exception

If configured in policy and user does not have unprotect access rights, then the UDF terminates with an error message explaining what went wrong.

Example

```
SELECT PTY.upd_hash_varchar2('DE_Hash', 'ASertcv2013; CUxdcs3675; ccNNddfF9084;
hjMjCS0123;',0) "Test of UPDATE HASH function" from dual;
```

4.5.12 BLOB UDFs

These UDFs can be used to encrypt and decrypt the data stored in the BLOB data type.

4.5.12.1 pty.ins_encrypt_blob

This function is used to encrypt the data stored in a *BLOB* with an encryption data element.

pty.ins_encrypt_blob(dataelement CHAR, input_data BLOB , scid INTEGER)

Parameters

Name	Type	Description
dataelement	CHAR	Specifies the name of the data element
input_data	BLOB	Specifies the input data for the UDF
scid	INTEGER	Specifies the security co-ordinate ID, which is not used. The value should be set to zero.

Returns

This UDF returns the encrypted value as the BLOB data.

> **Note:**
> If you perform a protect operation with the input data as null or empty, then the output will be an *empty_blob*.

Exception

If the user does not have *protect* privileges in the policy, then the UDF terminates with an error message explaining what went wrong.

Example

```
select pty.ins_encrypt_blob('AES256',TO_BLOB('691F89CD2BCBF055EFD4F3B51470AEF6'),0)
from dual;
```

Note

> **Caution:**
> - The *pty.ins_encrypt_blob* UDF supports protection for a maximum of 1.5 GB of input data.
> - The *pty.ins_encrypt_blob* UDF will return an unexpected behaviour if you exceed the maximum input data limit of 1.5 GB. For example: *ORA-28579: network error during callback from external procedure agent*.

4.5.12.2 pty.sel_decrypt_blob

This function is used to decrypt the encrypted data stored in a BLOB with an encryption data element.

pty.sel_decrypt_blob(dataelement CHAR, input_data BLOB, scid INTEGER)

Parameters

Name	Type	Description
dataelement	CHAR	Specifies the name of the data element
input_data	BLOB	Specifies the input data for the UDF
scid	INTEGER	Specifies the security co-ordinate ID, which is not used. The value should be set to zero.

Returns

This UDF returns the decrypted value as the BLOB data.

This UDF returns the decrypted value as an EMPTY_BLOB, when the user has no access to the database.

> **Note:**
> If you perform a unprotect operation with the input data as null or empty, then the output will be an *empty_blob*.

Exception

If the user does not have *unprotect* access rights in the policy, then the UDF terminates with an error message explaining what went wrong.

Example

```
select
pty.sel_decrypt_blob('AES256',pty.ins_encrypt_blob('AES256',TO_BLOB('691F89CD2BCBF055EFD
4F3B51470AEF6'),0),0) from dual;
```

4.5.13 CLOB UDFs

These UDFs can be used to encrypt and decrypt the data stored in the CLOB data type.

4.5.13.1 pty.ins_encrypt_clob

This function is used to encrypt the data stored in a *CLOB* with an encryption data element.

pty.ins_encrypt_clob(dataelement CHAR, input_data CLOB, scid INTEGER)

> **Caution:**
> Ensure that the input data stored in the CLOB data type does not contain multibyte characters. If you pass data containing multibyte characters to the CLOB UDF, then an unexpected behaviour is observed.
>
> For example: An error *'ORA-28579: network error during callback from external procedure agent'* is returned or the input data is corrupted.
>
> For more information about CLOB data type, refer to the *Oracle Help Center.*

Parameters

Name	Type	Description
dataelement	CHAR	Specifies the name of the data element

Name	Type	Description
input_data	CLOB	Specifies the input data for the UDF
scid	INTEGER	Specifies the security co-ordinate ID, which is not used. The value should be set to zero.

Returns

This UDF returns the encrypted value as the BLOB data.

> **Note:**
> If you perform a protect operation with the input data as null or empty, then the output will be an *empty_blob*.

Exception

If the user does not have *protect* access rights in the policy, then the UDF terminates with an error message explaining what went wrong.

Example

```
select pty.ins_encrypt_clob('AES256','John',0) from dual;
```

Note

> **Caution:**
> - The *pty.ins_encrypt_clob* UDF supports protection for a maximum of 500 MB of input data.
> - The *pty.ins_encrypt_clob* UDF will return an unexpected behaviour if you exceed the maximum input data limit of 500 MB. For example: *ORA-28579: network error during callback from external procedure agent*.

4.5.13.2 pty.sel_decrypt_clob

This function is used to decrypt the encrypted data stored in a BLOB with an encryption data element.

pty.sel_decrypt_clob(dataelement CHAR, input_data BLOB, scid INTEGER)

Parameters

Name	Type	Description
dataelement	CHAR	Specifies the name of the data element
input_data	BLOB	Specifies the input data for the UDF
scid	INTEGER	Specifies the security co-ordinate ID, which is not used. The value should be set to zero.

Returns

This UDF returns the decrypted value as the CLOB data.

This UDF returns the decrypted value as an EMPTY_CLOB, when the user has no access to the database.

> **Note:**

> If you perform a unprotect operation with the input data as null or empty, then the output will be an *empty_clob*.

Exception

If the user does not have *unprotect* access rights in the policy, then the UDF terminates with an error message explaining what went wrong.

Example

```
select pty.sel_decrypt_clob('AES256',pty.ins_encrypt_clob('AES256','John',0),0) from
dual;
```

4.5.14 Appendix A: Oracle Input Datatype to UDF Mapping

This section provide tables with the Oracle input data type to the appropriate UDF mapping. It also provides the data element information that you must consider when creating a policy.

Table 4-1: Oracle Input Datatype to UDF Mapping for Insert operation to Update operation

Oracle UDF - Insert	Oracle UDF - Update	Oracle Input Type	Output Type	Data Element Type
pty.ins_char	pty.upd_char	CHAR	CHAR	No Encryption
pty.ins_encrypt_char/ pty.ins_encrypt	pty.upd_encrypt_char/ pty.upd_encrypt	CHAR	RAW	3DES, AES-128, AES-256
pty.ins_encrypt	pty.upd_encrypt	CHAR	RAW	CUSP 3DES, CUSP AES 128, CUSP AES 156
pty.ins_char	pty.upd_char	CHAR	CHAR	TOKENS-Numeric(0-9)
pty.ins_char	pty.upd_char	CHAR	CHAR	TOKENS-Alpha(a-z,A-Z)
pty.ins_char	pty.upd_char	CHAR	CHAR	TOKENS-Uppercase Alpha(A-Z)
pty.ins_char	pty.upd_char	CHAR	CHAR	TOKENS-Alpha(a-z,A-Z)
pty.ins_char	pty.upd_char	CHAR	CHAR	TOKENS-Alpha-Numeric (0-9,a-z,A-Z)
pty.ins_char	pty.upd_char	CHAR	CHAR	TOKENS-Uppercase Alpha-Numeric(0-9,A-Z)
pty.ins_char	pty.upd_char	CHAR	CHAR	TOKENS-Printable
pty.ins_char	pty.upd_char	CHAR	CHAR	TOKENS-Credit card(0-9)
pty.ins_char	pty.upd_char	CHAR	CHAR	TOKENS-Lower ASCII (lower part of ASCII table)
pty.ins_char	pty.upd_char	CHAR	CHAR	TOKENS-Email

pty.ins_varchar2	pty.ins_varchar2	VARCHAR2	VARCHAR2	No Encryption
pty.ins_encrypt_varchar2	pty.upd_encrypt_varchar2	VARCHAR2	RAW	3DES, AES-128, AES-256
pty.ins_encrypt_varchar2	pty.upd_encrypt_varchar2	VARCHAR2	RAW	CUSP 3DES, CUSP AES 128, CUSP AES 156
pty.ins_varchar2	pty.upd_varchar2	VARCHAR2	VARCHAR2	TOKENS-Numeric(0-9)
pty.ins_varchar2	pty.upd_varchar2	VARCHAR2	VARCHAR2	TOKENS-Alpha(a-z,A-Z)
pty.ins_varchar2	pty.upd_varchar2	VARCHAR2	VARCHAR2	TOKENS-Uppercase Alpha(A-Z)
pty.ins_varchar2	pty.upd_varchar2	VARCHAR2	VARCHAR2	TOKENS-Alpha(a-z,A-Z)
pty.ins_varchar2	pty.upd_varchar2	VARCHAR2	VARCHAR2	TOKENS-Alpha-Numeric (0-9,a-z,A-Z)
pty.ins_varchar2	pty.upd_varchar2	VARCHAR2	VARCHAR2	TOKENS-Uppercase Alpha-Numeric(0-9,A-Z)
pty.ins_varchar2	pty.upd_varchar2	VARCHAR2	VARCHAR2	TOKENS-Printable
pty.ins_varchar2	pty.upd_varchar2	VARCHAR2	VARCHAR2	TOKENS-Credit card(0-9)
pty.ins_varchar2	pty.upd_varchar2	VARCHAR2	VARCHAR2	TOKENS-Lower ASCII (lower part of ASCII table)
pty.ins_varchar2	pty.upd_varchar2	VARCHAR2	VARCHAR2	TOKENS-Email
pty.ins_date	pty.upd_date	DATE	DATE	No Encryption
pty.ins_encrypt_date	pty.upd_encrypt_date	DATE	RAW	Encryption-AES-256
pty.ins_varchar2	pty.upd_varchar2	DATE	DATE	TOKENS-Date(YYYY-MM-DD)
pty.ins_varchar2	pty.upd_varchar2	DATE	DATE	TOKENS-Date(DD/MM/YYYY)
pty.ins_varchar2	pty.upd_varchar2	DATE	DATE	TOKENS-Date(MM/DD/YYYY)
pty.ins_varchar2	pty.upd_varchar2	DATE	DATE	TOKENS-Datetime(YYYY-MM-DD HH:MM:SS MMM)

pty.ins_integer	pty.upd_integer	INTEGER	INTEGER	No Encryption
pty.ins_encrypt_integer	pty.upd_encrypt_integer	INTEGER	RAW	Encryption-AES-256
pty.ins_integer	pty.upd_integer	INTEGER	INTEGER	TOKENS-INTEGER
pty.ins_number	pty.upd_number	NUMBER	NUMBER	No Encryption
pty.ins_encrypt_number	pty.upd_encrypt_number	NUMBER	RAW	Encryption-AES-256
pty.ins_number	pty.upd_number	NUMBER	NUMBER	TOKENS-Decimal (numeric with decimal point and sign)
pty.ins_real	pty.upd_real	REAL	REAL	No Encryption
pty.ins_encrypt_real	pty.upd_encrypt_real	REAL	RAW	Encryption-AES-256
pty.ins_float	pty.upd_float	FLOAT	FLOAT	No Encryption
pty.ins_encrypt_float	pty.upd_encrypt_float	FLOAT	RAW	Encryption-AES-256
pty.ins_raw	pty.upd_raw	RAW	RAW	No Encryption
pty.ins_encrypt_raw	pty.upd_encrypt_raw	RAW	RAW	Encryption-AES-256
		BINARY		Tokenization is not supported for BINARY for ORACLE
		UNICODE		Tokenization is not supported for UNICODE for ORACLE

Table 4-2: Oracle Input Datatype to UDF Mapping for Insert operation to Select operation

Oracle UDF - Insert	Oracle UDF - Select	Oracle Input Type	Output Type	Data Element Type
pty.ins_encrypt_blob	pty.sel_decrypt_blob	BLOB	BLOB	3DES, AES-128, AES-256
pty.ins_encrypt_clob	pty.sel_decrypt_clob	CLOB	CLOB	3DES, AES-128, AES-256

4.6 Teradata User Defined Functions

This section provides a detailed list of UDFs (User Defined Functions) for general information, and protection and unprotection of data with different data types. Run the sample queries in BTEQ (Basic Teradata Query). For more information, refer to the sample scripts provided in the default location, `/opt/protegrity/defiance_dps/pep/sqlscripts/teradata`.

> **Note:** Protegrity UDFs can support the JSON format for protection and unprotection starting from Teradata 15.10.

> It is not possible to mask data stored in XML or JSON (JavaScript Object Notation) formats. While executing the Unprotect UDFs for these formats, clear data is returned along with an error message. Masking is supported only with Varchar and Char UDFs.

4.6.1 Teradata UDFs for Protection and Tokenization

This section provides a detailed list of UDFs (User Defined Functions) for general information, and protection, unprotection, and tokenization of data with different data types.

Teradata UDFs - Deterministic and Non-deterministic clauses

Teradata supports following two optional clauses to categorize if the UDF returns identical results for identical inputs or not:

- DETERMINISTIC - This clause specifies that the UDF function returns same results for identical inputs. The de-tokenization and decryption UDFs are defined with DETERMINISTIC clause.
- NOT DETERMINISTIC - This clause specifies that the UDF function returns non-identical results for identical inputs. This is the default option. The tokenization and encryption UDFs are defined with NOT DETERMINISTIC clause.

Risk

In case of a query with constant arguments to the DETERMINISTIC UDF call, Teradata may cache the result of evaluated UDF, as designed. During subsequent query execution, the results may be fetched from the Teradata internal cache without evaluating the UDF.

This is a risk as it can cause unauthorized access to the protected data due to lack of authorization check during the UDF execution. In addition to this, altering the clause to NOT DETERMINISTIC may cause performance issues as UDFs defined with DETERMINISTIC clause execute faster in comparison to UDFs defined with the NOT DETERMINISTIC clause.

Mitigation

As per your usage, if you are not using any constants in the UDF call, then you can re-create the UDF with DETERMINISTIC clause to ensure faster performance.

4.6.1.1 General UDFs

This section includes list of general UDFs that can be used to retrieve the Teradata Protector version and the current user.

4.6.1.1.1 pty_whoami

This UDF returns the name of the user who is currently logged in. The user should have no-policy access rights to run this UDF.

pty_whoami()

Parameters

None

Returns

The name of user logged in to the database.

Example

```
select pty_whoami();
```

4.6.1.1.2 pty_getversion

This UDF returns the version of the installed Teradata Database Protector. The user should have no-policy access rights to run this UDF.

pty_getversion()

Parameters

None

Returns

The version of the product as a string.

Example

```
select pty_getversion();
```

4.6.1.1.3 pty_getdbsinfo

This UDF returns the Teradata session, statement, and request numbers. These parameters are captured in audit logs and can be cross-referenced in ESA Forensics View. The user should have no-policy access rights to run this UDF.

pty_getdbsinfo()

Parameters

None

Returns

The following parameters in a string.

Name	Type	Description
session	STRING	Specifies the Teradata session number.
request	STRING	Specifies the Teradata request number.
statement	STRING	Specifies the Teradata statement identifier.

Example

```
select pty_getdbsinfo()
```

4.6.1.1.4 pty_status

This function checks if the policy and policy contents are consistent across all AMPs (Access Module Processors) or nodes in Teradata. The status is returned in a table with one row for each node.

pty_status(communicationid INTEGER)

Parameters

Name	Type	Description
communicationid	INTEGER	Specifies where the UDF will find the policy. The value must be the same as configured in the *pepserver.cfg* file.

Returns

One row for each AMP. Every row contains the following fields:

Name	Type	Description
username	VARCHAR	Specifies the name of the user logged on to the database.
hdrhash	VARBYTE	Specifies the checksum of the policy header for the policy stored on the node.
numdataelements	INTEGER	Specifies the number of data elements in the policy stored on the node.
numusers	INTEGER	Specifies the number of users in the policy stored on the node.
nodeid	INTEGER	Specifies the Teradata node ID.
ampid	INTEGER	Specifies the Teradata AMP ID.

Name	Type	Description
status	INTEGER	Specifies the status code. 0=Failure, 1=Success In case of failure there will be an error text in the MESSAGE column.
message	VARCHAR	Specifies the message content.
version	VARCHAR	Specifies the product version as a string.

Example

```
select * FROM TABLE (PTY_STATUS(0)) AS PEP_STATUS;
```

4.6.1.1.5 pty_getcurrentkeyid

This UDF returns the current key ID for a data element and is typically used together with *PTY_GETKEYID* to determine if some data is protected with the most recent key for a given data element. The user should have access rights for protection.

pty_getcurrentkeyid(dataelement VARCHAR, communicationid INTEGER)

Parameters

Name	Type	Description
dataelement	VARCHAR	Specifies the name of the data element.
communicationid	INTEGER	Specifies where the UDF will find the policy. The value must be the same as configured in the *pepserver.cfg* file.

Returns

Current key ID as INTEGER.

Exception

If the data element is missing in the policy or if the data element does not contain a key id, then the UDF terminates with an error message explaining what went wrong.

Example

```
select PTY_GETCURRENTKEYID('AES256_IV_CRC_KID', 0);
```

4.6.1.1.6 pty_getkeyid

This UDF returns the key ID that was used to protect an item of data. It is typically used together with *PTY_GETCURRENTKEYID* to determine if some data is protected with the most recent key for a given data element.

pty_getkeyid(dataelement VARCHAR, data VARBYTE, communicationid INTEGER)

Parameters

Name	Type	Description
dataelement	VARCHAR	Specifies the name of the data element.
data	VARBYTE	Specifies the data that has been protected with encryption and is using key ID.
communicationid	INTEGER	Specifies the location where the UDF will find the policy. The value must be the same as configured in the *pepserver.cfg* file.

Returns

Current key ID as INTEGER.

Exception

If the data element is missing in the policy or if the data element does not contain a key id, then the UDF terminates with an error message explaining what went wrong.

Example

```
select PTY_GETKEYID('AES256_IV_CRC_KID', PTY_VARCHARLATINENC('ProtegrityProt',
'AES256_IV_CRC_KID', 100,0,0), 0);
```

4.6.1.2 Access Check UDFs

This section includes list of UDFs that can be used to check access related information.

4.6.1.2.1 pty_checkselaccess

This UDF checks if a user has unprotect access for a set of data elements. To run this UDF, the user should be granted access rights for protection.

pty_checkselaccess(dataelement<n> VARCHAR, resultlen INTEGER, communicationid INTEGER)

Parameters

Name	Type	Description
dataelement1	VARCHAR	Specifies the name of the data element to check.
dataelement2	VARCHAR	Specifies the name of the data element to check.
dataelement3	VARCHAR	Specifies the name of the data element to check.
resultlen	INTEGER	Specifies the length of the buffer to hold the result.
communicationid	INTEGER	Specifies the location where the UDF will find the policy. The value must be the same as configured in the *pepserver.cfg* file.

Returns

A 3-CHARACTER string.

- Position 1: Value *1* indicates select permission on *dataelement1*, value *0* indicates no select permission
- Position 2: Value *1* indicates select permission on *dataelement2*, value *0* indicates no select permission
- Position 3: Value *1* indicates select permission on *dataelement3*, value *0* indicates no select permission

Exception

None

Example

```
select PTY_CHECKSELACCESS('AES256', 'AES256', 'AES_IV_CRC_KID', 3, 0);
```

4.6.1.3 VARCHAR LATIN UDFs

The varchar Latin UDFs accept string data encoded in the Latin character set.

4.6.1.3.1 pty_varcharlatinenc

This UDF protects the string data with a data element for encryption.

pty_varcharlatinenc(col VARCHAR, dataelement VARCHAR, resultlen INTEGER, communicationid INTEGER, scid INTEGER)

Parameters

Name	Type	Description
col	VARCHAR	Specifies the data to protect.
dataeement	VARCHAR	Specifies the name of the data element.

Name	Type	Description
resultlen	INTEGER	Specifies the length of the buffer to hold the result.
communicationid	INTEGER	Specifies the location where the UDF will find the policy. The value must be the same as configured in the *pepserver.cfg* file.
scid	INTEGER	Specifies the security coordinate ID. The parameter is no longer used. The value of the parameter must be set to zero.

Returns

Protected VARBYTE value.

Exception

If the user does not have protect access rights in the policy, then the UDF terminates with an error message explaining what went wrong.

Example

```
select PTY_VARCHARLATINENC('Any character value! ', 'AES256',500,0,0);
```

4.6.1.3.2 pty_varcharlatindec

This UDF unprotects the protected string data.

pty_varcharlatindec(col VARBYTE, dataelement VARCHAR, resultlen INTEGER, communicationid INTEGER, scid INTEGER)

Parameters

Name	Type	Description
col	VARBYTE	Specifies the data to unprotect.
dataelement	VARCHAR	Specifies the name of the data element.
resultlen	INTEGER	Specifies the length of the buffer to hold the result.
communicationid	INTEGER	Specifies the location where the UDF will find the policy. The value must be the same as configured in the *pepserver.cfg* file.
scid	INTEGER	Specifies the security coordinate ID. The parameter is no longer used. The value of the parameter must be set to zero.

Returns

Unprotected character value.

NULL: When user has no access to data in the policy.

Exception

If configured in policy and user does not have access, then the UDF terminates with an error message explaining what went wrong.

Example

```
select PTY_VARCHARLATINDEC(pty_varcharlatinenc('Any character value! ',
'dataelement',500,0,0 ), 'dataelement',500,0,0 );
```

4.6.1.3.3 pty_varcharlatindecex

This UDF unprotects the protected string data, and raises error instead of returning NULL, if user does not have access rights.

pty_varcharlatindecex(col VARCHAR, dataelement VARCHAR, resultlen INTEGER, communicationid INTEGER, scid INTEGER)

Parameters

Name	Type	Description
col	VARCHAR	Specifies the data to unprotect.
dataelement	VARCHAR	Specifies the name of the data element to check.
resultlen	INTEGER	Specifies the length of the buffer to hold the result.
communicationid	INTEGER	Specifies the location where the UDF will find the policy. The value must be the same as configured in the *pepserver.cfg* file.
scid	INTEGER	Specifies the security coordinate ID. The parameter is no longer used. The value of the parameter must be set to zero.

Returns

Unprotected character value.

Exception

If the user does not have access rights in the policy, then the UDF terminates with an error message explaining what went wrong.

Example

```
select PTY_VARCHARLATINDECEX(PTY_VARCHARLATINENC('ProtegrityProt', 'AES256',100,0,0 ),
'AES256',100,0,0 );
```

4.6.1.3.4 pty_varcharlatinins

This UDF protects string data with type-preserving data elements, such as, tokens, Format Preserving Encryption (FPE) data elements with ASCII as the plaintext encoding, and No Encryption for access control.

pty_varcharlatinins(col VARCHAR, dataelement VARCHAR, resultlen INTEGER, communicationid INTEGER, scid INTEGER)

Parameters

Name	Type	Description
col	VARCHAR	Specifies the data to protect.
dataelement	VARCHAR	Specifies the name of the data element.
resultlen	INTEGER	Specifies the length of the buffer to hold the result.
communicationid	INTEGER	Specifies the location where the UDF will find the policy. The value must be the same as configured in the *pepserver.cfg* file.
scid	INTEGER	Specifies the security coordinate ID. The parameter is no longer used. The value of the parameter must be set to zero.

Returns

Protected VARCHAR value.

Exception

If the user does not have access rights in the policy, then the UDF terminates with an error message explaining what went wrong.

Example

```
SELECT pty_varcharlatinins('Any character value! ', 'dataelement',500,0,0 );
```

Email Tokenization

This UDF can be used to tokenize email input type. In the following example, *email* is a token element created in the ESA of *email* type.

```
pty_varcharlatinins('email@protegrity.com','email',32,0,0);
```

Timestamp Tokenization

This UDF can be used to tokenize timestamp data. The following example displays a sample of timestamp tokenization:

```
sel pty_varcharlatinins(cast('22-09-1990' as varchar(32)),'alphanum',64,0,0);
```

4.6.1.3.5 pty_varcharlatinsel

This UDF unprotects the protected string data.

PTY_VARCHARLATINSEL(col VARCHAR, dataelement VARCHAR, resultlen INTEGER, communicationid INTEGER, scid INTEGER)

Parameters

Name	Type	Description
col	VARCHAR	Specifies the data to unprotect.
dataelement	VARCHAR	Specifies the name of the data element.
resultlen	INTEGER	Specifies the length of the buffer to hold the result.
communicationid	INTEGER	Specifies the location where the UDF will find the policy. The value must be the same as configured in the *pepserver.cfg* file.
scid	INTEGER	Specifies the security coordinate ID. The parameter is no longer used. The value of the parameter must be set to zero.

Returns

Unprotected character value.

Protected value: If this option is configured in the policy and user does not have access to data.

NULL: When user has no access to data in the policy.

Exception

If the user does not have access rights in the policy, then the UDF terminates with an error message explaining what went wrong.

Example

```
SELECT pty_varcharlatinsel(pty_varcharlatinins('Any character value! ',
'dataelement',500,0,0 ), 'dataelement',500,0,0 );
```

Email De-tokenization

This UDF can be used to de-tokenize email input type tokenized using the *pty_varcharlatinins* UDF. In the following example, *email* is a token element created in the ESA of *email* type.

```
pty_varcharlatinsel('F00CJ@protegrity.com','email',32,0,0);
```

Timestamp Data De-tokenization

This UDF can be used to de-tokenize timestamp data tokenized using the *pty_varcharlatinins* UDF. The following example displays a sample of timestamp data de-tokenization.

```
sel pty_varcharlatinsel('Lv/xo/Qx','alphanum',64,0,0);
```

4.6.1.3.6 pty_varcharlatinselex

This UDF unprotects the protected string data and raises error instead of returning NULL if the user does not have access.

pty_varcharlatinselex(col VARCHAR, dataelement VARCHAR, resultlen INTEGER, communicationid INTEGER, scid INTEGER)

Parameters

Name	Type	Description
col	VARCHAR	Specifies the data to unprotect.
dataelement	VARCHAR	Specifies the name of the data element.
resultlen	INTEGER	Specifies the length of the buffer to hold the result.
communicationid	INTEGER	Specifies the location where the UDF will find the policy. The value must be the same as configured in the *pepserver.cfg* file.
scid	INTEGER	Specifies the security coordinate ID. The parameter is no longer used. The value of the parameter must be set to zero.

Returns

Unprotected character value.

Protected value: If this option is configured in the policy and user does not have access to data.

Exception

If the user does not have access rights in the policy, then the UDF terminates with an error message explaining what went wrong.

Example

```
SELECT pty_varcharlatinselex(pty_varcharlatinins('Any character value! ',
'dataelement',500,0,0 ), 'dataelement',500,0,0 );
```

4.6.1.3.7 PTY_VARCHARLATINHASH

This UDF calculates the hash of a string data. This is a one-way function and data cannot be unprotected.

PTY_VARCHARLATINHASH(col VARCHAR, dataelement VARCHAR, resultlen INTEGER, communicationid INTEGER, scid INTEGER)

Parameters

Name	Type	Description
col	VARCHAR	Specifies the data to protect.
dataelement	VARCHAR	Specifies the name of the data element.
resultlen	INTEGER	Specifies the length of the buffer to hold the result.
communicationid	INTEGER	Specifies the location where the UDF will find the policy. The value must be the same as configured in the *pepserver.cfg* file.
scid	INTEGER	Specifies the security coordinate ID. The parameter is no longer used. The value of the parameter must be set to zero.

Returns

Protected VARBYTE value.

Exception

If the user does not have protect access rights in the policy, then the UDF terminates with an error message explaining what went wrong.

Example

```
SELECT pty_varcharlatinhash ('ProtegrityProt', 'HMAC_SHA1', 100,0,0);
```

4.6.1.4 VARCHAR UNICODE UDFs

The varchar UNICODE UDFs accept string data encoded in the UNICODE character set.

4.6.1.4.1 PTY_VARCHARUNICODEENC

This UDF protects Unicode string with a data element for encryption.

PTY_VARCHARUNICODEENC(col VARCHAR, dataelement VARCHAR, resultlen INTEGER, communicationid INTEGER, scid INTEGER)

Parameters

Name	Type	Description
col	VARCHAR	Specifies the data to protect.
dataelement	VARCHAR	Specifies the name of the data element.
resultlen	INTEGER	Specifies the length of the buffer to hold the result.
communicationid	INTEGER	Specifies the location where the UDF will find the policy. The value must be the same as configured in the *pepserver.cfg* file.
scid	INTEGER	Specifies the security coordinate ID. The parameter is no longer used. The value of the parameter must be set to zero.

Returns

Protected VARBYTE value.

Exception

If the user does not have protect access rights in the policy, then the UDF terminates with an error message explaining what went wrong.

Example

```
SELECT pty_varcharunicodeenc (TRANSLATE(CAST('ProtegrityProt' AS VARCHAR(50)) USING
LATIN_TO_UNICODE), 'AES_128',100,0,0 );
```

4.6.1.4.2 PTY_VARCHARUNICODEDEC

This UDF unprotects protected string data.

pty_varcharunicodedec(col VARBYTE, dataelement VARCHAR, resultlen INTEGER, communicationid INTEGER, scid INTEGER)

Parameters

Name	Type	Description
col	VARBYTE	Specifies the data to unprotect.
dataelement	VARCHAR	Specifies the name of the data element.
resultlen	INTEGER	Specifies the length of the buffer to hold the result.
communicationid	INTEGER	Specifies the location where the UDF will find the policy. The value must be the same as configured in the *pepserver.cfg* file.
scid	INTEGER	Specifies the security coordinate ID. The parameter is no longer used. The value of the parameter must be set to zero.

Returns

Unprotected Unicode character value.

NULL: When user has no access to data in the policy.

Exception

If configured in policy and user does not have access, then the UDF terminates with an error message explaining what went wrong.

Example

```
SELECT pty_varcharunicodedec( protegrity.pty_varcharunicodeenc(TRANSLATE(CAST
('ProtegrityProt' AS VARCHAR(50)) USING LATIN_TO_UNICODE, 'AES256',100,0,0),
'AES256',100,0,0 );
```

4.6.1.4.3 PTY_VARCHARUNICODEDECEX

This UDF unprotects protected string data and raises an error instead of returning NULL if the user does not have access.

PTY_VARCHARUNICODEDECEX(col VARBYTE, dataelement VARCHAR, resultlen INTEGER, communicationid INTEGER, scid INTEGER)

Parameters

Name	Type	Description
col	VARBYTE	Specifies the data to unprotect.
dataelement	VARCHAR	Specifies the name of the data element.
resultlen	INTEGER	Specifies the length of the buffer to hold the result.
communicationid	INTEGER	Specifies the location where the UDF will find the policy. The value must be the same as configured in the *pepserver.cfg* file.
scid	INTEGER	Specifies the security coordinate ID. The parameter is no longer used. The value of the parameter must be set to zero.

Returns

Unprotected character value.

NULL: When user has no access to data in the policy.

Exception

If the user does not have access rights in the policy, then the UDF terminates with an error message explaining what went wrong.

Example

```
select PTY_VARCHARUNICODEDECEX(PTY_VARCHARUNICODEENC(TRANSLATE(CAST ('ProtegrityProt'
AS VARCHAR(50)) USING LATIN_TO_UNICODE), 'AES256', 100, 0,0), 'AES256', 100, 0,0);
```

4.6.1.4.4 PTY_VARCHARUNICODEINS

This UDF protects Unicode string data with data elements, such as, tokens (Unicode Base64 and Unicode Gen2), Format Preserving Encryption (FPE) data elements with UTF-8, UTF-16LE, UTF-16BE as the plaintext encoding, and No Encryption for access control.

pty_varcharunicodeins(col VARCHAR, dataelement VARCHAR, resultlen INTEGER, communicationid INTEGER, scid INTEGER)

Parameters

Name	Type	Description

col	VARCHAR	Specifies the data to protect.
		
		Note:
		• The maximum input size for single-byte characters is 4096 code points.
		• The maximum input size for multi-byte characters will vary depending on the session character set.
		  • For the UTF-8/16 session character set, the UDF will accept a maximum of 2048 code points.
		  • For the ASCII session character set, the UDF will accept a maximum of 1024 code points.

dataelement	VARCHAR	Specifies the name of the data element.

Name	Type	Description
resultlen	INTEGER	Specifies the length of the buffer to hold the result.
communicationid	INTEGER	Specifies the location where the UDF will find the policy. The value must be the same as configured in the *pepserver.cfg* file.
scid	INTEGER	Specifies the security coordinate ID. The parameter is no longer used. The value of the parameter must be set to zero.

Returns

Protected VARCHAR value.

Exception

If the user does not have protect access rights in the policy, then the UDF terminates with an error message explaining what went wrong.

Example Unicode Base64

```
select PTY_VARCHARUNICODEINS(TRANSLATE(CAST ('ProtegrityProt' AS VARCHAR(50)) USING
LATIN_TO_UNICODE), 'TE_Unicode_base64', 100, 0,0);
```

Example Unicode Gen2

> **Note:** The unicode Gen2 data elements supports the newly introduced SLT_X_1 tokenizer along with the existing SLT_1_3 tokenizer.
>
> For more information about the Unicode Gen2 data elements, refer to section *Unicode Gen2* in the *Protection Methods Reference Guide 9.1.0.0*.

```
select PTY_VARCHARUNICODEINS(TRANSLATE(CAST ('ProtegrityProt' AS VARCHAR(50))
USINGLATIN_TO_UNICODE), 'TE_UG2_SLT_13_L2R2_Y_BasicLatin', 100, 0,0);
```

```
select PTY_VARCHARUNICODEINS(TRANSLATE(CAST ('▒ AS VARCHAR(1000))
USINGLATIN_TO_UNICODE), 'TE_UG2_SLTX1_L2R2_N_IPA_Greek_Coptic_UTF16LE', 1000, 0,0);
```

4.6.1.4.5 PTY_VARCHARUNICODESEL

This UDF unprotects Unicode string data protected by data elements, such as, tokens (Unicode Base64 and Unicode Gen2), Format Preserving Encryption (FPE) data elements with any plaintext encoding type, and No Encryption for access control.

> **Note:** This UDF does not support masking.

pty_varcharunicodesel(col VARCHAR, dataelement VARCHAR, resultlen INTEGER, communicationid INTEGER, scid INTEGER)

Parameters

Name	Type	Description
col	VARCHAR	Specifies the data to unprotect. **Note:** • The maximum input size for single-byte characters is 4096 code points. • The maximum input size for multi-byte characters will vary depending on the session character set. • For the UTF-8/16 session character set, the UDF will accept a maximum of 2048 code points.

Name	Type	Description
		• For the ASCII session character set, the UDF will accept a maximum of 1024 code points.
dataelement	VARCHAR	Specifies the name of the data element.
resultlen	INTEGER	Specifies the length of the buffer to hold the result.
communicationid	INTEGER	Specifies the location where the UDF will find the policy. The value must be the same as configured in the *pepserver.cfg* file.
scid	INTEGER	Specifies the security coordinate ID. The parameter is no longer used. The value of the parameter must be set to zero.

Returns

Unprotected character value.

Protected value: If this option is configured in the policy and the user does not have access to data.

NULL: When the user has no access to data in the policy.

Exception

If configured in policy and the user does not have access, then the UDF terminates with an error message explaining what went wrong.

Example Unicode Base64

```
select PTY_VARCHARUNICODESEL(PTY_VARCHARUNICODEINS(TRANSLATE(CAST ('ProtegrityProt'
AS VARCHAR(50)) USING LATIN_TO_UNICODE), 'TE_Unicode_base64', 100, 0,0),
'TE_Unicode_base64', 100, 0,0);
```

Example Unicode Gen2

> **Note:** The Unicode Gen2 data elements support the newly introduced SLT_X_1 tokenizer along with the existing SLT_1_3 tokenizer. For more information, refer to section *Unicode Gen2* in the *Protection Methods Reference Guide 9.1.0.0*.

```
select PTY_VARCHARUNICODESEL(PTY_VARCHARUNICODEINS(TRANSLATE(CAST ('ProtegrityProt' AS
VARCHAR(50)) USING LATIN_TO_UNICODE), 'TE_UG2_SLT_13_L2R2_Y_BasicLatin', 100, 0,0),
'TE_UG2_SLT_13_L2R2_Y_BasicLatin', 100, 0,0);
```

```
select PTY_VARCHARUNICODESEL(PTY_VARCHARUNICODEINS(TRANSLATE(CAST ('▒ AS VARCHAR(1000))
USINGLATIN_TO_UNICODE), 'TE_UG2_SLTX1_L2R2_N_IPA_Greek_Coptic_UTF16LE', 1000, 0,0),
'TE_UG2_SLTX1_L2R2_N_IPA_Greek_Coptic_UTF16LE', 1000, 0,0);
```

4.6.1.4.6 PTY_VARCHARUNICODESELEX

This UDF unprotects string data protected by data elements, such as, tokens, Format Preserving Encryption (FPE) data elements with any plaintext encoding type, and No Encryption for access control and raises errors instead of returning NULL if the user does not have access.

> **Note:** This UDF does not support masking.

pty_varcharunicodeselex(col VARCHAR, dataelement VARCHAR, resultlen INTEGER, communicationid INTEGER, scid INTEGER)

Parameters

Name	Type	Description
col	VARCHAR	Specifies the data to unprotect.

Name	Type	Description
dataelement	VARCHAR	Specifies the name of the data element.
resultlen	INTEGER	Specifies the length of the buffer to hold the result.
communicationid	INTEGER	Specifies the location where the UDF will find the policy. The value must be the same as configured in the *pepserver.cfg* file.
scid	INTEGER	Specifies the security coordinate ID. The parameter is no longer used. The value of the parameter must be set to zero.

Returns

Unprotected character value.

Protected value: If this option is configured in the policy and user does not have access to data.

NULL: When user has no access to data In the policy.

Exception

If the user does not have access rights in the policy, then the UDF terminates with an error message.

Example

```
select PTY_VARCHARUNICODESELEX(PTY_VARCHARUNICODEINS(TRANSLATE(CAST ('ProtegrityProt'
AS VARCHAR(50)) USING LATIN_TO_UNICODE), 'NoEncryption', 100, 0,0), 'NoEncryption',
100, 0,0);
```

4.6.1.5 FLOAT UDFs

4.6.1.5.1 PTY_FLOATENC

This UDF protects float value with data element for encryption.

PTY_FLOATENC(col FLOAT, dataelement VARCHAR, resultlen INTEGER, communicationid INTEGER, scid INTEGER)

Parameters

Name	Type	Description
col	FLOAT	Specifies the data to protect.
dataelement	VARCHAR	Specifies the name of the data element.
resultlen	INTEGER	Specifies the length of the buffer to hold the result.
communicationid	INTEGER	Specifies the location where the UDF will find the policy. The value must be the same as configured in the *pepserver.cfg* file.
scid	INTEGER	Specifies the security coordinate ID. The parameter is no longer used. The value of the parameter must be set to zero.

Returns

Protected VARBYTE value.

Exception

If the user does not have protect access rights in the policy, then the UDF terminates with an error message explaining what went wrong.

Example

```
select PTY_FLOATENC(26656.0, 'AES256', 100, 0,0);
```

4.6.1.5.2 PTY_FLOATDEC

This UDF unprotects protected float value.

PTY_FLOATDEC(col VARBYTE, dataelement VARCHAR, communicationid INTEGER, scid INTEGER)

Parameters

Name	Type	Description
col	VARBYTE	Specifies the data to unprotect.
dataelement	VARCHAR	Specifies the name of the data element.
communicationid	INTEGER	Specifies the location where the UDF will find the policy. The value must be the same as configured in the *pepserver.cfg* file.
scid	INTEGER	Specifies the security coordinate ID. The parameter is no longer used. The value of the parameter must be set to zero.

Returns

Unprotected FLOAT value.

NULL: When user has no access to data in the policy.

Exception

If configured in policy and user does not have access, then the UDF terminates with an error message explaining what went wrong.

Example

```
select PTY_FLOATDEC(PTY_FLOATENC(26656.0, 'AES256', 100, 0,0), 'AES256', 0,0);
```

4.6.1.5.3 PTY_FLOATDECEX

This UDF unprotects protected float value and raises an error instead of returning NULL if user does not have access.

PTY_FLOATDECEX(col VARBYTE, dataelement VARCHAR, communicationid INTEGER, scid INTEGER)

Parameters

Name	Type	Description
col	VARBYTE	Specifies the data to unprotect.
dataelement	VARCHAR	Specifies the name of the data element.
communicationid	INTEGER	Specifies the location where the UDF will find the policy. The value must be the same as configured in the *pepserver.cfg* file.
scid	INTEGER	Specifies the security coordinate ID. The parameter is no longer used. The value of the parameter must be set to zero.

Returns

Unprotected FLOAT value.

NULL: When user has no access to data in the policy.

Exception

If the user does not have access rights in the policy, then the UDF terminates with an error message.

Example

```
select PTY_FLOATDECEX(PTY_FLOATENC(26656.0, 'AES256', 100, 0,0), 'AES256', 0,0);
```

4.6.1.5.4 PTY_FLOATHASH

This UDF calculates the hash value for a float value. This is a one-way function and the data cannot be unprotected.

pty_floathash(col FLOAT, dataelement VARCHAR, resultlen INTEGER, communicationid INTEGER, scid INTEGER)

Parameters

Name	Type	Description
col	FLOAT	Specifies the data to unprotect.
dataelement	VARCHAR	Specifies the name of the data element.
resultlen	INTEGER	Specifies the length of the buffer to hold the result.
communicationid	INTEGER	Specifies the location where the UDF will find the policy. The value must be the same as configured in the *pepserver.cfg* file.
scid	INTEGER	Specifies the security coordinate ID. The parameter is no longer used. The value of the parameter must be set to zero.

Returns

Unprotected VARBYTE value.

NULL: When user has no access to data in the policy.

Exception

If the user does not have protect access rights in the policy, then the UDF terminates with an error message explaining what went wrong.

Example

```
select PTY_FLOATHASH(26656.0, 'HMAC_SHA1', 100, 0,0);
```

4.6.1.6 INTEGER UDFs

4.6.1.6.1 PTY_INTEGERENC

This UDF protects integer value with a data element for encryption.

pty_integerenc(col INTEGER, dataelement VARCHAR, resultlen INTEGER, communicationid INTEGER, scid INTEGER)

Parameters

Name	Type	Description
col	INTEGER	Specifies the data to protect.
dataelement	VARCHAR	Specifies the name of the data element.
resultlen	INTEGER	Specifies the length of the buffer to hold the result.
communicationid	INTEGER	Specifies the location where the UDF will find the policy. The value must be the same as configured in the *pepserver.cfg* file.
scid	INTEGER	Specifies the security coordinate ID. The parameter is no longer used. The value of the parameter must be set to zero.

Returns

Protected VARBYTE value.

NULL: When user has no access to data in the policy.

Exception

If the user does not have protect access rights in the policy, then the UDF terminates with an error message explaining what went wrong.

Example

```
select PTY_INTEGERENC(1234, 'AES256', 100, 0,0);
```

4.6.1.6.2 PTY_INTEGERDEC

This UDF unprotects protected integer value.

pty_integerdec(col VARBYTE, dataelement VARCHAR, communicationid INTEGER, scid INTEGER)

Parameters

Name	Type	Description
col	VARBYTE	Specifies the data to unprotect.
dataelement	VARCHAR	Specifies the name of the data element.
communicationid	INTEGER	Specifies the location where the UDF will find the policy. The value must be the same as configured in the *pepserver.cfg* file.
scid	INTEGER	Specifies the security coordinate ID. The parameter is no longer used. The value of the parameter must be set to zero.

Returns

Unprotected INTEGER value.

NULL: When user has no access to data in the policy.

Exception

If configured in policy and user does not have access, then the UDF terminates with an error message explaining what went wrong.

Example

```
select PTY_INTEGERDEC(PTY_INTEGERENC(1234, 'AES256', 100, 0,0), 'AES256', 0,0);
```

4.6.1.6.3 PTY_INTEGERDECEX

This UDF unprotects protected integer value and throws an error instead of returning NULL if user does not have access.

pty_integerdecex(col VARBYTE, dataelement VARCHAR, communicationid INTEGER, scid INTEGER)

Parameters

Name	Type	Description
col	VARBYTE	Specifies the data to unprotect.
dataelement	VARCHAR	Specifies the name of the data element.
communicationid	INTEGER	Specifies the location where the UDF will find the policy. The value must be the same as configured in the *pepserver.cfg* file.
scid	INTEGER	Specifies the security coordinate ID. The parameter is no longer used. The value of the parameter must be set to zero.

Returns

Unprotected INTEGER value.

NULL: When user has no access to data in the policy.

Exception

If the user does not have access rights in the policy, then the UDF terminates with an error message.

Example

```
select PTY_INTEGERDECEX(PTY_INTEGERENC(1234, 'AES256', 100, 0,0), 'AES256', 0,0);
```

4.6.1.6.4 PTY_INTEGERINS

This UDF protects integer value with type-preserving data elements, such as, tokens and No Encryption for access control.

pty_integerins(col INTEGER, dataelement VARCHAR, resultlen INTEGER, communicationid INTEGER, scid INTEGER)

Parameters

Name	Type	Description
col	INTEGER	Specifies the data to protect.
dataelement	VARCHAR	Specifies the name of the data element.
resultlen	INTEGER	Specifies the length of the buffer to hold the result.
communicationid	INTEGER	Specifies the location where the UDF will find the policy. The value must be the same as configured in the *pepserver.cfg* file.
scid	INTEGER	Specifies the security coordinate ID. The parameter is no longer used. The value of the parameter must be set to zero.

Returns

Protected INTEGER value.

NULL: When user has no access to data in the policy.

Exception

If the user does not have protect access rights in the policy, then the UDF terminates with an error message explaining what went wrong.

Example

```
select PTY_INTEGERINS(1234, 'TE_INT_4', 100, 0,0);
```

4.6.1.6.5 PTY_INTEGERSEL

This UDF unprotects the protected integer value.

pty_integersel(col INTEGER, dataelement VARCHAR, communicationid INTEGER, scid INTEGER)

Parameters

Name	Type	Description
col	INTEGER	Specifies the data to unprotect.
dataelement	VARCHAR	Specifies the name of the data element.
communicationid	INTEGER	Specifies the location where the UDF will find the policy. The value must be the same as configured in the *pepserver.cfg* file.
scid	INTEGER	Specifies the security coordinate ID. The parameter is no longer used. The value of the parameter must be set to zero.

Returns

Unprotected INTEGER value.

Protected value: If this option is configured in policy and user does not have access to data.

NULL: When user has no access to data in the policy.

Exception

If configured in policy and user does not have access, then the UDF terminates with an error message explaining what went wrong.

Example

```
select PTY_INTEGERSEL(PTY_INTEGERINS(1234, 'TE_INT_4', 100, 0,0), 'TE_INT_4', 0,0);
```

4.6.1.6.6 PTY_INTEGERSELEX

This UDF unprotects protected integer value and throws an error instead of returning NULL if user does not have access.

pty_integerselex(col INTEGER, dataelement VARCHAR, communicationid INTEGER, scid INTEGER)

Parameters

Name	Type	Description
col	INTEGER	Specifies the data to unprotect.
dataelement	VARCHAR	Specifies the name of the data element.
communicationid	INTEGER	Specifies the location where the UDF will find the policy. The value must be the same as configured in the *pepserver.cfg* file.
scid	INTEGER	Specifies the security coordinate ID. The parameter is no longer used. The value of the parameter must be set to zero.

Returns

Unprotected INTEGER value.

Protected value: If the option is configured in policy and user does not have access to data.

NULL: When user has no access to data in the policy.

Exception

If the user does not have access rights in the policy, then the UDF terminates with an error message.

Example

```
select PTY_INTEGERSELEX(PTY_INTEGERINS(1234, 'TE_INT_4', 100, 0,0), 'TE_INT_4', 0,0);
```

4.6.1.6.7 PTY_INTEGERHASH

This UDF calculates hash value for integer value. This is a one-way function and data cannot be unprotected.

pty_integerhash(col INTEGER, dataelement VARCHAR, resultlen INTEGER, communicationid INTEGER, scid INTEGER)

Parameters

Name	Type	Description
col	INTEGER	Specifies the data to protect.
dataelement	VARCHAR	Specifies the name of the data element.
resultlen	INTEGER	Specifies the length of the buffer to hold the result.
communicationid	INTEGER	Specifies the location where the UDF will find the policy. The value must be the same as configured in the *pepserver.cfg* file.
scid	INTEGER	Specifies the security coordinate ID. The parameter is no longer used. The value of the parameter must be set to zero.

Returns

Protected VARBYTE value

NULL: When user has no access to data in the policy.

Exception

If the user does not have protect access rights in the policy, then the UDF terminates with an error message explaining what went wrong.

Example

```
select PTY_INTEGERHASH(1234, 'HMAC_SHA1', 100, 0,0);
```

4.6.1.7 BIGINT UDFs

4.6.1.7.1 PTY_BIGINTENC

This UDF protects BIGINT value with a data element for encryption.

pty_bigintenc(col BIGINT, dataelement VARCHAR, resultlen INTEGER, communicationid INTEGER, scid INTEGER)

Parameters

Name	Type	Description
col	BIGINT	Specifies the data to protect.
dataelement	VARCHAR	Specifies the name of the data element.
resultlen	INTEGER	Specifies the length of the buffer to hold the result.
communicationid	INTEGER	Specifies the location where the UDF will find the policy. The value must be the same as configured in the *pepserver.cfg* file.
scid	INTEGER	Specifies the security coordinate ID. The parameter is no longer used. The value of the parameter must be set to zero.

Returns

Protected VARBYTE value

NULL: When user has no access to data in the policy.

Exception

If the user does not have protect access rights in the policy, then the UDF terminates with an error message explaining what went wrong.

Example

```
select PTY_BIGINTENC(12345678,'AES256',100,0,0);
```

4.6.1.7.2 PTY_BIGINTDEC

This UDF unprotects bigint value.

pty_bigintdec(col VARBYTE, dataelement VARCHAR, communicationid INTEGER, scid INTEGER)

Parameters

Name	Type	Description
col	VARBYTE	Specifies the data to unprotect.
dataelement	VARCHAR	Specifies the name of the data element.
communicationid	INTEGER	Specifies the location where the UDF will find the policy. The value must be the same as configured in the *pepserver.cfg* file.
scid	INTEGER	Specifies the security coordinate ID. The parameter is no longer used. The value of the parameter must be set to zero.

Returns

Unprotected BIGINT value.

NULL: When user has no access to data in the policy.

Exception

If configured in policy and user does not have access, then the UDF terminates with an error message explaining what went wrong.

Example

```
select PTY_BIGINTDEC(PTY_BIGINTENC(12345678,'AES256',100,0,0),'AES256',0,0);
```

4.6.1.7.3 PTY_BIGINTDECEX

This UDF unprotects the protected bigint value and throws an error instead of returning NULL if the user does not have access.

pty_bigintdecex(col VARBYTE, dataelement VARCHAR, communicationid INTEGER, scid INTEGER)

Parameters

Name	Type	Description
col	VARBYTE	Specifies the data to unprotect.
dataelement	VARCHAR	Specifies the name of the data element.
communicationid	INTEGER	Specifies the location where the UDF will find the policy. The value must be the same as configured in the *pepserver.cfg* file.
scid	INTEGER	Specifies the security coordinate ID. The parameter is no longer used. The value of the parameter must be set to zero.

Returns

Unprotected BIGINT value.

NULL: When user has no access to data in the policy.

Exception

If the user does not have access rights in the policy, then the UDF terminates with an error message.

Example

```
select PTY_BIGINTDECEX(PTY_BIGINTENC(12345678,'AES256',100,0,0),'AES256',0,0);
```

4.6.1.7.4 PTY_BIGINTINS

This UDF protects the BIGINT value with type-preserving data elements, such as, tokens and No Encryption for access control.

pty_bigintins(col BIGINT, dataelement VARCHAR, resultlen INTEGER, communicationid INTEGER, scid INTEGER)

Parameters

Name	Type	Description
col	BIGINT	Specifies the data to protect.
dataelement	VARCHAR	Specifies the name of the data element.
resultlen	INTEGER	Specifies the length of the buffer to hold the result.
communicationid	INTEGER	Specifies the location where the UDF will find the policy. The value must be the same as configured in the *pepserver.cfg* file.
scid	INTEGER	Specifies the security coordinate ID. The parameter is no longer used. The value of the parameter must be set to zero.

Returns

Protected BIGINT value

NULL: When user has no access to data in the policy.

Exception

If the user does not have protect access rights in the policy, then the UDF terminates with an error message explaining what went wrong.

Example

```
select PTY_BIGINTINS(12345678, 'TE_INT_8', 100, 0,0);
```

4.6.1.7.5 PTY_BIGINTSEL

This UDF unprotects the bigint value.

pty_bigintsel(col BIGINT, dataelement VARCHAR, communicationid INTEGER, scid INTEGER)

Parameters

Name	Type	Description
col	BIGINT	Specifies the data to unprotect.
dataelement	VARCHAR	Specifies the name of the data element.
communicationid	INTEGER	Specifies the location where the UDF will find the policy. The value must be the same as configured in the *pepserver.cfg* file.
scid	INTEGER	Specifies the security coordinate ID. The parameter is no longer used. The value of the parameter must be set to zero.

Returns

Unprotected BIGINT value.

Protected value: If this option is configured in policy and user does not have access to data.

NULL: When user has no access to data in the policy.

Exception

If configured in policy and user does not have access, then the UDF terminates with an error message explaining what went wrong.

Example

```
select PTY_BIGINTSEL(PTY_BIGINTINS(12345678, 'TE_INT_8', 100, 0,0), 'TE_INT_8',0,0);
```

4.6.1.7.6 PTY_BIGINTSELEX

This UDF unprotects the protected bigint value and throws an error instead of returning NULL if user does not have access.

pty_bigintselex(col BIGINT, dataelement VARCHAR, communicationid INTEGER, scid INTEGER)

Parameters

Name	Type	Description
col	BIGINT	Specifies the data to unprotect.
dataelement	VARCHAR	Specifies the name of the data element.
communicationid	INTEGER	Specifies the location where the UDF will find the policy. The value must be the same as configured in the *pepserver.cfg* file.
scid	INTEGER	Specifies the security coordinate ID. The parameter is no longer used. The value of the parameter must be set to zero.

Returns

Unprotected BIGINT value.

Protected value: If this option is configured in policy and user does not have access to data.

NULL: When user has no access to data in the policy.

Exception

If the user user does not have access rights in the policy, then the UDF terminates with an error message explaining what went wrong.

Example

```
select PTY_BIGINTSELEX(PTY_BIGINTINS(12345678, 'TE_INT_8', 100, 0,0), 'TE_INT_8',0,0);
```

4.6.1.8 DATE UDFs

The dates can be protected using encryption and tokenization as the data protection method. The native UDFs, such as, *DATEENC* and *DATEDEC*, can be used for encryption and decryption respectively. To tokenize the date formats using the date data element, the data must be cast to the VARCHAR/CHAR type and then protected/unprotected with the *pty_varcharlatinins* or *pty_varcharlatinsel* UDFs.

To avoid any performance issues resulting due to casting of the data, a general best practice is to protect the data and present the decryption related UDFs in the tables as views to authorized users only. This eliminates the unauthorized user's access to the decryption UDFs and has the protected data only. The decryption process is limited to authorized users and thus, does not cause any performance impact because the UDFs are executed restrictively.

4.6.1.8.1 PTY_DATEENC

This UDF protects the date value with a data element for encryption.

pty_dateenc(col DATE, dataelement VARCHAR, resultlen INTEGER, communicationid INTEGER, scid INTEGER)

Parameters

Name	Type	Description
col	DATE	Specifies the data to protect.
dataelement	VARCHAR	Specifies the name of the data element.
resultlen	INTEGER	Specifies the length of the buffer to hold the result.
communicationid	INTEGER	Specifies the location where the UDF will find the policy. The value must be the same as configured in the *pepserver.cfg* file.
scid	INTEGER	Specifies the security coordinate ID. The parameter is no longer used. The value of the parameter must be set to zero.

Returns

Protected VARBYTE value

Exception

If the user does not have protect access rights in the policy, then the UDF terminates with an error message explaining what went wrong.

Example

```
select PTY_DATEENC(CAST ('22 Nov 90' AS DATE FORMAT 'DD-MMM-YY'), 'AES256', 100, 0,0);
```

4.6.1.8.2 PTY_DATEDEC

This UDF unprotects the protected date value.

pty_datedec(col VARBYTE, dataelement VARCHAR, communicationid INTEGER, scid INTEGER)

Parameters

Name	Type	Description
col	VARBYTE	Specifies the data to unprotect.
dataelement	VARCHAR	Specifies the name of the data element.
communicationid	INTEGER	Specifies the location where the UDF will find the policy. The value must be the same as configured in the *pepserver.cfg* file.

Name	Type	Description
scid	INTEGER	Specifies the security coordinate ID. The parameter is no longer used. The value of the parameter must be set to zero.

Returns

Unprotected DATE value.

> **Note:** The output displayed is as per the system date format.

Exception

If configured in policy and user does not have access, then any of the following values is returned depending on your data element and policy permission settings:

- Exception - The UDF terminates with an error message explaining what went wrong.
- Protected value
- NULL

For more information about the data element and policy permission settings, refer to section *Adding Permissions to Policy* in the *Policy Management Guide*.

Example

```
select PTY_DATEDEC(PTY_DATEENC(CAST ('22 Sep 90' AS DATE FORMAT 'DD-MMM-YY'), 'AES256',
100, 0,0), 'AES256', 0,0);
```

4.6.1.8.3 PTY_DATEDECEX

This UDF unprotects the protected date value and throws an error instead of returning NULL if user does not have access.

pty_datedecex(col VARBYTE, dataelement VARCHAR, communicationid INTEGER, scid INTEGER)

Parameters

Name	Type	Description
col	VARBYTE	Specifies the data to unprotect.
dataelement	VARCHAR	Specifies the name of the data element.
communicationid	INTEGER	Specifies the location where the UDF will find the policy. The value must be the same as configured in the *pepserver.cfg* file.
scid	INTEGER	Specifies the security coordinate ID. The parameter is no longer used. The value of the parameter must be set to zero.

Returns

Unprotected DATE value.

Exception

If the user does not have access rights in the policy, then the UDF terminates with an error message explaining what went wrong.

Example

```
select PTY_DATEDECEX(PTY_DATEENC(CAST ('22 Sep 90' AS DATE FORMAT 'DD-MMM-YY'),
'AES256', 100, 0,0), 'AES256', 0,0);
```

4.6.1.9 8-BYTE AND 16-BYTE DECIMAL UDFs

These UDFs work on DECIMAL data types that are either 8 or 16 bytes in size. The 8-byte DECIMALs have a precision between 10 and 18 digits, while 16-byte DECIMALs have a precision between 19 and 38 digits.

Only one set of DECIMAL UDFs can be created for each range. The UDF name must be provided by the user. It is recommended that you replace <n> with, for example, 10_2 if the target data type is DECIMAL(10,2) to get a function PTY_DECIMAL_10_2ENC, or 22_3 if target data type is DECIMAL(22,3) to get PTY_DECIMAL_22_3ENC.

4.6.1.9.1 PTY_DECIMAL<n>ENC

This UDF protects the decimal value with a data element for encryption.

pty_decimal<n>enc(col DECIMAL<m, n>, dataelement VARCHAR, resultlen INTEGER, communicationid INTEGER, scid INTEGER)

Parameters

Name	Type	Description
col	DECIMAL(m,n)	Specifies the data to protect.
dataelement	VARCHAR	Specifies the name of the data element.
resultlen	INTEGER	Specifies the length of the buffer to hold the result.
communicationid	INTEGER	Specifies the location where the UDF will find the policy. The value must be the same as configured in the *pepserver.cfg* file.
scid	INTEGER	Specifies the security coordinate ID. The parameter is no longer used. The value of the parameter must be set to zero.

Returns

Protected VARBYTE value.

NULL: When user has no access to data in the policy.

Exception

If the user does not have access rights in the policy, then the UDF terminates with an error message explaining what went wrong.

Example

```
select PTY_DECIMAL37_1ENC(26656.0, 'AES256', 100, 0,0);
```

4.6.1.9.2 PTY_DECIMAL<n>DEC

This UDF unprotects the protected decimal value.

pty_decimal<n>dec(col VARBYTE, dataelement VARCHAR, communicationid INTEGER, scid INTEGER)

Parameters

Name	Type	Description
col	VARBYTE	Specifies the data to unprotect.
dataelement	VARCHAR	Specifies the name of the data element.
communicationid	INTEGER	Specifies the location where the UDF will find the policy. The value must be the same as configured in the *pepserver.cfg* file.
scid	INTEGER	Specifies the security coordinate ID. The parameter is no longer used. The value of the parameter must be set to zero.

Returns

Unprotected DECIMAL value.

NULL: When user has no access to data in the policy.

Exception

If configured in policy and user does not have access, then the UDF terminates with an error message explaining what went wrong.

Example

```
select PTY_DECIMAL37_1DEC(PTY_DECIMAL37_1ENC(26656.0, 'AES256', 100, 0,0), 'AES256',
0,0);
```

4.6.1.9.3 PTY_DECIMAL<n>DECEX

This UDF unprotects the protected decimal value and throws an error instead of returning NULL if user does not have access.

pty_decimal<n>decex(col VARBYTE, dataelement VARCHAR, communicationid INTEGER, scid INTEGER)

Parameters

Name	Type	Description
col	VARBYTE	Specifies the data to unprotect.
dataelement	VARCHAR	Specifies the name of the data element.
communicationid	INTEGER	Specifies the location where the UDF will find the policy. The value must be the same as configured in the *pepserver.cfg* file.
scid	INTEGER	Specifies the security coordinate ID. The parameter is no longer used. The value of the parameter must be set to zero.

Returns

Unprotected DECIMAL value.

NULL: When user has no access to data in the policy.

Exception

UIf configured in policy and user does not have access, then the UDF terminates with an error message explaining what went wrong.

Example

```
select PTY_DECIMAL37_1DECEX(PTY_DECIMAL37_1ENC(26656.0, 'AES256', 100, 0,0), 'AES256',
0,0);
```

4.6.1.10 JSON UDFs

These UDFs are used to protect and unprotect data for JSON data type. These UDFs have been introduced for Teradata 15.10 Release and later to support LOB or Large Objects that can be loaded to or extracted from Teradata Database tables. Depending on the data element chosen, the data is tokenized or encrypted. The data in JSON are protected as CLOBs.

The examples provided for protection and unprotection are for single queries.

4.6.1.10.1 PTY_JSONINS

This UDF protects JSON value with data elements, such as, tokens and No Encryption for access control.

pty_jsonins(col JSON, dataelement VARCHAR, resultlen INTEGER, communicationid INTEGER, scid INTEGER)

Parameters

Name	Type	Description
col or data	JSON	Specifies the JSON data to protect.
dataelement	VARCHAR	Specifies the name of the data element.
resultlen	INTEGER	Specifies the length of the buffer to hold the result.
communicationid	INTEGER	Specifies the location where the UDF will find the policy. The value must be the same as configured in the *pepserver.cfg* file.
scid	INTEGER	Specifies the security coordinate ID. The parameter is no longer used. The value of the parameter must be set to zero.

Returns

Protected CLOB (Character Large Objects) value.

NULL: When user has no access to data in the policy.

Exception

If the user does not have protect access rights in the policy, then the UDF terminates with an error message explaining what went wrong.

> **Note:** Tokenizing JSON data with Printable tokenization will not return a valid JSON format output.

Example

```
SELECT pty_jsonins(NEW JSON('{"emp_name" : "John Doe", "emp_address" : "Stamford 1"}'),
'TE_A_N_S23_L2R2_Y', 500, 0, 0);
```

4.6.1.10.2 PTY_JSONSEL

This UDF unprotects the protected JSON CLOBs.

pty_jsonsel(col CLOB, dataelement VARCHAR, resultlen INTEGER, communicationid INTEGER, scid INTEGER)

Parameters

Name	Type	Description
col or data	CLOB	Specifies the CLOB data to unprotect.
dataelement	VARCHAR	Specifies the name of the data element.
resultlen	INTEGER	Specifies the length of the buffer to hold the result.
communicationid	INTEGER	Specifies the location where the UDF will find the policy. The value must be the same as configured in the *pepserver.cfg* file.
scid	INTEGER	Specifies the security coordinate ID. The parameter is no longer used. The value of the parameter must be set to zero.

Returns

Unprotected JSON values.

NULL: When user has no access to data in the policy.

Exception

If configured in policy and user does not have access, UDF terminates with an error message explaining what went wrong.

Example

```
SELECT pty_jsonsel(NEW JSON('{"emp_name" : "John Doe", "emp_address" : "Stamford 1"}'),
'TE_A_N_S23_L2R2_Y', 500, 0, 0);
```

4.6.1.10.3 PTY_JSONSELEX

This UDF unprotects the protected JSON CLOBs using tokenization and returns an error instead of NULL, if user does not have access rights.

pty_jsonselex(col CLOB, dataelement VARCHAR, resultlen INTEGER, communicationid INTEGER, scid INTEGER)

Parameters

Name	Type	Description
col or data	CLOB	Specifies the CLOB data to unprotect.
dataelement	VARCHAR	Specifies the name of the data element.

Name	Type	Description
resultlen	INTEGER	Specifies the length of the buffer to hold the result.
communicationid	INTEGER	Specifies the location where the UDF will find the policy. The value must be the same as configured in the *pepserver.cfg* file.
scid	INTEGER	Specifies the security coordinate ID. The parameter is no longer used. The value of the parameter must be set to zero.

Returns

Unprotected JSON values.

NULL: When user has no access to data in the policy.

Exception

If the user does not have access rights in the policy, then the UDF terminates with an error.

Example

```
SELECT pty_jsonselex(NEW JSON('{"emp_name" : "John Doe", "emp_address" : "Stamford
1"}'), 'TE_A_N_S23_L2R2_Y', 500, 0, 0);
```

4.6.1.10.4 PTY_JSONENC

This UDF protects the JSON value with a data element for encryption.

pty_jsonenc(col JSON, dataelement VARCHAR, resultlen INTEGER, communicationid INTEGER, scid INTEGER)

Parameters

Name	Type	Description
col or data	JSON	Specifies the JSON data to protect.
dataelement	VARCHAR	Specifies the name of the data element.
resultlen	INTEGER	Specifies the length of the buffer to hold the result.
communicationid	INTEGER	Specifies the location where the UDF will find the policy. The value must be the same as configured in the *pepserver.cfg* file.
scid	INTEGER	Specifies the security coordinate ID. The parameter is no longer used. The value of the parameter must be set to zero.

Returns

Protected CLOB (Character Large Objects) value.

NULL: When user has no access to data in the policy.

Exception

If the user does not have protect access rights in the policy, then the UDF terminates with an error message explaining what went wrong.

Example

```
SELECT pty_jsonenc(NEW JSON('{"emp_name" : "John Doe", "emp_address" : "Stamford 1"}'),
'AES256', 500, 0, 0);
```

4.6.1.10.5 PTY_JSONDEC

This UDF unprotects the protected CLOB value with strong encryption.

pty_jsondec(col CLOB, dataelement VARCHAR, resultlen INTEGER, communicationid INTEGER, scid INTEGER)

Parameters

Name	Type	Description
col or data	CLOB	Specifies the CLOB data to unprotect.
dataelement	VARCHAR	Specifies the name of the data element.
resultlen	INTEGER	Specifies the length of the buffer to hold the result.
communicationid	INTEGER	Specifies the location where the UDF will find the policy. The value must be the same as configured in the *pepserver.cfg* file.
scid	INTEGER	Specifies the security coordinate ID. The parameter is no longer used. The value of the parameter must be set to zero.

Returns

Unprotected JSON values.

NULL: When user has no access to data in the policy.

Exception

If configured in policy and user does not have access, then the UDF terminates with an error message explaining what went wrong.

Example

```
SELECT pty_jsondec(NEW JSON('{"emp_name" : "John Doe", "emp_address" : "Stamford 1"}'),
'AES256', 500, 0, 0);
```

4.6.1.10.6 PTY_JSONDECEX

This UDF unprotects CLOB values with strong encryption and returns an error instead of NULL, if the user does not have access rights.

pty_jsondecex(col CLOB, dataelement VARCHAR, resultlen INTEGER, communicationid INTEGER, scid INTEGER)

Parameters

Name	Type	Description
col or data	CLOB	Specifies the CLOB data to unprotect.
dataelement	VARCHAR	Specifies the name of the data element.
resultlen	INTEGER	Specifies the length of the buffer to hold the result.
communicationid	INTEGER	Specifies the location where the UDF will find the policy. The value must be the same as configured in the *pepserver.cfg* file.
scid	INTEGER	Specifies the security coordinate ID. The parameter is no longer used. The value of the parameter must be set to zero.

Returns

Unprotected JSON values.

NULL: When user has no access to data in the policy.

Exception

If the user does not have access rights in the policy, then the UDF terminates with an error message explaining what went wrong.

Example

```
SELECT pty_jsondecex(NEW JSON('{"emp_name" : "John Doe", "emp_address" : "Stamford
1"}'), 'AES256', 500, 0, 0);
```

4.6.1.11 XML UDFs

These UDFs support the XML data type. XML content is stored in compact binary form or CLOBs that preserve the information set of the XML document. These UDFs have been introduced for Teradata 15.10 Release and later to support XML files that

can be loaded to or extracted from Teradata Database tables. Depending on the data element chosen, the data is tokenized or encrypted.

4.6.1.11.1 PTY_XMLINS

This UDF protects XML value with data elements, such as, tokens and No Encryption for access control.

pty_xmlins(col XML, dataelement VARCHAR, resultlen INTEGER, communicationid INTEGER, scid INTEGER)

Parameters

Name	Type	Description
col	XML	Specifies the XML data to protect.
dataelement	VARCHAR	Specifies the name of the data element.
resultlen	INTEGER	Specifies the length of the buffer to hold the result.
communicationid	INTEGER	Specifies the location where the UDF will find the policy. The value must be the same as configured in the *pepserver.cfg* file.
scid	INTEGER	Specifies the security coordinate ID. The parameter is no longer used. The value of the parameter must be set to zero.

Returns

Protected CLOB value.

NULL: When user has no access to data in the policy.

Exception

If the user does not have protect access rights in the policy, then the UDF terminates with an error message explaining what went wrong.

> **Note:** Tokenizing XML data with Printable tokenization will not return a valid XML format output.

Example

```
select
PTY_XMLINS(CREATEXML('<?xml version="1.0" encoding="UTF-8"?>
<Customer ID="C00-10101">
<Name>John Hancock</Name>
<Address>100 1st Street, San Francisco, CA 94118</Address>
<Phone1>(858)555-1234</Phone1>
<Phone2>(858)555-9876</Phone2>
<Fax>(858)555-9999</Fax>
<Email>John@somecompany.com</Email>
<Order Number="NW-01-16366" Date="2012-02-28">
<Contact>Mary Jane</Contact>
<Phone>(987)654-3210</Phone>
<ShipTo>Some company, 2467 Pioneer Road, San Francisco, CA - 94117</ShipTo>
<SubTotal>434.99</SubTotal>
<Tax>32.55</Tax>
<Total>467.54</Total>
<Item ID="001">
<Quantity>10</Quantity>
<PartNumber>F54709</PartNumber>
<Description>Motorola S10-HD Bluetooth Stereo Headphones</Description>
<UnitPrice>29.50</UnitPrice>
<Price>295.00</Price>
</Item>
<Item ID="101">
<Quantity>1</Quantity>
<PartNumber>Z19743</PartNumber>
<Description>Motorola Milestone XT800 Cell Phone</Description>
<UnitPrice>139.99</UnitPrice>
<Price>139.99</Price>
</Item>
```

```
</Order>
</Customer>'),'TE_A_N_S23_L2R2_Y',1500,0,0) "Protected Data";
```

4.6.1.11.2 PTY_XMLSEL

This UDF unprotects the protected CLOB value.

pty_xmlsel(col CLOB, dataelement VARCHAR, resultlen INTEGER, communicationid INTEGER, scid INTEGER)

Parameters

Name	Type	Description
col	CLOB	Specifies the CLOB data to unprotect.
dataelement	VARCHAR	Specifies the name of the data element.
resultlen	INTEGER	Specifies the length of the buffer to hold the result.
communicationid	INTEGER	Specifies the location where the UDF will find the policy. The value must be the same as configured in the *pepserver.cfg* file.
scid	INTEGER	Specifies the security coordinate ID. The parameter is no longer used. The value of the parameter must be set to zero.

Returns

Unprotected XML values.

NULL: When user has no access to data in the policy.

Exception

If configured in policy and user does not have access, then the UDF terminates with an error message explaining what went wrong.

Example

```
sel
PTY_XMLSEL(
PTY_XMLINS(CREATEXML('<?xml version="1.0" encoding="UTF-8"?>
<Customer ID="C00-10101">
<Name>John Hancock</Name>
<Address>100 1st Street, San Francisco, CA 94118</Address>
<Phone1>(858)555-1234</Phone1>
<Phone2>(858)555-9876</Phone2>
<Fax>(858)555-9999</Fax>
<Email>John@somecompany.com</Email>
<Order Number="NW-01-16366" Date="2012-02-28">
<Contact>Mary Jane</Contact>
<Phone>(987)654-3210</Phone>
<ShipTo>Some company, 2467 Pioneer Road, San Francisco, CA - 94117</ShipTo>
<SubTotal>434.99</SubTotal>
<Tax>32.55</Tax>
<Total>467.54</Total>
<Item ID="001">
<Quantity>10</Quantity>
<PartNumber>F54709</PartNumber>
<Description>Motorola S10-HD Bluetooth Stereo Headphones</Description>
<UnitPrice>29.50</UnitPrice>
<Price>295.00</Price>
</Item>
<Item ID="101">
<Quantity>1</Quantity>
<PartNumber>Z19743</PartNumber>
<Description>Motorola Milestone XT800 Cell Phone</Description>
<UnitPrice>139.99</UnitPrice>
<Price>139.99</Price>
</Item>
</Order>
</Customer>'),'TE_A_N_S23_L2R2_Y',1500,0,0),'TE_A_N_S23_L2R2_Y',1500,0,0) "UnProtected
Data";
```

4.6.1.11.3 PTY_XMLSELEX

This UDF unprotects the protected CLOB value with strong encryption and returns an error instead of NULL, if the user does not have access rights.

pty_xmlselex(col CLOB, dataelement VARCHAR, resultlen INTEGER, communicationid INTEGER, scid INTEGER)

Parameters

Name	Type	Description
col	CLOB	Specifies the CLOB data to unprotect.
dataelement	VARCHAR	Specifies the name of the data element.
resultlen	INTEGER	Specifies the length of the buffer to hold the result.
communicationid	INTEGER	Specifies the location where the UDF will find the policy. The value must be the same as configured in the *pepserver.cfg* file.
scid	INTEGER	Specifies the security coordinate ID. The parameter is no longer used. The value of the parameter must be set to zero.

Returns

Unprotected XML values.

NULL: When user has no access to data in the policy.

Exception

If the user does not have access rights in the policy, then the UDF terminates with an error message explaining what went wrong.

Example

```
sel
PTY_XMLSELEX(
PTY_XMLINS(CREATEXML('<?xml version="1.0" encoding="UTF-8"?>
<Customer ID="C00-10101">
<Name>John Hancock</Name>
<Address>100 1st Street, San Francisco, CA 94118</Address>
<Phone1>(858)555-1234</Phone1>
<Phone2>(858)555-9876</Phone2>
<Fax>(858)555-9999</Fax>
<Email>John@somecompany.com</Email>
<Order Number="NW-01-16366" Date="2012-02-28">
<Contact>Mary Jane</Contact>
<Phone>(987)654-3210</Phone>
<ShipTo>Some company, 2467 Pioneer Road, San Francisco, CA - 94117</ShipTo>
<SubTotal>434.99</SubTotal>
<Tax>32.55</Tax>
<Total>467.54</Total>
<Item ID="001">
<Quantity>10</Quantity>
<PartNumber>F54709</PartNumber>
<Description>Motorola S10-HD Bluetooth Stereo Headphones</Description>
<UnitPrice>29.50</UnitPrice>
<Price>295.00</Price>
</Item>
<Item ID="101">
<Quantity>1</Quantity>
<PartNumber>Z19743</PartNumber>
<Description>Motorola Milestone XT800 Cell Phone</Description>
<UnitPrice>139.99</UnitPrice>
<Price>139.99</Price>
</Item>
</Order>
</Customer>'),'TE_A_N_S23_L2R2_Y',1500,0,0),'TE_A_N_S23_L2R2_Y',1500,0,0) "UnProtected
Data";
```

4.6.1.11.4 PTY_XMLENC

This UDF protects the XML data with a data element for encryption.

pty_xmlenc(col XML, dataelement VARCHAR, resultlen INTEGER, communicationid INTEGER, scid INTEGER)

Parameters

Name	Type	Description
col	XML	Specifies the XML data to protect.
dataelement	VARCHAR	Specifies the name of the data element.
resultlen	INTEGER	Specifies the length of the buffer to hold the result.
communicationid	INTEGER	Specifies the location where the UDF will find the policy. The value must be the same as configured in the *pepserver.cfg* file.
scid	INTEGER	Specifies the security coordinate ID. The parameter is no longer used. The value of the parameter must be set to zero.

Returns

Protected CLOB value.

NULL: When user has no access to data in the policy.

Exception

If the user does not have protect access rights in the policy, then the UDF terminates with an error message explaining what went wrong.

Example

```
sel
PTY_XMLENC(CREATEXML('<?xml version="1.0" encoding="UTF-8"?>
<Customer ID="C00-10101">
<Name>John Hancock</Name>
<Address>100 1st Street, San Francisco, CA 94118</Address>
<Phone1>(858)555-1234</Phone1>
<Phone2>(858)555-9876</Phone2>
<Fax>(858)555-9999</Fax>
<Email>John@somecompany.com</Email>
<Order Number="NW-01-16366" Date="2012-02-28">
<Contact>Mary Jane</Contact>
<Phone>(987)654-3210</Phone>
<ShipTo>Some company, 2467 Pioneer Road, San Francisco, CA - 94117</ShipTo>
<SubTotal>434.99</SubTotal>
<Tax>32.55</Tax>
<Total>467.54</Total>
<Item ID="001">
<Quantity>10</Quantity>
<PartNumber>F54709</PartNumber>
<Description>Motorola S10-HD Bluetooth Stereo Headphones</Description>
<UnitPrice>29.50</UnitPrice>
<Price>295.00</Price>
</Item>
<Item ID="101">
<Quantity>1</Quantity>
<PartNumber>Z19743</PartNumber>
<Description>Motorola Milestone XT800 Cell Phone</Description>
<UnitPrice>139.99</UnitPrice>
<Price>139.99</Price>
</Item>
</Order>
</Customer>'),'AES256',1500,0,0) "Protected Data";
```

4.6.1.11.5 PTY_XMLDEC

This UDF unprotects the protected CLOB values.

pty_xmldec(col CLOB, dataelement VARCHAR, resultlen INTEGER, communicationid INTEGER, scid INTEGER)

Parameters

Name	Type	Description
col	CLOB	Specifies the CLOB data to unprotect.
dataelement	VARCHAR	Specifies the name of the data element.
resultlen	INTEGER	Specifies the length of the buffer to hold the result.
communicationid	INTEGER	Specifies the location where the UDF will find the policy. The value must be the same as configured in the *pepserver.cfg* file.
scid	INTEGER	Specifies the security coordinate ID. The parameter is no longer used. The value of the parameter must be set to zero.

Returns

Unprotected XML value.

NULL: When user has no access to data in the policy.

Exception

If the user does not have access rights in the policy, then the UDF terminates with an error message.

Example

```
select
PTY_XMLDEC(
PTY_XMLENC(CREATEXML('<?xml version="1.0" encoding="UTF-8"?>
<Customer ID="C00-10101">
<Name>John Hancock</Name>
<Address>100 1st Street, San Francisco, CA 94118</Address>
<Phone1>(858)555-1234</Phone1>
<Phone2>(858)555-9876</Phone2>
<Fax>(858)555-9999</Fax>
<Email>John@somecompany.com</Email>
<Order Number="NW-01-16366" Date="2012-02-28">
<Contact>Mary Jane</Contact>
<Phone>(987)654-3210</Phone>
<ShipTo>Some company, 2467 Pioneer Road, San Francisco, CA - 94117</ShipTo>
<SubTotal>434.99</SubTotal>
<Tax>32.55</Tax>
<Total>467.54</Total>
<Item ID="001">
<Quantity>10</Quantity>
<PartNumber>F54709</PartNumber>
<Description>Motorola S10-HD Bluetooth Stereo Headphones</Description>
<UnitPrice>29.50</UnitPrice>
<Price>295.00</Price>
</Item>
<Item ID="101">
<Quantity>1</Quantity>
<PartNumber>Z19743</PartNumber>
<Description>Motorola Milestone XT800 Cell Phone</Description>
<UnitPrice>139.99</UnitPrice>
<Price>139.99</Price>
</Item>
</Order>
</Customer>'),'AES256',1500,0,0),'AES256',1500,0,0) "UnProtected Data";
```

4.6.1.11.6 PTY_XMLDECEX

This UDF unprotects the protected CLOB value with strong encryption and returns an error instead of NULL if the user does not have access rights.

pty_xmldecex(col CLOB, dataelement VARCHAR, resultlen INTEGER, communicationid INTEGER, scid INTEGER)

Parameters

Name	Type	Description
col	CLOB	Specifies the CLOB data to protect.
dataelement	VARCHAR	Specifies the name of the data element.
resultlen	INTEGER	Specifies the length of the buffer to hold the result.
communicationid	INTEGER	Specifies the location where the UDF will find the policy. The value must be the same as configured in the *pepserver.cfg* file.
scid	INTEGER	Specifies the security coordinate ID. The parameter is no longer used. The value of the parameter must be set to zero.

Returns

Unprotected XML value.

NULL: When user has no access to data in the policy.

Exception

If the user does not have protect access rights in the policy, UDF terminates with an error message explaining what went wrong.

Example

```
select
PTY_XMLDECEX(
PTY_XMLENC(CREATEXML('<?xml version="1.0" encoding="UTF-8"?>
<Customer ID="C00-10101">
<Name>John Hancock</Name>
<Address>100 1st Street, San Francisco, CA 94118</Address>
<Phone1>(858)555-1234</Phone1>
<Phone2>(858)555-9876</Phone2>
<Fax>(858)555-9999</Fax>
<Email>John@somecompany.com</Email>
<Order Number="NW-01-16366" Date="2012-02-28">
<Contact>Mary Jane</Contact>
<Phone>(987)654-3210</Phone>
<ShipTo>Some company, 2467 Pioneer Road, San Francisco, CA - 94117</ShipTo>
<SubTotal>434.99</SubTotal>
<Tax>32.55</Tax>
<Total>467.54</Total>
<Item ID="001">
<Quantity>10</Quantity>
<PartNumber>F54709</PartNumber>
<Description>Motorola S10-HD Bluetooth Stereo Headphones</Description>
<UnitPrice>29.50</UnitPrice>
<Price>295.00</Price>
</Item>
<Item ID="101">
<Quantity>1</Quantity>
<PartNumber>Z19743</PartNumber>
<Description>Motorola Milestone XT800 Cell Phone</Description>
<UnitPrice>139.99</UnitPrice>
<Price>139.99</Price>
</Item>
</Order>
</Customer>'),'AES256',1500,0,0),'AES256',1500,0,0) "UnProtected Data";
```

4.6.2 Teradata UDFs for No Encryption

This section provides a detailed list of No Encryption User Defined Functions (UDFs) that can be used for access control.

4.6.2.1 FLOAT UDFs

4.6.2.1.1 PTY_FLOATINS

This UDF can be used with the No Encryption data element only.

pty_floatins(col FLOAT, dataelement VARCHAR, resultlen INTEGER, communicationid INTEGER, scid INTEGER)

Parameters

Name	Type	Description
col	FLOAT	Specifies the data to protect.
dataelement	VARCHAR	Specifies the name of the data element.
resultlen	INTEGER	Specifies the length of the buffer to hold the result.
communicationid	INTEGER	Specifies the location where the UDF will find the policy. The value must be the same as configured in the *pepserver.cfg* file.
scid	INTEGER	Specifies the security coordinate ID. The parameter is no longer used. The value of the parameter must be set to zero.

Returns

Protected FLOAT value.

Exception

If the user does not have protect access rights in the policy, then the UDF terminates with an error message explaining what went wrong.

Example

```
select PTY_FLOATINS(26656.0, 'NoEncryption', 100, 0,0);
```

4.6.2.1.2 PTY_FLOATSEL

This UDF unprotects the float value for a No Encryption data element.

pty_floatsel(col FLOAT, dataelement VARCHAR, communicationid INTEGER, scid INTEGER)

Parameters

Name	Type	Description
col	FLOAT	Specifies the data to unprotect.
dataelement	VARCHAR	Specifies the name of the data element.
resultlen	INTEGER	Specifies the length of the buffer to hold the result.
communicationid	INTEGER	Specifies the location where the UDF will find the policy. The value must be the same as configured in the *pepserver.cfg* file.
scid	INTEGER	Specifies the security coordinate ID. The parameter is no longer used. The value of the parameter must be set to zero.

Returns

Unprotected FLOAT value.

Exception

If configured in policy and user does not have access, then the UDF terminates with an error message explaining what went wrong.

Example

```
select PTY_FLOATSEL(PTY_FLOATINS(26656.0, 'NoEncryption', 100, 0,0), 'NoEncryption',
0,0);
```

4.6.2.1.3 PTY_FLOATSELEX

This UDF unprotects the float value protected with a No Encryption data element and returns an error instead of NULL if the user does not have access.

pty_floatselex(col FLOAT, dataelement VARCHAR, communicationid INTEGER, scid INTEGER)

Parameters

Name	Type	Description
col	FLOAT	Specifies the data to unprotect.
dataelement	VARCHAR	Specifies the name of the data element.
communicationid	INTEGER	Specifies the location where the UDF will find the policy. The value must be the same as configured in the *pepserver.cfg* file.
scid	INTEGER	Specifies the security coordinate ID. The parameter is no longer used. The value of the parameter must be set to zero.

Returns

Unprotected FLOAT value.

Exception

If the user does not have access rights in the policy, then the UDF terminates with an error message explaining what went wrong.

Example

```
select PTY_FLOATSELEX(PTY_FLOATINS(26656.0, 'NoEncryption', 100, 0,0), 'NoEncryption',
0,0);
```

4.6.2.2 DATE UDFs

This section provides Date UDFs that are applicable for No Encryption data elements.

4.6.2.2.1 PTY_DATEINS

This UDF protects a date value with a No Encryption data element to impose access control.

pty_dateins(col DATE, dataelement VARCHAR, resultlen INTEGER, communicationid INTEGER, scid INTEGER)

Parameters

Name	Type	Description
col	DATE	Specifies the data to protect.
dataelement	VARCHAR	Specifies the name of the data element.
resultlen	INTEGER	Specifies the length of the buffer to hold the result.
communicationid	INTEGER	Specifies the location where the UDF will find the policy. The value must be the same as configured in the *pepserver.cfg* file.
scid	INTEGER	Specifies the security coordinate ID. The parameter is no longer used. The value of the parameter must be set to zero.

Returns

Input value as is.

> **Note:** The output is displayed as per the system date format.

Exception

If the user does not have protect access rights in the policy, then the UDF terminates with an error message explaining what went wrong.

Example

```
select PTY_DATEINS(CAST ('22-09-1990' AS DATE FORMAT 'DD-MM-YYYY'), 'NoEncryption',
100, 0,0);
```

4.6.2.2.2 PTY_DATESEL

This UDF unprotects the date value that is protected using a No Encryption data element.

pty_datesel(col DATE, dataelement VARCHAR, communicationid INTEGER, scid INTEGER)

Parameters

Name	Type	Description
col	DATE	Specifies the data to unprotect.
dataelement	VARCHAR	Specifies the name of the data element.
communicationid	INTEGER	Specifies the location where the UDF will find the policy. The value must be the same as configured in the *pepserver.cfg* file.
scid	INTEGER	Specifies the security coordinate ID. The parameter is no longer used. The value of the parameter must be set to zero.

Returns

Input value as is.

Exception

If configured in policy and the user does not have access, then any of the following values is returned depending on your data element and policy permission settings:

- Exception - The UDF terminates with an error message explaining what went wrong.
- Protected value
- NULL

For more information about data element and policy permission settings, refer to section *Adding Permissions to Policy* in the *Policy Management Guide*.

Example

```
select PTY_DATESEL(PTY_DATEINS(CAST ('22-09-1990' AS DATE FORMAT 'DD-MM-YYYY'),
'NoEncryption', 100, 0,0), 'NoEncryption', 0,0);
```

4.6.2.2.3 PTY_DATESELEX

This UDF unprotects the date value that is protected with a No Encryption data element and returns an error instead of NULL if the user does not have access.

pty_dateselex(col DATE, dataelement VARCHAR, communicationid INTEGER, scid INTEGER)

Parameters

Name	Type	Description
col	DATE	Specifies the data to unprotect.
dataelement	VARCHAR	Specifies the name of the data element.
communicationid	INTEGER	Specifies the location where the UDF will find the policy. The value must be the same as configured in the *pepserver.cfg* file.
scid	INTEGER	Specifies the security coordinate ID. The parameter is no longer used. The value of the parameter must be set to zero.

Returns

Input value as is.

Exception

If the user does not have access rights in the policy, then the UDF terminates with an error message.

Example

```
select PTY_DATESELEX(PTY_DATEINS(CAST ('22-09-1990' AS DATE FORMAT 'DD-MM-YYYY'),
'NoEncryption', 100, 0,0), 'NoEncryption', 0,0);
```

4.6.2.3 8-BYTE AND 16-BYTE DECIMAL UDFs

These UDFs work on DECIMAL data types that are either 8 or 16 bytes in size. 8-byte DECIMALs have a precision between 10 and 18 digits, while 16-byte DECIMALs have a precision between 19 and 38 digits. These UDFs apply to No Encryption data elements only.

4.6.2.3.1 PTY_DECIMAL<n>INS

This UDF protects the decimal value with a No Encryption data element.

pty_decimal<n>ins(col DECIMAL<M,N>, dataelement VARCHAR, resultlen INTEGER, communicationid INTEGER, scid INTEGER)

Parameters

Name	Type	Description
col	DECIMAL(m,n)	Specifies the data to protect.
dataelement	VARCHAR	Specifies the name of the data element.
resultlen	INTEGER	Specifies the length of the buffer to hold the result.
communicationid	INTEGER	Specifies the location where the UDF will find the policy. The value must be the same as configured in the *pepserver.cfg* file.
scid	INTEGER	Specifies the security coordinate ID. The parameter is no longer used. The value of the parameter must be set to zero.

Returns

Unprotected DECIMAL value.

Exception

If the user does not have protect access rights in the policy, then the UDF terminates with an error message explaining what went wrong.

Example

```
select PTY_DECIMAL37_1INS(26656.0, 'NoEncryption', 100, 0,0);
```

4.6.2.3.2 PTY_DECIMAL<n>SEL

This UDF unprotects the decimal value that is protected with a No Encryption data element.

pty_decimal<n>sel(col DECIMAL<M,N>, dataelement VARCHAR, communicationid INTEGER, SCID INTEGER)

Parameters

Name	Type	Description
col	DECIMAL(m,n)	Specifies the data to unprotect.
dataelement	VARCHAR	Specifies the name of the data element.
communicationid	INTEGER	Specifies the location where the UDF will find the policy. The value must be the same as configured in the *pepserver.cfg* file.
scid	INTEGER	Specifies the security coordinate ID. The parameter is no longer used. The value of the parameter must be set to zero.

Returns

Unprotected DECIMAL value.

Exception

If configured in policy and user does not have access, then the UDF terminates with an error message explaining what went wrong.

Example

```
select PTY_DECIMAL37_1SEL(PTY_DECIMAL37_1INS(26656.0, 'NoEncryption', 100, 0,0),
'NoEncryption', 0,0);
```

4.6.2.3.3 PTY_DECIMAL<n>SELEX

This UDF unprotects the decimal value that is protected by a No Encryption data element and returns an error instead of NULL if the user does not have access.

pty_decimal<n>selex(col DECIMAL(m,n), dataelement VARCHAR, communicationid INTEGER, scid INTEGER)

Parameters

Name	Type	Description
col	DECIMAL(m,n)	Specifies the data to unprotect.
dataelement	VARCHAR	Specifies the name of the data element.
communicationid	INTEGER	Specifies the location where the UDF will find the policy. The value must be the same as configured in the *pepserver.cfg* file.
scid	INTEGER	Specifies the security coordinate ID. The parameter is no longer used. The value of the parameter must be set to zero.

Returns

Unprotected DECIMAL value.

Exception

If the user does not have access rights in the policy, then the UDF terminates with an error message.

Example

```
Select PTY_BIGINTSELEX(PTY_BIGINTINS(12345678, 'TE_INT_8, 100, 0, 0), 'TE_INT_8, 0, 0);
```

4.7 Trino Protector User Defined Functions

This section provides a detailed list of User Defined Functions (UDFs) for general information, and protection and unprotection of different data types.

4.7.1 General UDFs

This section includes list of general UDFs that can be used to retrieve the Trino Protector version and the current user.

4.7.1.1 ptyWhoAmI()

This function returns the name of the user.

ptyWhoAmI()

Parameters

None

Result

Name of user logged in to the database as VARCHAR.

Example

```
SELECT ptyWhoAmI();
```

4.7.1.2 ptyGetVersion()

This UDF returns the current version of the PEP server.

ptyGetVersion()

Parameters

None

Result

This UDF returns the current version of the PEP server.

Example

```
select ptyGetVersion();
```

4.7.2 VARCHAR UDFs

This section provides a list of Varchar UDFs for the protect, unprotect, and reprotect operations.

> **Note:** Consider a Trino session where you impersonate a user using the ***--user*** parameter as shown in the following example.
>
> ```
> ./TrinoCLI --server localhost:8080 --catalog hive --schema default --user=<sample_user>
> ```
>
> If you execute any UDF after impersonating a user, then the query execution happens for the impersonated user ***<sample_user>***. This is a limitation of Trino.

4.7.2.1 ptyProtectStr()

This UDF protects the *varchar* values.

ptyProtectStr(varchar input, varchar dataElement)

Parameters

`varchar input`: The *varchar* value to protect.

`varchar dataElement`: Name of the data element to protect *varchar* value.

Result

This UDF returns the protected *varchar* value.

Example

```
select PtyProtectStr('ProtegrityProt','Varchar_DE');
```

Table 4-3: Supported Protection Methods

Trino UDFs	Supported Protection Methods					
	Tokenization	**Encryption**	**FPE**	**No Encryption**	**Masking**	**Monitoring**
ptyProtectStr()	• Numeric (0-9) • Credit Card • Alpha • Upper Case Alpha • Alpha Numeric	No	Yes	Yes	Yes	Yes

Trino UDFs	Supported Protection Methods					
	Tokenization	Encryption	FPE	No Encryption	Masking	Monitoring
	• Upper Alpha Numeric • Lower ASCII • Datetime (YYYY-MM-DD HH:MM:SS) • Date (YYYY-MM-DD, DD/MM/YYYY, MM.DD.YYYY) • Decimal • Email • Unicode (Legacy) • Unicode (Base64 - Encoded Byte's Charset should match Dataelement's Encoding Type) • Unicode (Gen2)					

Note: The protection methods, that are not mentioned in the *Supported Protection Methods* table, are not supported.

4.7.2.2 ptyUnprotectStr()

This UDF unprotects the existing protected *varchar* value.

ptyUnprotectStr(varchar input, varchar dataElement)

Parameters

`varchar input:` The protected *varchar* value to unprotect.

`varchar dataElement:` Name of the data element to unprotect the *varchar* value.

Result

This UDF returns the unprotected *varchar* value.

Example

```
select PtyUnProtectStr(PtyProtectStr('ProtegrityProt','Varchar_DE'),'Varchar_DE');
```

Table 4-4: Supported Protection Methods

Trino UDFs	Supported Protection Methods					
	Tokenization	Encryption	FPE	No Encryption	Masking	Monitoring
ptyUnprotectStr()	• Numeric (0-9) • Credit Card • Alpha • Upper Case Alpha • Alpha Numeric • Upper Alpha Numeric	No	Yes	Yes	Yes	Yes

Trino UDFs	Supported Protection Methods					
	Tokenization	Encryption	FPE	No Encryption	Masking	Monitoring
	• Lower ASCII • Datetime (YYYY-MM-DD HH:MM:SS) • Date (YYYY-MM-DD, DD/MM/YYYY, MM.DD.YYYY) • Decimal • Email • Unicode (Legacy) • Unicode (Base64 - Encoded Byte's Charset should match Dataelement's Encoding Type) • Unicode (Gen2)					

Note: The protection methods, that are not mentioned in the *Supported Protection Methods* table, are not supported.

4.7.2.3 ptyReprotect()

This UDF reprotects the *varchar* protected data, which was earlier protected using the *ptyProtectStr* UDF, with a different data element.

ptyReprotect(varchar input, varchar oldDataElement, varchar newDataElement)

Parameters

`varchar input:` The *varchar* value to reprotect.

`varchar oldDataElement:` Name of the data element used to protect the data earlier.

`varchar newDataElement:` Name of the new data element to reprotect the data.

Result

This UDF returns the protected *varchar* value.

Example

```
select
ptyReprotect(PtyProtectStr('ProtegrityProt','Varchar_DE'),'Varchar_DE','new_Varchar_DE')
;
```

Table 4-5: Supported Protection Methods

Trino UDFs	Supported Protection Methods					
	Tokenization	Encryption	FPE	No Encryption	Masking	Monitoring
ptyReprotect()	• Numeric (0-9) • Credit Card • Alpha • Upper Case Alpha • Alpha Numeric	No	Yes	Yes	Yes	Yes

Trino UDFs	Supported Protection Methods					
	Tokenization	Encryption	FPE	No Encryption	Masking	Monitoring
	• Upper Alpha Numeric • Lower ASCII • Datetime (YYYY-MM-DD HH:MM:SS) • Date (YYYY-MM-DD, DD/MM/YYYY, MM.DD.YYYY) • Decimal • Email • Unicode (Legacy) • Unicode (Base64 - Encoded Byte's Charset should match Dataelement's Encoding Type) • Unicode (Gen2)					

Note: The protection methods, that are not mentioned in the *Supported Protection Methods* table, are not supported.

4.7.3 BIGINT UDFs

This section provides a list of *BigInt UDFs* for the protect, unprotect, and reprotect operations.

Note: Consider a Trino session where you impersonate a user using the **`--user`** parameter as shown in the following example.

```
./TrinoCLI --server localhost:8080 --catalog hive --schema default --user=<sample_user>
```

If you execute any UDF after impersonating a user, then the query execution happens for the impersonated user **`<sample_user>`**. This is a limitation of Trino.

4.7.3.1 ptyProtectBigInt()

This UDF protects the *BigInt* value.

ptyProtectBigInt(bigint input, varchar dataElement)

Parameters

`bigint input:` The value to protect.

`varchar dataElement:` Name of the data element to protect the value.

Result

This UDF returns the protected *BigInt* value.

Example

```
select PtyProtectBigInt(1234567, 'BigInt_DE');
```

Table 4-6: Supported Protection Methods

Trino UDFs	Supported Protection Methods					
	Tokenization	Encryption	FPE	No Encryption	Masking	Monitoring
ptyProtectBigInt()	Integer 8 Bytes	No	No	Yes	No	Yes

Note: The protection methods, that are not mentioned in the *Supported Protection Methods* table, are not supported.

4.7.3.2 ptyUnprotectBigInt()

This UDF unprotects the protected *BigInt* value.

ptyProtectBigInt(bigint input, varchar dataElement)

Parameters

`bigint input:` The protected value to unprotect.

`varchar dataElement:` Name of the data element to unprotect the value.

Result

This UDF returns the unprotected *BigInt* value.

Example

```
select PtyUnProtectBigInt(PtyProtectBigInt(1234567, 'BigInt_DE'), 'BigInt_DE');
```

Table 4-7: Supported Protection Methods

Trino UDFs	Supported Protection Methods					
	Tokenization	Encryption	FPE	No Encryption	Masking	Monitoring
ptyUnprotectBigInt()	Integer 8 Bytes	No	No	Yes	No	Yes

Note: The protection methods, that are not mentioned in the *Supported Protection Methods* table, are not supported.

4.7.3.3 ptyReprotect()

This UDF reprotects the *BigInt* format protected data with a different data element.

Note: If you are using numeric data with the **`ptyReprotect()`** UDF for protection, then ensure that you cast the data to **`BigInt`** before using the UDF.

ptyReprotect(bigint input, varchar oldDataElement, varchar newDataElement)

Parameters

`bigint input:` The *BigInt* value to reprotect.

`varchar oldDataElement:` Name of the data element used to protect the data earlier.

`varchar newDataElement:` Name of the new data element to reprotect the data.

Result

This UDF returns the protected *BigInt* value.

Example

```
select ptyReprotect(PtyProtectBigInt(123456, 'BigInt_DE'),'BigInt_DE','new_BigInt_DE');
```

Table 4-8: Supported Protection Methods

Trino UDFs	Supported Protection Methods					
	Tokenization	Encryption	FPE	No Encryption	Masking	Monitoring
ptyReprotect()	Integer 8 Bytes	No	No	Yes	No	Yes

Note: The protection methods, that are not mentioned in the *Supported Protection Methods* table, are not supported.

4.7.4 SMALLINT UDFs

This section provides a list of *SmallInt* UDFs for the protect, unprotect, and reprotect operations.

Note: Consider a Trino session where you impersonate a user using the **`--user`** parameter as shown in the following example.

```
./TrinoCLI --server localhost:8080 --catalog hive --schema default --user=<sample_user>
```

If you execute any UDF after impersonating a user, then the query execution happens for the impersonated user **`<sample_user>`**. This is a limitation of Trino.

4.7.4.1 ptyProtectSmallInt()

This UDF protects the *SmallInt* values.

ptyProtectSmallInt(smallint input, varchar dataElement)

Parameters

`smallint input:` The *SmallInt* value to protect.

`varchar dataElement:` Name of the data element to protect the *SmallInt* value.

Result

This UDF returns the protected *SmallInt* value.

Example

```
select ptyProtectSmallInt(cast(12 as smallint), 'SmallInt_DE');
```

Table 4-9: Supported Protection Methods

Trino UDFs	Supported Protection Methods					
	Tokenization	Encryption	FPE	No Encryption	Masking	Monitoring
ptyProtectSmallInt()	Integer 2 Bytes	No	No	Yes	No	Yes

Note: The protection methods, that are not mentioned in the *Supported Protection Methods* table, are not supported.

4.7.4.2 ptyUnprotectSmallInt()

This UDF unprotects the *SmallInt* values.

ptyUnprotectSmallInt(smallint input, varchar dataElement)

Parameters

`smallint input:` The *SmallInt* value to protect.

`varchar dataElement:` Name of the data element to protect *SmallInt* value.

Result

This UDF returns the unprotected *SmallInt* value.

Example

```
select PtyUnprotectSmallInt(PtyProtectSmallInt(cast(12 as smallint), 'SmallInt_DE'),
'SmallInt_DE');
```

Table 4-10: Supported Protection Methods

Trino UDFs	Supported Protection Methods					
	Tokenization	Encryption	FPE	No Encryption	Masking	Monitoring
ptyUnprotectSmallInt()	Integer 2 Bytes	No	No	Yes	No	Yes

Note: The protection methods, that are not mentioned in the *Supported Protection Methods* table, are not supported.

4.7.4.3 ptyReprotect()

This UDF reprotects the *SmallInt* format protected data, which was earlier protected using the ***ptyProtectSmallInt*** UDF, with a different data element.

ptyReprotect (smallInt input, varchar oldDataElement, varchar newDataElement)

Parameters

`smallint input:` The *SmallInt* value to protect.

`varchar oldDataElement:` Name of the data element used to protect the data earlier.

`varchar newDataElement:` Name of the new data element to reprotect the data.

Result

This UDF returns the protected *SmallInt* value.

Example

```
select ptyReprotect(PtyProtectSmallInt(cast(12 as smallint),
'SmallInt_DE'),'SmallInt_DE','new_SmallInt_DE');
```

Table 4-11: Supported Protection Methods

Trino UDFs	Supported Protection Methods					
	Tokenization	Encryption	FPE	No Encryption	Masking	Monitoring
ptyReprotect()	Integer 2 Bytes	No	No	Yes	No	Yes

Note: The protection methods, that are not mentioned in the *Supported Protection Methods* table, are not supported.

4.7.5 INT UDFs

This section provides a list of *Int* UDFs for the protect, unprotect, and reprotect operations.

Note: Consider a Trino session where you impersonate a user using the ***--user*** parameter as shown in the following example.

```
./TrinoCLI --server localhost:8080 --catalog hive --schema default --user=<sample_user>
```

If you execute any UDF after impersonating a user, then the query execution happens for the impersonated user ***<sample_user>***. This is a limitation of Trino.

4.7.5.1 ptyProtectInt()

This UDF protects the *Int* values.

ptyProtectInt(int input, varchar dataElement)

Parameters

`int input:` The *Int* value to protect.

`varchar dataElement:` Name of the data element to protect the *Int* value.

Result

This UDF returns the protected *Int* value.

Example

```
select ptyProtectInt(1234567, 'Int_DE');
```

Table 4-12: Supported Protection Methods

Trino UDFs	Supported Protection Methods					
	Tokenization	**Encryption**	**FPE**	**No Encryption**	**Masking**	**Monitoring**
ptyProtectInt()	Integer 4 Bytes	No	No	Yes	No	Yes

> **Note:** The protection methods, that are not mentioned in the *Supported Protection Methods* table, are not supported.

4.7.5.2 ptyUnprotectInt()

This UDF unprotects the existing protected integer value.

ptyUnprotectInt(int input, varchar dataElement)

Parameters

`int input:` The protected *Int* value to unprotect.

`varchar dataElement:` Name of the data element to unprotect the *Int* value.

Result

This UDF returns the unprotected *Int* value.

Example

```
select ptyUnprotectInt(ptyProtectInt(1234567, 'Int_DE'), 'Int_DE');
```

Table 4-13: Supported Protection Methods

Trino UDFs	Supported Protection Methods					
	Tokenization	**Encryption**	**FPE**	**No Encryption**	**Masking**	**Monitoring**
ptyUnprotectInt()	Integer 4 Bytes	No	No	Yes	No	Yes

> **Note:** The protection methods, that are not mentioned in the *Supported Protection Methods* table, are not supported.

4.7.5.3 ptyReprotect()

This UDF reprotects the *Int* format protected data, which was earlier protected using the ***`ptyProtectInt`*** UDF, with a different data element.

ptyReprotect(int input, varchar oldDataElement, varchar newDataElement)

Parameters

`int input:` The *Int* value to reprotect.

`varchar oldDataElement:` Name of the data element used to protect the data earlier.

`varchar newDataElement:` Name of the new data element to reprotect the data.

Result

This UDF returns the protected *Int* value.

Example

```
select ptyReprotect(ptyProtectInt(1234567, 'Token_Integer'),
'Token_Integer','new_Token_Integer');
```

Table 4-14: Supported Protection Methods

Trino UDFs	Supported Protection Methods					
	Tokenization	**Encryption**	**FPE**	**No Encryption**	**Masking**	**Monitoring**
ptyReprotect()	Integer 4 Bytes	No	No	Yes	No	Yes

Note: The protection methods, that are not mentioned in the *Supported Protection Methods* table, are not supported.

4.7.6 DATE UDFs

This section provides a list of *Date* UDFs for the protect, unprotect, and reprotect operations.

Note: Consider a Trino session where you impersonate a user using the **`--user`** parameter as shown in the following example.

```
./TrinoCLI --server localhost:8080 --catalog hive --schema default --user=<sample_user>
```

If you execute any UDF after impersonating a user, then the query execution happens for the impersonated user **`<sample_user>`**. This is a limitation of Trino.

Note: There are inconsistencies observed when Trino is used to fetch and store date values from HDFS, where data was stored using Hive. It is recommended to verify if the correct date and datetime values are retrieved when the data is fetched from or stored in HDFS without using the Trino UDFs. If the data consistency is maintained, only then go ahead with using the Trino Date or DateTime UDFs.

4.7.6.1 ptyProtectDate()

This UDF protects the *Date* format data, which is provided as input.

Note: In the Trino Protector, version 7.1 release, the supported date format is *YYYY-MM-DD* only.

ptyProtectDate(date input, varchar dataElement)

Parameters

`date input:` The *Date* format data, which needs to be protected.

`varchar dataElement:` The data element that will be used to protect the *Date* format data.

Result

This UDF returns the protected *Date* format value.

Example

```
select PtyProtectDate(cast('2018-10-10' as date), 'Date_DE');
```

Table 4-15: Supported Protection Methods

Trino UDFs	Supported Protection Methods					
	Tokenization	**Encryption**	**FPE**	**No Encryption**	**Masking**	**Monitoring**
ptyProtectDate()	Date	No	No	Yes	No	Yes

Note: The protection methods, that are not mentioned in the *Supported Protection Methods* table, are not supported.

4.7.6.2 ptyUnprotectDate()

This UDF unprotects the protected *Date* format data, which is provided as input.

> **Note:** In the Trino Protector, version 7.1 release, the supported date format is *YYYY-MM-DD* only.

ptyUnprotectDate(date input, varchar dataElement)

Parameters

`date input:` The protected *Date* format data, which is provided as input.

`varchar dataElement:` The data element that will be used to unprotect the *Date* format data.

Result

This UDF returns unprotected *Date* format value.

Example

```
select ptyUnprotectDate(PtyProtectDate(cast('2018-10-10' as date), 'Date_DE'),
'Date_DE');
```

Table 4-16: Supported Protection Methods

HIVE UDFs	Supported Protection Methods					
	Tokenization	Encryption	FPE	No Encryption	Masking	Monitoring
ptyUnprotectDate()	Date	No	No	Yes	No	Yes

> **Note:** The protection methods, that are not mentioned in the *Supported Protection Methods* table, are not supported.

4.7.6.3 ptyReprotect()

This UDF reprotects the *Date* format protected data, which was earlier protected using the **`ptyProtectDate`** UDF, with a different data element.

> **Note:** In the Trino Protector, version 7.1 release, the supported date format is *YYYY-MM-DD* only.

ptyReprotect(date input, varchar oldDataElement, varchar newDataElement)

Parameters

`date input:` The date format data, which needs to be reprotected.

`varchar oldDataElement:` The data element that was used to protect the data earlier.

`varchar newDataElement:` The new data that will be used to reprotect the data.

> **Warning:** Ensure that you use the data element with the *No Encryption* method only. Using any other data element might cause corruption of data.

Result

This UDF returns the protected *Date* format value.

Example

```
select PtyReprotect(cast('2018-10-10' as date), 'Date_DE', 'new_Date_DE');
```

Table 4-17: Supported Protection Methods

Trino UDFs	Supported Protection Methods					
	Tokenization	Encryption	FPE	No Encryption	Masking	Monitoring
ptyReprotect()	Date	No	No	Yes	No	Yes

Note: The protection methods, that are not mentioned in the *Supported Protection Methods* table, are not supported.

4.7.7 DATETIME UDFs

This section provides a list of *DateTime* UDFs for the protect, unprotect, and reprotect operations.

Note: Consider a Trino session where you impersonate a user using the **`--user`** parameter as shown in the following example.

```
./TrinoCLI --server localhost:8080 --catalog hive --schema default --user=<sample_user>
```

If you execute any UDF after impersonating a user, then the query execution happens for the impersonated user **`<sample_user>`**. This is a limitation of Trino.

Note: There are inconsistencies observed when Trino is used to fetch and store date values from HDFS, where data was stored using Hive. It is recommended to verify if the correct date and datetime values are retrieved when data is fetched from or stored in HDFS without using the Trino UDFs. If the data consistency is maintained, only then go ahead with using the Trino Date or DateTime UDFs.

4.7.7.1 ptyProtectDateTime()

This UDF protects the *TIMESTAMP* format data, which is provided as input.

Note: In the Trino Protector, version 7.1 release, the supported timestamp format is *YYYY-MM-DD HH:MM:SS*.

ptyProtectDateTime(timestamp input, varchar dataElement)

Parameters

`timestamp input:` The data in the *Timestamp* format, which needs to be protected.

`varchar dataElement:` The data element that will be used to protect the *Timestamp* format data.

Result

This UDF returns the protected *Timestamp* format value.

Example

```
select ptyProtectDateTime(cast('2018-10-10' as TIMESTAMP), 'DateTime_DE');
```

Table 4-18: Supported Protection Methods

Trino UDFs	Supported Protection Methods					
	Tokenization	Encryption	FPE	No Encryption	Masking	Monitoring
ptyProtectDateTime( )	Datetime	No	No	Yes	No	Yes

Note: The protection methods, that are not mentioned in the *Supported Protection Methods* table, are not supported.

4.7.7.2 ptyUnprotectDateTime()

This UDF unprotects the protected *Timestamp* format data, which is provided as input.

> **Note:** In the Trino Protector, version 7.1 release, the supported timestamp format is *YYYY-MM-DD HH:MM:SS*.

ptyUnprotectDateTime(timestamp input, varchar dataElement)

Parameters

`timestamp input:` The protected data in the *Timestamp* format, which needs to be unprotected.

`varchar dataElement:` The data element that will be used to unprotect the *Timestamp* format data.

Result

This UDF returns the unprotected *Timestamp* format value.

Example

```
select ptyUnprotectDateTime(ptyProtectDateTime(cast('2018-10-10 03:04:05' as TIMESTAMP),
'DateTime_DE'), 'DateTime_DE');
```

Table 4-19: Supported Protection Methods

Trino UDFs	Supported Protection Methods					
	Tokenization	**Encryption**	**FPE**	**No Encryption**	**Masking**	**Monitoring**
ptyUnprotectDateTime()	Datetime	No	No	Yes	No	Yes

> **Note:** The protection methods, that are not mentioned in the *Supported Protection Methods* table, are not supported.

4.7.7.3 ptyReprotect()

This UDF reprotects the *Timestamp* format protected data, which was earlier protected using the *ptyProtectDateTime* UDF, with a different data element.

> **Note:** In the Trino Protector, version 7.1 release, the supported timestamp format is *YYYY-MM-DD HH:MM:SS*.

ptyReprotect(timestamp input, varchar oldDataElement, varchar newDataElement)

Parameters

`timestamp input:` The data in the *Timestamp* format, which needs to be reprotected.

`varchar oldDataElement:` The data element that was used to protect the data earlier.

`varchar newDataElement:` The new data element that will be used to reprotect the data.

Result

This UDF returns the protected *Timestamp* format value.

Example

```
select ptyReprotect(ptyProtectDateTime(cast('2018-10-10 03:04:05' as TIMESTAMP),
'DateTime_DE'), 'DateTime_DE', 'new_DateTime_DE');
```

Table 4-20: Supported Protection Methods

Trino UDFs	Supported Protection Methods					
	Tokenization	Encryption	FPE	No Encryption	Masking	Monitoring
ptyReprotect()	Datetime	No	No	Yes	No	Yes

Note: The protection methods, that are not mentioned in the *Supported Protection Methods* table, are not supported.

4.7.8 VarChar Encryption UDFs

This section provides a list of VarChar encryption UDFs for the protect, unprotect, and reprotect operations.

Note: Consider a Trino session where you impersonate a user using the **`--user`** parameter as shown in the following example.

```
./TrinoCLI --server localhost:8080 --catalog hive --schema default --user=<sample_user>
```

If you execute any UDF after impersonating a user, then the query execution happens for the impersonated user **`<sample_user>`**. This is a limitation of Trino.

4.7.8.1 ptyStringEnc()

This UDF encrypts the *Varchar* value.

ptyStringEnc(varchar input, varchar DataElement)

Parameters

`varchar input:` The *Varchar* value to encrypt.

`varchar DataElement:` Name of the data element used to encrypt the *Varchar* value.

Warning:

- The string encryption UDFs are limited to accept 2 GB data size at maximum as input.
- Ensure that the field size for the protected binary data post the required encoding does not exceed the 2 GB input limit.

Warning:

- The field size to store the input data is dependent on the encryption algorithm selected, such as, AES-128, AES-256, 3DES, and CUSP, and the encoding type selected, such as, No Encoding, Base64, and Hex.
- Ensure that you set the input data size based on the required encryption algorithm and encoding so that the it does not exceed the 2 GB input limit.
- For more information about estimating the field size of the data, refer to the section *Guidelines for Estimating Field Size of Data* .

Result

This UDF returns the encrypted *Varbinary* value.

Example

```
select ptyStringEnc('ProtegrityProt','AES128_DE');
```

Exception

`ptyTrinoProtectorException: INPUT-ERROR: Tokenization or Format Preserving Data Elements are not supported`: An unsupported data element is provided.

`java.io.IOException: Too many bytes before newline: 2147483648`: The length of the input needs to be less than the maximum limit of 2 GB.

Table 4-21: Supported Protection Methods

Trino UDFs	Supported Protection Methods					
	Tokenization	**Encryption**	**FPE**	**No Encryption**	**Masking**	**Monitoring**
ptyStringEnc()	No	• AES-128 • AES-256 • 3DES • CUSP	No	Yes	No	Yes

Note: The protection methods, that are not mentioned in the *Supported Protection Methods* table, are not supported.

4.7.8.2 ptyStringDec()

This UDF decrypts the *Varbinary* value.

ptyStringDec(varbinary input, varchar DataElement)

Parameters

`varbinary input:` The protected *Varbinary* value to unprotect.

`varchar DataElement:` Name of the data element that was used to encrypt the *Varchar* value in the **`ptyStringEnc()`** UDF.

Result

This UDF returns the decrypted *Varchar* value.

Example

```
select ptyStringDec(ptyStringEnc('ProtegrityProt','AES128_DE'),'AES128_DE');
```

Exception

`ptyTrinoProtectorException: INPUT-ERROR: First argument (Input Data to be unprotected) is not a valid Binary Datatype`: The input data, which is not in binary format is provided.

`ptyHiveProtectorException: INPUT-ERROR: Tokenization or Format Preserving Data Elements are not supported`: An unsupported data element is provided.

Table 4-22: Supported Protection Methods

Trino UDFs	Supported Protection Methods					
	Tokenization	**Encryption**	**FPE**	**No Encryption**	**Masking**	**Monitoring**
ptyStringDec()	No	• AES-128 • AES-256 • 3DES • CUSP	No	Yes	No	Yes

Note: The protection methods, that are not mentioned in the *Supported Protection Methods* table, are not supported.

4.7.8.3 ptyStringReEnc()

This UDF reencrypts the *Varbinary* format encrypted data with a different data element.

ptyStringReEnc(varbinary input, varchar oldDataElement, varchar newDataElement)

Parameters

`varbinary input:` The *Varbinary* value to reencrypt.

varchar oldDataElement: Name of the data element used to encrypt the data earlier.

varchar newDataElement: Name of the new data element to reencrypt the data.

Result

This UDF returns the *Varbinary* format data, which is reencrypted.

Example

```
select
ptyStringReEnc(ptyStringEnc('ProtegrityProt','AES128_DE'),'AES128_DE','new_AES128_DE');
```

Exception

ptyTrinoProtectorException: INPUT-ERROR: First argument (Input Data to be reprotected) is not a valid Binary Datatype: The input data, which is not in binary format is provided.

java.io.IOException: Too many bytes before newline: 2147483648: The length of the input needs to be less than the maximum limit of 2 GB.

com.protegrity.hive.udf.ptyTrinoProtectorException: 26, Unsupported algorithm or unsupported action for the specific data element: An unsupported data element is provided.

Table 4-23: Supported Protection Methods

Trino UDFs	Supported Protection Methods					
	Tokenization	Encryption	FPE	No Encryption	Masking	Monitoring
ptyStringReEnc()	No	• AES-128 • AES-256 • 3DES • CUSP	No	Yes	No	Yes

Note: The protection methods, that are not mentioned in the *Supported Protection Methods* table, are not supported.

4.7.9 Unicode UDFs

This section provides a list of Unicode UDFs for the protect, unprotect, and reprotect operations.

Note: Consider a Trino session where you impersonate a user using the **--user** parameter as shown in the following example.

```
./TrinoCLI --server localhost:8080 --catalog hive --schema default --user=<sample_user>
```

If you execute any UDF after impersonating a user, then the query execution happens for the impersonated user **<sample_user>**. This is a limitation of Trino.

4.7.9.1 ptyProtectUnicode()

This UDF protects the *Varchar* (Unicode) values.

ptyProtectUnicode(varchar input, varchar dataElement)

Parameters

varchar input: The *Varchar* (Unicode) value to protect.

varchar dataElement: Name of the data element to protect the *Varchar* (Unicode) value.

Warning:

- This UDF should be used only if you need to tokenize Unicode data in Trino, and migrate the tokenized data from Trino to a Teradata database and detokenize the data using the Protegrity Database Protector.
- Ensure that you use this UDF with a Unicode tokenization data element only.

For more information about migrating tokenized Unicode data to a Teradata database, refer to the *Big Data Protector Guide 9.1.0.0*.

Result

This UDF returns the protected *Varchar* value.

Example

```
select ptyProtectUnicode('ProtegrityProt','Unicode_DE');
```

Table 4-24: Supported Protection Methods

Trino UDFs	Supported Protection Methods					
	Tokenization	**Encryption**	**FPE**	**No Encryption**	**Masking**	**Monitoring**
ptyProtectUnicode()	• Unicode (Legacy) • Unicode Base64	No	No	Yes	No	Yes

Note: The protection methods, that are not mentioned in the *Supported Protection Methods* table, are not supported.

4.7.9.2 ptyUnprotectUnicode()

This UDF unprotects the existing protected string value.

ptyUnprotectUnicode(varchar input, varchar dataElement)

Parameters

`varchar input:` The protected *Varchar* value to unprotect.

`varchar dataElement:` Name of the data element to unprotect *Varchar* value.

Warning:

- This UDF should be used only if you need to tokenize Unicode data in Trino, and migrate the tokenized data from Trino to a Teradata database and detokenize the data using the Protegrity Database Protector.
- Ensure that you use this UDF with a Unicode tokenization data element only.

  For more information about migrating tokenized Unicode data to a Teradata database, refer to the *Big Data Protector Guide 9.1.0.0*.

Result

This UDF returns the unprotected *Varchar* (Unicode) value.

Example

```
select
ptyUnprotectUnicode(ptyProtectUnicode('ProtegrityProt','Unicode_DE'),'Unicode_DE');
```

Table 4-25: Supported Protection Methods

Trino UDFs	Supported Protection Methods					
	Tokenization	**Encryption**	**FPE**	**No Encryption**	**Masking**	**Monitoring**
ptyUnprotectUnicode()	• Unicode (Legacy)	No	No	Yes	No	Yes

Trino UDFs	Supported Protection Methods					
	Tokenization	Encryption	FPE	No Encryption	Masking	Monitoring
	• Unicode Base64					

Note: The protection methods, that are not mentioned in the *Supported Protection Methods* table, are not supported.

4.7.9.3 ptyReprotectUnicode()

This UDF reprotects the *Varchar* format protected data, which was protected earlier using the *ptyProtectUnicode()* UDF, with a different data element.

ptyReprotectUnicode(varchar input, varchar oldDataElement, varchar newDataElement)

Parameters

`varchar input:` The *Varchar*(Unicode) value to reprotect.

`varchar oldDataElement:` Name of the data element used to protect the data earlier.

`varchar newDataElement:` Name of the new data element to reprotect the data.

Warning:

- This UDF should be used only if you need to tokenize Unicode data in Trino, and migrate the tokenized data from Trino to a Teradata database and detokenize the data using the Protegrity Database Protector.
- Ensure that you use this UDF with a Unicode tokenization data element only.

  For more information about migrating tokenized Unicode data to a Teradata database, refer to the *Big Data Protector Guide 9.1.0.0*.

Result

This UDF returns the protected *Varchar* value.

Example

```
select
ptyReprotectUnicode(ptyProtectUnicode('ProtegrityProt','Unicode_DE'),'Unicode_DE','new_U
nicode_DE');
```

Table 4-26: Supported Protection Methods

Trino UDFs	Supported Protection Methods					
	Tokenization	Encryption	FPE	No Encryption	Masking	Monitoring
ptyReprotectUnicode()	• Unicode (Legacy) • Unicode Base64	No	No	Yes	No	Yes

Note: The protection methods, that are not mentioned in the *Supported Protection Methods* table, are not supported.

4.7.10 Decimal UDFs

This section provides a list of *Decimal* UDFs for the protect, unprotect, and reprotect operations.

Note: Consider a Trino session where you impersonate a user using the **`--user`** parameter as shown in the following example.

```
./TrinoCLI --server localhost:8080 --catalog hive --schema default --user=<sample_user>
```

If you execute any UDF after impersonating a user, then the query execution happens for the impersonated user **<sample_user>**. This is a limitation of Trino.

4.7.10.1 ptyProtectDecimal()

This UDF protects the *Decimal* value.

ptyProtectDecimal(decimal input, varchar dataElement)

Parameters

decimal input: The *Decimal* value to protect.

varchar dataElement: Name of the data element to protect *Decimal* value.

Warning: Ensure that you use the data element with the *No Encryption* method only. Using any other data element might cause corruption of data.

Result

This UDF returns the protected *Decimal* value.

Note: If the input value is NULL and the data element is not a part of the policy, then the output value returned is NULL.

Example

```
select ptyProtectDecimal(12332212222223.033, 'NoEnc');
```

Table 4-27: Supported Protection Methods

Trino UDFs	Supported Protection Methods					
	Tokenization	Encryption	FPE	No Encryption	Masking	Monitoring
ptyProtectDecimal()	No	No	No	Yes	No	Yes

Note: The protection methods, that are not mentioned in the *Supported Protection Methods* table, are not supported.

4.7.10.2 ptyUnprotectDecimal()

This UDF unprotects the *Decimal* value.

ptyUnprotectDecimal(decimal input, varchar dataElement)

Parameters

decimal input: The *Decimal* value to protect.

varchar dataElement: Name of the data element to unprotect *Decimal* value.

Warning: Ensure that you use the data element with the *No Encryption* method only. Using any other data element might cause corruption of data.

Result

This UDF returns the unprotected *Decimal* value.

Note: If the input value is NULL and the data element is not a part of the policy, then the output value returned is NULL.

Example

```
select ptyUnprotectDecimal(12332212222223.033, 'NoEnc');
```

Table 4-28: Supported Protection Methods

Trino UDFs	Supported Protection Methods					
	Tokenization	Encryption	FPE	No Encryption	Masking	Monitoring
ptyUnprotectDecimal()	No	No	No	Yes	No	Yes

Note: The protection methods, that are not mentioned in the *Supported Protection Methods* table, are not supported.

4.7.10.3 ptyReprotect()

This UDF reprotects the *Decimal* format protected data with a different data element.

ptyReprotect(decimal input, varchar oldDataElement, varchar newDataElement)

Parameters

`decimal input:` The *Decimal* value to reprotect.

`varchar oldDataElement:` Name of the data element used to protect the data earlier.

`varchar newDataElement:` Name of the new data element to reprotect the data.

Warning: Ensure that you use the data element with the *No Encryption* method only. Using any other data element might cause corruption of data.

Result

This UDF returns the protected *Decimal* value.

Example

```
select ptyReprotect(12332212222223.033,'NoEnc','NoEnc');
```

Table 4-29: Supported Protection Methods

Trino UDFs	Supported Protection Methods					
	Tokenization	Encryption	FPE	No Encryption	Masking	Monitoring
ptyReprotect()	No	No	No	Yes	No	Yes

Note: The protection methods, that are not mentioned in the *Supported Protection Methods* table, are not supported.

4.7.11 Double UDFs

This section provides a list of *Double* UDFs for the protect, unprotect, and reprotect operations.

Note: Consider a Trino session where you impersonate a user using the **`--user`** parameter as shown in the following example.

```
./TrinoCLI --server localhost:8080 --catalog hive --schema default --user=<sample_user>
```

If you execute any UDF after impersonating a user, then the query execution happens for the impersonated user **`<sample_user>`**. This is a limitation of Trino.

4.7.11.1 ptyProtectDouble()

This UDF protects the *Double* values.

ptyProtectDouble(double input, varchar dataElement)

Parameters

`double input:` The *Double* value to protect.

`varchar dataElement:` Name of the data element to protect the *Double* value.

> **Warning:** Ensure that you use the data element with the *No Encryption* method only. Using any other data element might cause corruption of data.

> **Note:** It is an observed behavior with Trino that the UDF accepting double parameter also accepts decimal and integer parameter due to internal data type conversion.

Result

This UDF returns the protected *Double* value.

Example

```
select ptyProtectDouble(12345, 'No_Enc');
```

Table 4-30: Supported Protection Methods

Trino UDFs	Supported Protection Methods					
	Tokenization	**Encryption**	**FPE**	**No Encryption**	**Masking**	**Monitoring**
ptyProtectDouble()	No	No	No	Yes	No	Yes

> **Note:** The protection methods, that are not mentioned in the *Supported Protection Methods* table, are not supported.

4.7.11.2 ptyUnprotectDouble()

This UDF unprotects the protected *Double* values.

ptyUnprotectDouble(double input, varchar dataElement)

Parameters

`double input:` The *Double* value to unprotect.

`varchar dataElement:` Name of the data element to unprotect the *Double* value.

> **Warning:** Ensure that you use the data element with the *No Encryption* method only. Using any other data element might cause corruption of data.

Result

This UDF returns the original *Double* value.

Example

```
select ptyUnprotectDouble(12345, 'No_Enc');
```

Table 4-31: Supported Protection Methods

Trino UDFs	Supported Protection Methods					
	Tokenization	**Encryption**	**FPE**	**No Encryption**	**Masking**	**Monitoring**
ptyUnprotectDouble()	No	No	No	Yes	No	Yes

> **Note:** The protection methods, that are not mentioned in the *Supported Protection Methods* table, are not supported.

4.7.11.3 ptyReprotect() - Double data

This UDF reprotects the protected *Double* format data with a different data element.

ptyReprotect(double input, varchar oldDataElement, varchar newDataElement)

Parameters

`double input:` The *Double* value to reprotect.

`varchar oldDataElement:` Name of the data element used to protect the *Double* data earlier.

`varchar newDataElement:` Name of the new data element to protect the *Double* data.

> **Warning:** Ensure that you use the data element with the *No Encryption* method only. Using any other data element might cause corruption of data.

Result

This UDF returns the protected *Double* value.

Example

```
select ptyReprotect(09457, 'No_Enc', 'new_No_Enc');
```

Table 4-32: Supported Protection Methods

Trino UDFs	Supported Protection Methods					
	Tokenization	Encryption	FPE	No Encryption	Masking	Monitoring
ptyReprotect()	No	No	No	Yes	No	Yes

> **Note:** The protection methods, that are not mentioned in the *Supported Protection Methods* table, are not supported.

4.7.12 VarBinary Encryption UDFs

In the Database Trino Protector on the Trino environment, the Trino Encryption UDFs can be used to create permanent functions.

4.7.12.1 ptyBinaryEnc()

This UDF protects the *VarBinary* values.

> **Note:** If the image file size exceeds 32 MB, then set the configuration in the *configuration.properties* file in the Trino server and then restart the Trino server on all the nodes (co-ordinator and worker).
>
> ```
> node-manager.http-client.max-content-length=64MB
> exchange.http-client.max-content-length=64MB
> ```

ptyBinaryEnc(VarBinary input, Varchar DataElement)

Parameters

`VarBinary input:` The *VarBinary* value to encrypt.

`Varchar DataElement:` Name of the data element to encrypt the *VarBinary* value.

Result

This UDF returns the encrypted *VarBinary* value.

Example

```
select ptyBinaryEnc(X'12A23D43', 'AES256');
select ptyBinaryEnc(binary_col1, 'AES256') from table1;
```

Table 4-33: Supported Protection Methods

Trino UDFs	Supported Protection Methods					
	Tokenization	**Encryption**	**FPE**	**No Encryption**	**Masking**	**Monitoring**
ptyBinaryEnc()	No	• AES-128 • AES-256 • 3DES • CUSP	No	Yes	No	Yes

Note: The protection methods, that are not mentioned in the *Supported Protection Methods* table, are not supported.

4.7.12.2 ptyBinaryDec()

This UDF returns the decrypted *VarBinary* value.

ptyBinaryDec(VarBinary input, Varchar DataElement)

Parameters

`VarBinary input:` The *VarBinary* value to unprotect.

`Varchar DataElement:` Name of the data element to decrypt the *VarBinary* value.

Result

This UDF returns the decrypted *VarBinary* value.

Example

```
select ptyBinaryDec(X'215b807cdfbc', 'AES256');
select ptyBinaryDec(binary_col1, 'AES256') from table1;
```

Table 4-34: Supported Protection Methods

Trino UDFs	Supported Protection Methods					
	Tokenization	**Encryption**	**FPE**	**No Encryption**	**Masking**	**Monitoring**
ptyBinaryDec()	No	• AES-128 • AES-256 • 3DES • CUSP	No	Yes	No	Yes

Note: The protection methods, that are not mentioned in the *Supported Protection Methods* table, are not supported.

4.7.12.3 ptyBinaryReEnc()

This UDF returns the *VarBinary* format data, which is re-encrypted.

ptyBinaryReEnc(VarBinary input, Varchar oldDataElement, Varchar newDataElement)

Parameters

`VarBinary input:` The *VarBinary* value to re-encrypt.

`Varchar oldDataElement:` Name of the data element that was used to encrypt the data.

`Varchar newDataElement:` Name of the new data element to re-encrypt the data.

Result

This UDF returns the *VarBinary* format data, which is re-encrypted.

Note: Tokenization or Format Preserving Data Elements are not supported. Only encryption data elements are supported.

Example

```
select ptyBinaryReEnc(X'12A23D43', 'Old_AES256', 'New_AES256');
select ptyBinaryReEnc(binary_col1, 'Old_AES256', 'New_AES256') from table1;
```

Table 4-35: Supported Protection Methods

Trino UDFs	Supported Protection Methods					
	Tokenization	**Encryption**	**FPE**	**No Encryption**	**Masking**	**Monitoring**
ptyBinaryReEnc()	No	• AES-128 • AES-256 • 3DES • CUSP	No	Yes	No	Yes

Note: The protection methods, that are not mentioned in the *Supported Protection Methods* table, are not supported.

Chapter 5

z/OS Protector UDFs

This section describes all the Protegrity UDFs that are available for Mainframe z/OS.

> **Note:**
> To reduce performance issues that occur due to protection of data or casting of data, a general best practice is to protect the data and present the unprotect APIs, UDFs, or Commands, as applicable, to authorized users only. This eliminates access of the unauthorized users to the unprotection APIs, UDFs, or Commands as the data is in protected form only.
>
> The unprotection of protected data is therefore limited to authorized users and does not cause a significant performance impact as the APIs, UDFs, or Commands are executed restrictively.

5.1 General UDFs

This section lists the general UDFs including syntax.

5.1.1 pty.whoami

This function returns the name of the user who is currently logged in.

The external name is PDWHOAMI.

pty.whoami()

Parameters

None

Returns

This UDF returns the name of user logged in to the database as VARCHAR(256).

Example

```
SELECT PTY.whoami() from SYSIBM.SYSDUMMY1;
```

5.1.2 pty.getversion

This function returns the version of the product.

The external name is PDGETVER.

pty.getversion()

Parameters

None

Returns

This UDF returns the version of the product as VARCHAR(256).

Example

```
SELECT PTY.getversion() from SYSIBM.SYSDUMMY1;
```

5.1.3 pty.getcurrentkeyid

This function returns the current key ID for a data element. It is typically used together with **getkeyid** to determine if some data is protected with the most recent key for a given data element.

The external name is PDGCKEY.

pty.getcurrentkeyid(communicationid INTEGER, dataelement VARCHAR)

Parameters

Name	Type	Description
communicationid	INTEGER	Specifies location where policy can be found. Must be the same as configured in `pepserver.cfg`
dataelement	VARCHAR(64)	Specifies the name of data element.

Returns

This UDF returns the current key ID as INTEGER.

Example

```
SELECT PTY.getcurrentkeyid(8,'data-element name') from SYSIBM.SYSDUMMY1;
```

5.1.4 pty.getkeyid

This function returns the current key ID that was used to protect an item of data. It is typically used together with getcurrentkeyid to determine if some data is protected with the most recent key for a given data element.

The external name is PDGKEY.

pty.getkeyid(communicationid INTEGER, dataelement VARCHAR, data VARCHAR FOR BIT DATA)

Parameters

Name	Type	Description
communicationid	INTEGER	Specifies location where policy can be found. Must be the same as configured in `pepserver.cfg`
dataelement	VARCHAR(64)	Specifies the name of data element.
data	VARCHAR(32672) FOR BIT DATA	Specifies the data that has been encrypted using a key ID.

Returns

This UDF returns the key ID as INTEGER.

Example

```
SELECT PTY.getkeyid(8,'data-element name',pty.ins_enc_varchar (8,'data-element name' ,
'abc123456',0) ) from SYSIBM.SYSDUMMY1;
```

5.2 Access Check UDFs

These UDFs can be used to check access permissions. The procedures will pass if user has access, otherwise it will cast an exception with the reason for failure.

5.2.1 pty.have_sel_perm

This function is used to determine if the user has select access to a data element.

The external name is PDHSPERM.

pty.have_sel_perm(communicationid INTEGER, dataelement VARCHAR)

Parameters

Name	Type	Description
communicationid	INTEGER	Specifies location where policy can be found. Must be the same as configured in `pepserver.cfg`.
dataelement	VARCHAR(64)	Specifies the name of data element.

Returns

If the user has select access, the UDF returns an INTEGER value (1).

> **Note:** If the user does not have the select access, this UDF throws an exception with an appropriate message.

Example

```
SELECT pty.have_sel_perm(8,'data-element name') FROM SYSIBM.SYSDUMMY1;
```

5.2.2 pty.have_upd_perm

This function is used to determine if the user has update access to a data element.

The external name is PDHUPERM.

pty.have_upd_perm(communicationid INTEGER, dataelement VARCHAR)

Parameters

Name	Type	Description
communicationid	INTEGER	Specifies location where policy can be found. Must be the same as configured in `pepserver.cfg`.
dataelement	VARCHAR(64)	Specifies the name of data element.

Returns

If the user has update access, the UDF returns an INTEGER value (1).

> **Note:** If the user does not have the update access, this UDF throws an exception with an appropriate message.

Example

```
SELECT pty.have_upd_perm(8,'data-element name') FROM SYSIBM.SYSDUMMY1;
```

5.2.3 pty.have_ins_perm

This function is used to determine if the user has insert access to a data element.

The external name is PDHIPERM.

pty.have_ins_perm(communicationid INTEGER, dataelement VARCHAR)

Parameters

Name	Type	Description
communicationid	INTEGER	Specifies location where policy can be found. Must be the same as configured in `pepserver.cfg`.
dataelement	VARCHAR(64)	Specifies the name of data element.

Returns

If the user has insert access, the UDF returns an INTEGER value (1).

> **Note:** If the user does not have the insert access, this UDF throws an exception with an appropriate message.

Example

```
SELECT pty.have_ins_perm(8,'data-element name') FROM SYSIBM.SYSDUMMY1;
```

5.2.4 pty.have_del_perm

This function is used to determine if the user has delete access to a data element.

The external name is PDHDPERM.

pty.have_del_perm(communicationid INTEGER, dataelement VARCHAR, scid INTEGER)

Parameters

Name	Type	Description
communicationid	INTEGER	Specifies location where policy can be found. Must be the same as configured in `pepserver.cfg`.
dataelement	VARCHAR(64)	Specifies the name of data element.
scid	INTEGER	Specifies the security co-ordinate ID, which is not used. The value should be set to zero.

Returns

If the user has delete access, the UDF returns an INTEGER value (1).

> **Note:** If the user does not have the delete access, this UDF throws an exception with an appropriate message.

Example

```
SELECT pty.have_del_perm(8,'data-element name',0) FROM SYSIBM.SYSDUMMY1;
```

5.2.5 pty.del_check

This function is used to determine if the user has delete access to a data element.

The external name is PDDCHECK.

pty.del_check(communicationid INTEGER, dataelement VARCHAR, scid INTEGER)

Parameters

Name	Type	Description
communicationid	INTEGER	Specifies location where policy can be found. Must be the same as configured in `pepserver.cfg`.
dataelement	VARCHAR(64)	Specifies the name of data element.
scid	INTEGER	Specifies the security co-ordinate ID, which is not used. The value should be set to zero.

Returns

If the user has delete access, the UDF returns an INTEGER value (1).

> **Note:** If the user does not have the delete access, this UDF throws an exception with an appropriate message.

Example

```
SELECT pty. del_check(8,'data-element name',0) FROM SYSIBM.SYSDUMMY1;
```

5.3 VARCHAR UDFs

These UDFs can be used to encrypt or decrypt VARCHAR data.

> **Note:** z/OS UDFs do not support LONG VARCHAR data type.

5.3.1 pty.ins_enc_varchar

This function is used to encrypt data with a data element.

The external name is PDIVCHR.

pty.ins_enc_varchar(communicationid INTEGER, dataelement VARCHAR, input_data VARCHAR, scid INTEGER)

Parameters

Name	Type	Description
communicationid	INTEGER	Specifies the location where policy can be found. Must be the same as configured in `pepserver.cfg`.
dataelement	VARCHAR(64)	Specifies the name of data element.
input_data	VARCHAR(32672)	Specifies the input data for UDF.
scid	INTEGER	Specifies the security co-ordinate ID, which is not used. The value should be set to zero.

Returns

This UDF returns the encrypted data as VARCHAR(32672) FOR BIT DATA.

> **Note:** If the function call fails, this UDF throws an exception with an appropriate message.

Example

```
CREATE TABLE table-name (CCN VARCHAR(20) FOR BIT DATA)
CCSID EBCDIC IN DATABASE DSNDB04;
INSERT into table-name (CCN)
VALUES (PTY.ins_enc_varchar(8,'data-element','4234567890123456',0));
```

5.3.2 pty.upd_enc_varchar

This function is used to encrypt data with a data element.

The external name is PDUVCHR.

pty.upd_enc_varchar(communicationid INTEGER, dataelement VARCHAR, input_data VARCHAR, scid INTEGER)

Parameters

Name	Type	Description
communicationid	INTEGER	Specifies the location where policy can be found. Must be the same as configured in `pepserver.cfg`.

Name	Type	Description
dataelement	VARCHAR(64)	Specifies the name of data element.
input_data	VARCHAR(32672)	Specifies the input data for UDF.
scid	INTEGER	Specifies the security co-ordinate ID, which is not used. The value should be set to zero.

Returns

This UDF returns the encrypted data as VARCHAR(32672) FOR BIT DATA.

> **Note:** If the function call fails, this UDF throws an exception with an appropriate message.

Example

```
UPDATE table-name SET (CCN) =
PTY.upd_enc_varchar(8,'data-element','4234567890123456',0);
```

5.3.3 pty.sel_dec_varchar

This function is used to decrypt data with a data element.

The external name is PDSVCHR.

pty.sel_dec_varchar(communicationid INTEGER, dataelement VARCHAR, input_data VARCHAR FOR BIT DATA, scid INTEGER)

Parameters

Name	Type	Description
communicationid	INTEGER	Specifies the location where policy can be found. Must be the same as configured in `pepserver.cfg`.
dataelement	VARCHAR(64)	Specifies the name of data element
input_data	VARCHAR(32672) FOR BIT DATA	Specifies the input data for UDF.
scid	INTEGER	Specifies the security co-ordinate ID, which is not used. The value should be set to zero.

Returns

This UDF returns the decrypted data as VARCHAR(32672).

> **Note:** If the function call fails, this UDF throws an exception with an appropriate message.

Example

```
SELECT pty.sel_dec_varchar(8,'data-element',CCN,0) as CCN
from table-name;
```

5.3.4 pty.ins_varchar

This function is used for no encryption with a data element as well as for tokenization.

The external name is PDIVCHRM.

pty.ins_varchar(communicationid INTEGER, dataelement VARCHAR, input_data VARCHAR, scid INTEGER)

Parameters

Name	Type	Description
communicationid	INTEGER	Specifies the location where policy can be found. Must be the same as configured in `pepserver.cfg`.
dataelement	VARCHAR(64)	Specifies the name of data element.
input_data	VARCHAR(32672)	Specifies the input data for UDF.
scid	INTEGER	Specifies the security co-ordinate ID, which is not used. The value should be set to zero.

Returns

This UDF returns the encrypted data as VARCHAR(32672).

> **Note:** If the function call fails, this UDF throws an exception with an appropriate message.

Example

```
CREATE TABLE table-name (CCN VARCHAR(20))
CCSID EBCDIC IN DATABASE DSNDB04;

INSERT into table-name (CCN)
VALUES (PTY.ins_varchar(8,’token-element','4234567890123456',0));
```

5.3.5 pty.upd_varchar

This function is used for no encryption with a data element as well as for tokenization.

The external name is PDUVCHRM.

pty.upd_varchar(communicationid INTEGER, dataelement VARCHAR, input_data VARCHAR, scid INTEGER)

Parameters

Name	Type	Description
communicationid	INTEGER	Specifies location where policy can be found. Must be the same as configured in `pepserver.cfg`
dataelement	VARCHAR(64)	Specifies the name of data element.
input_data	VARCHAR(32672)	Specifies the input data for UDF.

Name	Type	Description
scid	INTEGER	Specifies the security co-ordinate ID, which is not used. The value should be set to zero.

Returns

This UDF returns the encrypted data as VARCHAR(32672).

> **Note:** If the function call fails, this UDF throws an exception with an appropriate message.

Example

```
UPDATE table-name SET (CCN) =
PTY.upd_varchar(8,'token-element','4234567890123456',0);
```

5.3.6 pty.sel_varchar

This function is used for no encryption with a data element as well as for de-tokenization.

The external name is PDSVCHRM.

pty.sel_varchar(communicationid INTEGER, dataelement VARCHAR, input_data VARCHAR, scid INTEGER)

Parameters

Name	Type	Description
communicationid	INTEGER	Specifies the location where policy can be found. Must be the same as configured in `pepserver.cfg`.
dataelement	VARCHAR(64)	Specifies the name of data element.
input_data	VARCHAR(32672)	Specifies the input data for UDF.
scid	INTEGER	Specifies the security co-ordinate ID, which is not used. The value should be set to zero.

Returns

This UDF returns the decrypted data as VARCHAR(32672).

> **Note:** If the function call fails, this UDF throws an exception with an appropriate message.

Example

```
SELECT (PTY.sel_varchar(8,'token-element',CCN,0) as CCN
   from table-name;
```

5.3.7 pty.ins_hash_varchar

This function uses the hash function to protect the data using a data element.

The external name is PDIVCHRH.

pty.ins_hash_varchar(communicationid INTEGER, dataelement VARCHAR, input_data VARCHAR, scid INTEGER)

Parameters

Name	Type	Description
communicationid	INTEGER	Specifies the location where policy can be found. Must be the same as configured in `pepserver.cfg`.
dataelement	VARCHAR(64)	Specifies the name of data element.
input_data	VARCHAR(32672)	Specifies the input data for UDF.
scid	INTEGER	Specifies the security co-ordinate ID, which is not used. The value should be set to zero.

Returns

This UDF returns the hash value as VARCHAR(32672) FOR BIT DATA.

> **Note:** If the function call fails, this UDF throws an exception with an appropriate message.

Example

```
CREATE TABLE table-name (CCN VARCHAR(20) FOR BIT DATA)
CCSID EBCDIC IN DATABASE DSNDB04;
INSERT into table-name (CCN)
VALUES (pty.ins_hash_varchar(10,'HASHING','567#$%JEEVES',0));
```

5.3.8 pty.upd_hash_varchar

This function uses the hash function to protect the data using a data element.

The external name is PDUVCHRH.

pty.upd_hash_varchar(communicationid INTEGER, dataelement VARCHAR, input_data VARCHAR, scid INTEGER)

Parameters

Name	Type	Description
communicationid	INTEGER	Specifies the location where policy can be found. Must be the same as configured in `pepserver.cfg`.
dataelement	VARCHAR(64)	Specifies the name of data element.
input_data	VARCHAR(32672)	Specifies the input data for UDF.
scid	INTEGER	Specifies the security co-ordinate ID, which is not used. The value should be set to zero.

Returns

This UDF returns the hash value as VARCHAR(32672) FOR BIT DATA.

> **Note:** If the function call fails, this UDF throws an exception with an appropriate message.

Example

```
UPDATE table-name SET (CCN) =
pty.upd_hash_varchar(10,'HASHING','HSHAIMN12345',0);
```

5.4 VARCHAR FOR BIT DATA UDFs

These UDFs can be used to encrypt or decrypt VARCHAR FOR BIT DATA.

5.4.1 pty.ins_enc_varcharfbd

This function is used to encrypt data using a data element.

The external name is PDIVCHRF.

pty.ins_enc_varcharfbd(communicationid INTEGER, dataelement VARCHAR, input_data VARCHAR FOR BIT DATA, scid INTEGER)

Parameters

Name	Type	Description
communicationid	INTEGER	Specifies the location where policy can be found. Must be the same as configured in `pepserver.cfg`.
dataelement	VARCHAR(64)	Specifies the name of data element.
input_data	VARCHAR(32672) FOR BIT DATA	Specifies the input data for UDF.
scid	INTEGER	Specifies the security co-ordinate ID, which is not used. The value should be set to zero.

Returns

This UDF returns the encrypted data as VARCHAR(32672) FOR BIT DATA.

> **Note:** If the function call fails, this UDF throws an exception with an appropriate message.

Example

```
CREATE TABLE table-name (CCN VARCHAR(20) FOR BIT DATA)
CCSID EBCDIC IN DATABASE DSNDB04;
INSERT into table-name (CCN)
VALUES (PTY.ins_enc_varcharfbd(8,'data-element','4234567890123456',0));
```

5.4.2 pty.upd_enc_varcharfbd

This function is used to encrypt data with a data element.

The external name is PDUVCHRF.

pty.upd_enc_varcharfbd(communicationid INTEGER, dataelement VARCHAR, input_data VARCHAR FOR BIT DATA, scid INTEGER)

Parameters

Name	Type	Description
communicationid	INTEGER	Specifies the location where policy can be found. Must be the same as configured in `pepserver.cfg`.
dataelement	VARCHAR(64)	Specifies the name of data element.
input_data	VARCHAR(32672) FOR BIT DATA	Specifies the input data for UDF.
scid	INTEGER	Specifies the security co-ordinate ID, which is not used. The value should be set to zero.

Returns

This UDF returns the encrypted data as VARCHAR(32672) FOR BIT DATA.

> **Note:** If the function call fails, this UDF throws an exception with an appropriate message.

Example

```
UPDATE table-name SET (CCN) =
PTY.upd_enc_varcharfbd(8,'data-element','4234567890123456',0);
```

5.4.3 pty.sel_dec_varcharfbd

This function is used to decrypt data with a data element.

The external name is PDSVCHRF.

pty.sel_dec_varcharfbd(communicationid INTEGER, dataelement VARCHAR, input_data VARCHAR FOR BIT DATA, scid INTEGER)

Parameters

Name	Type	Description
communicationid	INTEGER	Specifies the location where policy can be found. Must be the same as configured in `pepserver.cfg`.
dataelement	VARCHAR(64)	Specifies the name of data element.
input_data	VARCHAR(32672) FOR BIT DATA	Specifies the input data for UDF.
scid	INTEGER	Specifies the security co-ordinate ID, which is not used. The value should be set to zero.

Returns

This UDF returns the decrypted data as VARCHAR(32672) FOR BIT DATA.

> **Note:** If the function call fails, this UDF throws an exception with an appropriate message.

Example

```
SELECT pty.sel_dec_varcharfbd(8,'data-element',CCN,0) as CCN
from table-name;
```

5.4.4 pty.ins_varcharfbd

This function is used for no encryption with a data element as well as for tokenization.

The external name is PDIVCHF2

pty.ins_varcharfbd(communicationid INTEGER, dataelement VARCHAR, input_data VARCHAR FOR BIT DATA, scid INTEGER)

Parameters

Name	Type	Description
communicationid	INTEGER	Specifies the location where policy can be found. Must be the same as configured in `pepserver.cfg`.
dataelement	VARCHAR(64)	Specifies the name of data element.
input_data	VARCHAR(32672) FOR BIT DATA	Specifies the input data for UDF.
scid	INTEGER	Specifies the security co-ordinate ID, which is not used. The value should be set to zero.

Returns

This UDF returns the encrypted data as VARCHAR(32672) FOR BIT DATA.

> **Note:** If the function call fails, this UDF throws an exception with an appropriate message.

Example

```
CREATE TABLE table-name (CCN VARCHAR(20) FOR BIT DATA)
CCSID EBCDIC IN DATABASE DSNDB04;
INSERT into table-name (CCN)
VALUES (PTY.ins_varcharfbd(8,token-element','4234567890123456',0));
```

5.4.5 pty.upd_varcharfbd

This function is used for no encryption with a data element as well as for tokenization.

The external name is PDUVCHF2.

pty.upd_varcharfbd(communicationid INTEGER, dataelement VARCHAR, input_data VARCHAR FOR BIT DATA, scid INTEGER)

Parameters

Name	Type	Description
communicationid	INTEGER	Specifies the location where policy can be found. Must be the same as configured in `pepserver.cfg`.

Name	Type	Description
dataelement	VARCHAR(64)	Specifies the name of data element.
input_data	VARCHAR(32672) FOR BIT DATA	Specifies the input data for UDF.
scid	INTEGER	Specifies the security co-ordinate ID, which is not used. The value should be set to zero.

Returns

This UDF returns the encrypted data as VARCHAR(32672) FOR BIT DATA.

> **Note:** If the function call fails, this UDF throws an exception with an appropriate message.

Example

```
UPDATE table-name SET (CCN) =
PTY.upd_varcharfbd(8,'token-element','4234567890123456',0);
```

5.4.6 pty.sel_varcharfbd

This function is used for no encryption with a data element as well as for de-tokenization.

The external name is PDSVCHF2.

pty.sel_varcharfbd(communicationid INTEGER, dataelement VARCHAR, input_data VARCHAR FOR BIT DATA, scid INTEGER)

Parameters

Name	Type	Description
communicationid	INTEGER	Specifies the location where policy can be found. Must be the same as configured in `pepserver.cfg`.
dataelement	VARCHAR(64)	Specifies the name of data element.
input_data	VARCHAR(32672) FOR BIT DATA	Specifies the input data for UDF.
scid	INTEGER	Specifies the security co-ordinate ID, which is not used. The value should be set to zero.

Returns

This UDF returns the decrypted data as VARCHAR(32672) FOR BIT DATA.

> **Note:** If the function call fails, this UDF throws an exception with an appropriate message.

Example

```
SELECT pty.sel_varcharfbd(8,’token-element',CCN,0) as CCN
from table-name;
```

5.5 CHAR UDFs

These UDFs can be used to encrypt and decrypt CHAR data.

5.5.1 pty.ins_enc_char

This function is used to encrypt data with a data element.

The external name is PDIVCHR.

pty.ins_enc_char(communicationid INTEGER, dataelement VARCHAR, input_data VARCHAR, scid INTEGER)

Parameters

Name	Type	Description
communicationid	INTEGER	Specifies the location where policy can be found. Must be the same as configured in `pepserver.cfg`.
dataelement	VARCHAR(64)	Specifies the name of data element.
input_data	VARCHAR(254)	Specifies the input data for UDF.
scid	INTEGER	Specifies the security co-ordinate ID, which is not used. The value should be set to zero.

Returns

This UDF returns the encrypted data as VARCHAR(300) FOR BIT DATA.

> **Note:** If the function call fails, this UDF throws an exception with an appropriate message.

Example

```
CREATE TABLE table-name (CCN VARCHAR(20) FOR BIT DATA)
CCSID EBCDIC IN DATABASE DSNDB04;
INSERT into table-name (CCN)
   VALUES (PTY.ins_enc_char(8,'data-element','4234567890123456',0));
```

5.5.2 pty.upd_enc_char

This function is used to encrypt data with a data element.

The external name is PDUVCHR.

pty.upd_enc_char(communicationid INTEGER, dataelement VARCHAR, input_data VARCHAR, scid INTEGER)

Parameters

Name	Type	Description
communicationid	INTEGER	Specifies the location where policy can be found. Must be the same as configured in `pepserver.cfg`.
dataelement	VARCHAR(64)	Specifies the name of data element.

Name	Type	Description
input_data	VARCHAR(254)	Specifies the input data for UDF.
scid	INTEGER	Specifies the security co-ordinate ID, which is not used. The value should be set to zero.

Returns

This UDF returns the encrypted data as VARCHAR(300) FOR BIT DATA.

> **Note:** If the function call fails, this UDF throws an exception with an appropriate message.

Example

```
UPDATE table-name SET (CCN) =
   PTY.upd_enc_char(8,'data-element','4234567890123456',0);
```

5.5.3 pty.sel_dec_char

This function is used to decrypt data with a data element.

The external name is PDSVCHR.

pty.sel_dec_char(communicationid INTEGER, dataelement VARCHAR, input_data VARCHAR FOR BIT DATA, scid INTEGER)

Parameters

Name	Type	Description
communicationid	INTEGER	Specifies the location where policy can be found. Must be the same as configured in `pepserver.cfg`.
dataelement	VARCHAR(64)	Specifies the name of data element.
input_data	VARCHAR (300) FOR BIT DATA	Specifies the input data for UDF.
scid	INTEGER	Specifies the security co-ordinate ID, which is not used. The value should be set to zero.

Returns

This UDF returns the decrypted data as VARCHAR(254).

> **Note:** If the function call fails, this UDF throws an exception with an appropriate message.

Example

```
SELECT pty.sel_dec_char(8,'data-element',CCN,0) as CCN
from table-name;
```

5.6 CHAR FOR BIT DATA UDFs

These UDFs can be used to encrypt and decrypt CHAR FOR BIT DATA.

5.6.1 pty.ins_enc_charfbd

This function is used to encrypt data with a data element.

The external name is PDIVCHRF.

pty.ins_enc_charfbd(communicationid INTEGER, dataelement VARCHAR, input_data VARCHAR FOR BIT DATA, scid INTEGER)

Parameters

Name	Type	Description
communicationid	INTEGER	Specifies the location where policy can be found. Must be the same as configured in `pepserver.cfg`.
dataelement	VARCHAR(64)	Specifies the name of data element.
input_data	VARCHAR(254) FOR BIT DATA	Specifies the input data for UDF.
scid	INTEGER	Specifies the security co-ordinate ID, which is not used. The value should be set to zero.

Returns

This UDF returns the encrypted data as VARCHAR(300) FOR BIT DATA.

> **Note:** If the function call fails, this UDF throws an exception with an appropriate message.

Example

```
CREATE TABLE table-name (CCN VARCHAR(20) FOR BIT DATA)
CCSID EBCDIC IN DATABASE DSNDB04;
INSERT into table-name (CCN)
VALUES (PTY.ins_enc_charfbd(8,'data-element','4234567890123456',0));
```

5.6.2 pty.upd_enc_charfbd

This function is used to encrypt data with a data element.

The external name is PDUVCHRF.

pty.upd_enc_charfbd(communicationid INTEGER, dataelement VARCHAR, input_data VARCHAR FOR BIT DATA, scid INTEGER)

Parameters

Name	Type	Description
communicationid	INTEGER	Specifies the location where policy can be found. Must be the same as configured in `pepserver.cfg`.
dataelement	VARCHAR(64)	Specifies the name of data element.

Name	Type	Description
input_data	VARCHAR(254) FOR BIT DATA	Specifies the input data for UDF.
scid	INTEGER	Specifies the security co-ordinate ID, which is not used. The value should be set to zero.

Returns

This UDF returns the encrypted data as VARCHAR(300) FOR BIT DATA.

> **Note:** If the function call fails, this UDF throws an exception with an appropriate message.

Example

```
UPDATE table-name SET (CCN) =
   PTY.upd_enc_charfbd(8,'data-element','4234567890123456',0);
```

5.6.3 pty.sel_dec_charfbd

This function is used to decrypt data with a data element.

The external name is PDSVCHRF.

pty.sel_dec_charfbd(communicationid INTEGER, dataelement VARCHAR, input_data VARCHAR FOR BIT DATA, scid INTEGER)

Parameters

Name	Type	Description
communicationid	INTEGER	Specifies the location where policy can be found. Must be the same as configured in `pepserver.cfg`.
dataelement	VARCHAR(64)	Specifies the name of data element.
input_data	VARCHAR(300) FOR BIT DATA	Specifies the input data for UDF.
scid	INTEGER	Specifies the security co-ordinate ID, which is not used. The value should be set to zero.

Returns

This UDF returns the decrypted data as VARCHAR(254) FOR BIT DATA.

> **Note:** If the function call fails, this UDF throws an exception with an appropriate message.

Example

```
SELECT pty.sel_dec_charfbd(8,'data-element',CCN,0) as CCN
from table-name;
```

5.7 DATE UDFs

These UDFs can be used to encrypt and decrypt DATE data.

5.7.1 pty.ins_enc_date

This function is used to encrypt data with a data element.

The external name is PDIDATE.

pty.ins_enc_date(communicationid INTEGER, dataelement VARCHAR, input_data DATE, scid INTEGER)

Parameters

Name	Type	Description
communicationid	INTEGER	Specifies the location where policy can be found. Must be the same as configured in `pepserver.cfg`.
dataelement	VARCHAR(64)	Specifies the name of data element.
input_data	DATE	Specifies the input data for UDF.
scid	INTEGER	Specifies the security co-ordinate ID, which is not used. The value should be set to zero.

Returns

This UDF returns the encrypted data as VARCHAR(34) FOR BIT DATA.

> **Note:** If the function call fails, this UDF throws an exception with an appropriate message.

Example

```
CREATE TABLE table-name (DATE1 VARCHAR(34) FOR BIT DATA)
CCSID EBCDIC IN DATABASE DSNDB04;
INSERT into table-name (DATE1)
   VALUES (PTY.ins_enc_date(8,'data-element',DATE('12/31/2014'),0));
```

5.7.2 pty.upd_enc_date

This function can be used to encrypt data with a data element.

The external name is PDUDATE.

pty.upd_enc_date(communicationid INTEGER, dataelement VARCHAR, input_data DATE, scid INTEGER)

Parameters

Name	Type	Description
communicationid	INTEGER	Specifies the location where policy can be found. Must be the same as configured in `pepserver.cfg`.
dataelement	VARCHAR(64)	Specifies the name of data element.

Name	Type	Description
input_data	DATE	Specifies the input data for UDF.
scid	INTEGER	Specifies the security co-ordinate ID, which is not used. The value should be set to zero.

Returns

This UDF returns the encrypted data as VARCHAR(34) FOR BIT DATA.

> **Note:** If the function call fails, this UDF throws an exception with an appropriate message.

Example

```
UPDATE table-name SET (DATE1) =
   PTY.upd_enc_date(8,'data-element',DATE('07/02/1985'),0);
```

5.7.3 pty.sel_dec_date

This function is used to decrypt data with a data element.

The external name is PDSDATE.

pty.sel_dec_date(communicationid INTEGER, dataelement VARCHAR, input_data VARCHAR FOR BIT DATA, scid INTEGER)

Parameters

Name	Type	Description
communicationid	INTEGER	Specifies the location where policy can be found. Must be the same as configured in `pepserver.cfg`.
dataelement	VARCHAR(64)	Specifies the name of data element.
input_data	VARCHAR(34) FOR BIT DATA	Specifies the input data for UDF.
scid	INTEGER	Specifies the security co-ordinate ID, which is not used. The value should be set to zero.

Returns

This UDF returns the decrypted data as DATE.

> **Note:** If the function call fails, this UDF throws an exception with an appropriate message.

Example

```
SELECT pty.sel_dec_date(8,'data-element',DATE1,0) as DATE1
from table-name;
```

5.7.4 pty.ins_date

This function is used for no encryption with a data element as well as for tokenization.

The external name is PDIDATEM.

pty.ins_date(communicationid INTEGER, dataelement VARCHAR, input_data DATE, scid INTEGER)

Parameters

Name	Type	Description
communicationid	INTEGER	Specifies the location where policy can be found. Must be the same as configured in `pepserver.cfg`.
dataelement	VARCHAR(64)	Specifies the name of data element.
input_data	DATE	Specifies the input data for UDF.
scid	INTEGER	Specifies the security co-ordinate ID, which is not used. The value should be set to zero.

Returns

This UDF returns the encrypted data as DATE.

> **Note:** If the function call fails, this UDF throws an exception with an appropriate message.

Example

```
CREATE TABLE table-name (DATE1 DATE)
CCSID EBCDIC IN DATABASE DSNDB04;
INSERT into table-name (DATE1)
   VALUES (PTY.ins_date(8,'date-token',DATE('12/31/2014'),0));
```

5.7.5 pty.upd_date

This function is used for no encryption with a data element as well as for tokenization.

The external name is PDUDATEM.

pty.upd_date(communicationid INTEGER, dataelement VARCHAR, input_data DATE, scid INTEGER)

Parameters

Name	Type	Description
communicationid	INTEGER	Specifies the location where policy can be found. Must be the same as configured in `pepserver.cfg`.
dataelement	VARCHAR(64)	Specifies the name of data element.
input_data	DATE	Specifies the input data for UDF.
scid	INTEGER	Specifies the security co-ordinate ID, which is not used. The value should be set to zero.

Returns

This UDF returns the encrypted data as DATE.

> **Note:** If the function call fails, this UDF throws an exception with an appropriate message.

Example

```
UPDATE table-name SET (DATE1) =
   PTY.upd_date(8,'date-token’,DATE(‘07/02/1985’),0);
```

5.7.6 pty.sel_date

This function is used for no encryption with a data element as well as for de-tokenization.

The external name is PDSDATEM.

pty.sel_date(communicationid INTEGER, dataelement VARCHAR, input_data DATE, scid INTEGER)

Parameters

Name	Type	Description
communicationid	INTEGER	Specifies the location where policy can be found. Must be the same as configured in *pepserver.cfg*.
dataelement	VARCHAR(64)	Specifies the name of data element.
input_data	DATE	Specifies the input data for UDF.
scid	INTEGER	Specifies the security co-ordinate ID, which is not used. The value should be set to zero.

Returns

This UDF returns the decrypted data as DATE.

> **Note:** If the function call fails, this UDF throws an exception with an appropriate message.

Example

```
SELECT pty.sel_date(8,'date-token',DATE1,0) as DATE1
from table-name;
```

5.8 TIMESTAMP UDFs

These UDFs can be used to encrypt and decrypt TIMESTAMP data.

5.8.1 pty.ins_enc_timestamp

This function is used to encrypt data with a data element.

The external name is PDITSTMP.

pty.ins_enc_timestamp(communicationid INTEGER, dataelement VARCHAR, input_data TIMESTAMP, scid INTEGER)

Parameters

Name	Type	Description
communicationid	INTEGER	Specifies the location where policy can be found. Must be the same as configured in `pepserver.cfg`.
dataelement	VARCHAR(64)	Specifies the name of data element.
input_data	TIMESTAMP	Specifies the input data for UDF.
scid	INTEGER	Specifies the security co-ordinate ID, which is not used. The value should be set to zero.

Returns

This UDF returns the encrypted data as VARCHAR(50) FOR BIT DATA.

> **Note:** If the function call fails, this UDF throws an exception with an appropriate message.

Example

```
CREATE TABLE table-name (DATETIME VARCHAR(50) FOR BIT DATA)
CCSID EBCDIC IN DATABASE DSNDB04;
INSERT into table-name (DATETIME)
    VALUES (PTY.ins_enc_timestamp(8,'data-element',TIMESTAMP('2014-12-31 11:59:59'),0));
```

5.8.2 pty.upd_enc_timestamp

This function can be used to encrypt data with a data element.

The external name is PDUTSTMP.

pty.upd_enc_timestamp(communicationid INTEGER, dataelement VARCHAR, input_data TIMESTAMP, scid INTEGER)

Parameters

Name	Type	Description
communicationid	INTEGER	Specifies the location where policy can be found. Must be the same as configured in `pepserver.cfg`.
dataelement	VARCHAR(64)	Specifies the name of data element.
input_data	TIMESTAMP	Specifies the input data for UDF.
scid	INTEGER	Specifies the security co-ordinate ID, which is not used. The value should be set to zero.

Returns

This UDF returns the encrypted data as VARCHAR (50) FOR BIT DATA.

> **Note:** If the function call fails, this UDF throws an exception with an appropriate message.

Example

```
UPDATE table-name SET (DATETIME) =
   PTY.upd_enc_timestamp(8,'data-element', TIMESTAMP('2014-12-31 11:59:59'),0);
```

5.8.3 pty.sel_dec_timestamp

This function is used to decrypt data with a data element.

The external name is PDSTSTMP.

pty.sel_dec_timestamp(communicationid INTEGER, dataelement VARCHAR, input_data VARCHAR FOR BIT DATA, scid INTEGER)

Parameters

Name	Type	Description
communicationid	INTEGER	Specifies the location where policy can be found. Must be the same as configured in `pepserver.cfg`.
dataelement	VARCHAR(64)	Specifies the name of data element.
input_data	VARCHAR(50) FOR BIT DATA	Specifies the input data for UDF.
scid	INTEGER	Specifies the security co-ordinate ID, which is not used. The value should be set to zero.

Returns

This UDF returns the decrypted data as TIMESTAMP.

> **Note:** If the function call fails, this UDF throws an exception with an appropriate message.

Example

```
SELECT pty.sel_dec_timestamp(8,'data-element',DATETIME,0) as DATETIME
```

5.9 TIME UDFs

These UDFs can be used to encrypt and decrypt TIME data.

5.9.1 pty.ins_enc_time

This function is used to encrypt data with a data element.

The external name is PDITIME.

pty.ins_enc_time(communicationid INTEGER, dataelement VARCHAR, input_data TIME, scid INTEGER)

Parameters

Name	Type	Description
communicationid	INTEGER	Specifies the location where policy can be found. Must be the same as configured in `pepserver.cfg`.
dataelement	VARCHAR(64)	Specifies the name of data element.
input_data	TIME	Specifies the input data for UDF.
scid	INTEGER	Specifies the security co-ordinate ID, which is not used. The value should be set to zero.

Returns

This UDF returns the encrypted data as VARCHAR(34) FOR BIT DATA.

> **Note:** If the function call fails, this UDF throws an exception with an appropriate message.

Example

```
CREATE TABLE table-name (TIME1 VARCHAR(34) FOR BIT DATA)
CCSID EBCDIC IN DATABASE DSNDB04;
INSERT into table-name (TIME1)
   VALUES (PTY.ins_enc_time(8,'data-element',TIME('11:59:59'),0));
```

5.9.2 pty.upd_enc_time

This function can be used to encrypt data with a data element.

The external name is PDUTIME.

pty.upd_enc_time(communicationid INTEGER, dataelement VARCHAR, input_data TIME, scid INTEGER)

Parameters

Name	Type	Description
communicationid	INTEGER	Specifies the location where policy can be found. Must be the same as configured in `pepserver.cfg`.
dataelement	VARCHAR(64)	Specifies the name of data element.
input_data	TIME	Specifies the input data for UDF.
scid	INTEGER	Specifies the security co-ordinate ID, which is not used. The value should be set to zero.

Returns

This UDF returns the encrypted data as VARCHAR(34) FOR BIT DATA.

> **Note:** If the function call fails, this UDF throws an exception with an appropriate message.

Example

```
UPDATE table-name SET (TIME1) =
   PTY.upd_enc_time(8,'data-element',TIME('02:59:59'),0);
```

5.9.3 pty.sel_dec_time

This function is used to decrypt data with a data element.

The external name is PDSTIME.

pty.sel_dec_time(communicationid INTEGER, dataelement VARCHAR, input_data VARCHAR FOR BIT DATA, scid INTEGER)

Parameters

Name	Type	Description
communicationid	INTEGER	Specifies the location where policy can be found. Must be the same as configured in `pepserver.cfg`.
dataelement	VARCHAR(64)	Specifies the name of data element.
input_data	VARCHAR(34) FOR BIT DATA	Specifies the input data for UDF.
scid	INTEGER	Specifies the security co-ordinate ID, which is not used. The value should be set to zero.

Returns

This UDF returns the decrypted data as TIME.

> **Note:** If the function call fails, this UDF throws an exception with an appropriate message.

Example

```
SELECT pty.sel_dec_time(8,'data-element',TIME1,0) as TIME1
from table-name;
```

5.9.4 pty.ins_time

This function is used for no encryption with a data element as well as for tokenization.

The external name is PDITIME2.

pty.ins_time(communicationid INTEGER, dataelement VARCHAR, input_data TIME, scid INTEGER)

Parameters

Name	Type	Description
communicationid	INTEGER	Specifies the location where policy can be found. Must be the same as configured in `pepserver.cfg`.
dataelement	VARCHAR(64)	Specifies the name of data element.

Name	Type	Description
input_data	TIME	Specifies the input data for UDF.
scid	INTEGER	Specifies the security co-ordinate ID, which is not used. The value should be set to zero.

Returns

This UDF returns the encrypted data as TIME.

> **Note:** If the function call fails, this UDF throws an exception with an appropriate message.

Example

```
CREATE TABLE table-name (TIME1 TIME)
CCSID EBCDIC IN DATABASE DSNDB04;
INSERT into table-name (TIME1)
   VALUES (PTY.ins_time(8,'NOENC',TIME('11:59:59'),0));
```

5.9.5 pty.upd_time

This function is used for no encryption with a data element as well as for tokenization.

The external name is PDUTIME2.

pty.upd_time(communicationid INTEGER, dataelement VARCHAR, input_data TIME, scid INTEGER)

Parameters

Name	Type	Description
communicationid	INTEGER	Specifies the location where policy can be found. Must be the same as configured in `pepserver.cfg`.
dataelement	VARCHAR(64)	Specifies the name of data element.
input_data	TIME	Specifies the input data for UDF.
scid	INTEGER	Specifies the security co-ordinate ID, which is not used. The value should be set to zero.

Returns

This UDF returns the encrypted data as TIME.

> **Note:** If the function call fails, this UDF throws an exception with an appropriate message.

Example

```
UPDATE table-name SET (TIME1) =
   PTY.upd_time(8,'token-element',TIME('11:59:59'),0);
```

5.9.6 pty.sel_time

This function is used for no encryption with a data element as well as for de-tokenization.

The external name is PDSTIME2.

pty.sel_time(communicationid INTEGER, dataelement VARCHAR, input_data TIME, scid INTEGER)

Parameters

Name	Type	Description
communicationid	INTEGER	Specifies the location where policy can be found. Must be the same as configured in `pepserver.cfg`.
dataelement	VARCHAR(64)	Specifies the name of data element.
input_data	TIME	Specifies the input data for UDF.
scid	INTEGER	Specifies the security co-ordinate ID, which is not used. The value should be set to zero.

Returns

This UDF returns the decrypted data as TIME.

> **Note:** If the function call fails, this UDF throws an exception with an appropriate message.

Example

```
SELECT pty.sel_time(8,'token-element',TIME1,0) as TIME1
    from table-name;
```

5.10 INTEGER UDFs

These UDFs can be used to encrypt and decrypt INTEGER data.

5.10.1 pty.ins_enc_integer

This function is used to encrypt data with a data element.

The external name is PDIINT.

pty.ins_enc_integer(communicationid INTEGER, dataelement VARCHAR, input_data INTEGER, scid INTEGER)

Parameters

Name	Type	Description
communicationid	INTEGER	Specifies the location where policy can be found. Must be the same as configured in `pepserver.cfg`.
dataelement	VARCHAR(64)	Specifies the name of data element.
input_data	INTEGER	Specifies the input data for UDF.
scid	INTEGER	Specifies the security co-ordinate ID, which is not used. The value should be set to zero.

Returns

This UDF returns the encrypted data as VARCHAR(34) FOR BIT DATA.

> **Note:** If the function call fails, this UDF throws an exception with an appropriate message.

Example

```
CREATE TABLE table-name (INT4 VARCHAR(34) FOR BIT DATA)
CCSID EBCDIC IN DATABASE DSNDB04;
INSERT into table-name (INT4)
   VALUES (PTY.ins_enc_integer(8,'data-element',1234567890,0));
```

5.10.2 pty.upd_enc_integer

This function can be used to encrypt data with a data element.

The external name is PDUINT.

pty.upd_enc_integer(communicationid INTEGER, dataelement VARCHAR, input_data INTEGER, scid INTEGER)

Parameters

Name	Type	Description
communicationid	INTEGER	Specifies the location where policy can be found. Must be the same as configured in `pepserver.cfg`.
dataelement	VARCHAR(64)	Specifies the name of data element.
input_data	INTEGER	Specifies the input data for UDF.
scid	INTEGER	Specifies the security co-ordinate ID, which is not used. The value should be set to zero.

Returns

This UDF returns the encrypted data as VARCHAR (34) FOR BIT DATA.

> **Note:** If the function call fails, this UDF throws an exception with an appropriate message.

Example

```
UPDATE table-name SET (INT4) =
   PTY.upd_enc_integer(8,'data-element',9876543210,0);
```

5.10.3 pty.sel_dec_integer

This function is used to decrypt data with a data element.

The external name is PDSINT.

pty.sel_dec_integer(communicationid INTEGER, dataelement VARCHAR, input_data VARCHAR FOR BIT DATA, scid INTEGER)

Parameters

Name	Type	Description
communicationid	INTEGER	Specifies the location where policy can be found. Must be the same as configured in `pepserver.cfg`.
dataelement	VARCHAR(64)	Specifies the name of data element.
input_data	VARCHAR(34) FOR BIT DATA	Specifies the input data for UDF.
scid	INTEGER	Specifies the security co-ordinate ID, which is not used. The value should be set to zero.

Returns

This UDF returns the decrypted data as INTEGER.

> **Note:** If the function call fails, this UDF throws an exception with an appropriate message.

Example

```
SELECT pty.sel_dec_integer(8,'data-element',INT4,0) as INT4
from table-name;
```

5.10.4 pty.ins_integer

This function is used for no encryption with a data element as well as for tokenization.

The external name is PDIINT2.

pty.ins_integer(communicationid INTEGER, dataelement VARCHAR, input_data INTEGER, scid INTEGER)

Parameters

Name	Type	Description
communicationid	INTEGER	Specifies the location where policy can be found. Must be the same as configured in `pepserver.cfg`.
dataelement	VARCHAR(64)	Specifies the name of data element.
input_data	INTEGER	Specifies the input data for UDF.
scid	INTEGER	Specifies the security co-ordinate ID, which is not used. The value should be set to zero.

Returns

This UDF returns the encrypted data as INTEGER.

> **Note:** If the function call fails, this UDF throws an exception with an appropriate message.

Example

```
CREATE TABLE table-name (INT4 INTEGER)
CCSID EBCDIC IN DATABASE DSNDB04;
```

```
INSERT into table-name (INT4)
   VALUES (PTY.ins_integer(8,’integer-token(4)',1234567890,0));
```

5.10.5 pty.upd_integer

This function is used for no encryption with a data element as well as for tokenization.

The external name is PDUINT2.

pty.upd_integer(communicationid INTEGER, dataelement VARCHAR, input_data INTEGER, scid INTEGER)

Parameters

Name	Type	Description
communicationid	INTEGER	Specifies the location where policy can be found. Must be the same as configured in `pepserver.cfg`.
dataelement	VARCHAR(64)	Specifies the name of data element.
input_data	INTEGER	Specifies the input data for UDF.
scid	INTEGER	Specifies the security co-ordinate ID, which is not used. The value should be set to zero.

Returns

This UDF returns the encrypted data as INTEGER.

> **Note:** If the function call fails, this UDF throws an exception with an appropriate message.

Example

```
UPDATE table-name SET (INT4) =
   PTY.upd_integer(8,'integer-token(4)',9876543210,0);
```

5.10.6 pty.sel_integer

This function is used for no encryption with a data element as well as for de-tokenization.

The external name is PDSINT2.

pty.sel_time(communicationid INTEGER, dataelement VARCHAR, input_data INTEGER, scid INTEGER)

Parameters

Name	Type	Description
communicationid	INTEGER	Specifies the location where policy can be found. Must be the same as configured in `pepserver.cfg`.
dataelement	VARCHAR(64)	Specifies the name of data element.
input_data	INTEGER	Specifies the input data for UDF.

Name	Type	Description
scid	INTEGER	Specifies the security co-ordinate ID, which is not used. The value should be set to zero.

Returns

This UDF returns the decrypted data as INTEGER.

> **Note:** If the function call fails, this UDF throws an exception with an appropriate message.

Example

```
SELECT pty.sel_integer(8,'integer-token(4)',INT4,0) as INT4
from table-name;
```

5.11 SMALLINT UDFs

These UDFs can be used to encrypt and decrypt SMALLINT data.

5.11.1 pty.ins_enc_smallint

This function is used to encrypt data with a data element.

The external name is PDISINT.

pty.ins_enc_smallint(communicationid INTEGER, dataelement VARCHAR, input_data SMALLINT, scid INTEGER)

Parameters

Name	Type	Description
comunicationid	INTEGER	Specifies the location where policy can be found. Must be the same as configured in `pepserver.cfg`.
dataelement	VARCHAR(64)	Specifies the name of data element.
input_data	SMALLINT	Specifies the input data for UDF.
scid	INTEGER	Specifies the security co-ordinate ID, which is not used. The value should be set to zero.

Returns

This UDF returns the encrypted data as VARCHAR(34) FOR BIT DATA.

> **Note:** If the function call fails, this UDF throws an exception with an appropriate message.

Example

```
CREATE TABLE table-name (INT2 VARCHAR(34) FOR BIT DATA)
CCSID EBCDIC IN DATABASE DSNDB04;
INSERT into table-name (INT2)
   VALUES (PTY.ins_enc_smallint(8,'data-element',12345,0));
```

5.11.2 pty.upd_enc_smallint

This function can be used to encrypt data with a data element.

The external name is PDUSINT.

pty.upd_enc_smallint(communicationid INTEGER, dataelement VARCHAR, input_data SMALLINT, scid INTEGER)

Parameters

Name	Type	Description
comunicationid	INTEGER	Specifies the location where policy can be found. Must be the same as configured in `pepserver.cfg`.
dataelement	VARCHAR(64)	Specifies the name of data element.
input_data	SMALLINT	Specifies the input data for UDF.
scid	INTEGER	Specifies the security co-ordinate ID, which is not used. The value should be set to zero.

Returns

This UDF returns the encrypted data as VARCHAR(34) FOR BIT DATA.

> **Note:** If the function call fails, this UDF throws an exception with an appropriate message.

Example

```
UPDATE table-name SET (INT2) =
   PTY.upd_enc_smallint(8,'data-element',9876,0);
```

5.11.3 pty.sel_dec_smallint

This function is used to decrypt data with a data element.

The external name is PDSSINT.

pty.sel_dec_smallint(communicationid INTEGER, dataelement VARCHAR, input_data VARCHAR FOR BIT DATA, scid INTEGER)

Parameters

Name	Type	Description
communicationid	INTEGER	Specifies the location where policy can be found. Must be the same as configured in `pepserver.cfg`.
dataelement	VARCHAR(64)	Specifies the name of data element.
input_data	VARCHAR(34) FOR BIT DATA	Specifies the input data for UDF.

Name	Type	Description
scid	INTEGER	Specifies the security co-ordinate ID, which is not used. The value should be set to zero.

Returns

This UDF returns the decrypted data as SMALLINT.

> **Note:** If the function call fails, this UDF throws an exception with an appropriate message.

Example

```
SELECT pty.sel_dec_smallint(8,'data-element',INT2,0) as INT2
from table-name;
```

5.11.4 pty.ins_smallint

This function is used for no encryption with a data element as well as for tokenization.

The external name is PDISINT2.

pty.ins_smallint(communicationid INTEGER, dataelement VARCHAR, input_data SMALLINT, scid INTEGER)

Parameters

Name	Type	Description
communicationid	INTEGER	Specifies the location where policy can be found. Must be the same as configured in *`pepserver.cfg`*.
dataelement	VARCHAR(64)	Specifies the name of data element.
input_data	SMALLINT	Specifies the input data for UDF.
scid	INTEGER	Specifies the security co-ordinate ID, which is not used. The value should be set to zero.

Returns

This UDF returns the encrypted data as SMALLINT.

> **Note:** If the function call fails, this UDF throws an exception with an appropriate message.

Example

```
CREATE TABLE table-name (INT2 SMALLINT)
CCSID EBCDIC IN DATABASE DSNDB04;
INSERT into table-name (INT2)
   VALUES (PTY.ins_smallint(8,'smallint-token(2)',12345,0));
```

5.11.5 pty.upd_smallint

This function is used for no encryption with a data element as well as for tokenization.

The external name is PDUSINT2.

pty.upd_smallint(communicationid INTEGER, dataelement VARCHAR, input_data SMALLINT, scid INTEGER)

Parameters

Name	Type	Description
comunicationid	INTEGER	Specifies the location where policy can be found. Must be the same as configured in `pepserver.cfg`.
dataelement	VARCHAR(64)	Specifies the name of data element.
input_data	SMALLINT	Specifies the input data for UDF.
scid	INTEGER	Specifies the security co-ordinate ID, which is not used. The value should be set to zero.

Returns

This UDF returns the encrypted data as SMALLINT.

> **Note:** If the function call fails, this UDF throws an exception with an appropriate message.

Example

```
UPDATE table-name SET (INT2) =
   PTY.upd_smallint(8,'smallint-token(2)',987,0);
```

5.11.6 pty.sel_smallint

This function is used for no encryption with a data element as well as for de-tokenization.

The external name is PDSSINT2.

pty.sel_smallint(communicationid INTEGER, dataelement VARCHAR, input_data SMALLINT, scid INTEGER)

Parameters

Name	Type	Description
comunicationid	INTEGER	Specifies the location where policy can be found. Must be the same as configured in `pepserver.cfg`.
dataelement	VARCHAR(64)	Specifies the name of data element.
input_data	SMALLINT	Specifies the input data for UDF.
scid	INTEGER	Specifies the security co-ordinate ID, which is not used. The value should be set to zero.

Returns

This UDF returns the decrypted data as SMALLINT.

> **Note:** If the function call fails, this UDF throws an exception with an appropriate message.

Example

```
SELECT pty.sel_smallint(8,'smallint-token(2)',INT2,0) as INT2
from table-name;
```

5.12 REAL UDFs

These UDFs can be used to encrypt and decrypt REAL data.

5.12.1 pty.ins_enc_real

This function is used to encrypt data with a data element.

The external name is PDIREAL.

pty.ins_enc_real(communicationid INTEGER, dataelement VARCHAR, input_data REAL, scid INTEGER)

Parameters

Name	Type	Description
communicationid	INTEGER	Specifies the location where policy can be found. Must be the same as configured in `pepserver.cfg`.
dataelement	VARCHAR(64)	Specifies the name of data element.
input_data	REAL	Specifies the input data for UDF.
scid	INTEGER	Specifies the security co-ordinate ID, which is not used. The value should be set to zero.

Returns

This UDF returns the encrypted data as VARCHAR(34) FOR BIT DATA.

> **Note:** If the function call fails, this UDF throws an exception with an appropriate message.

Example

```
CREATE TABLE table-name (REAL1 VARCHAR(34) FOR BIT DATA)
CCSID EBCDIC IN DATABASE DSNDB04;
INSERT into table-name (REAL1)
   VALUES (PTY.ins_enc_real(8,'data-element', REAL(100.001),0));
```

5.12.2 pty.upd_enc_real

This function can be used to encrypt data with a data element.

The external name is PDUREAL.

pty.upd_enc_real(communicationid INTEGER, dataelement VARCHAR, input_data REAL, scid INTEGER)

Parameters

Name	Type	Description
communicationid	INTEGER	Specifies the location where policy can be found. Must be the same as configured in `pepserver.cfg`.
dataelement	VARCHAR(64)	Specifies the name of data element.
input_data	REAL	Specifies the input data for UDF.
scid	INTEGER	Specifies the security co-ordinate ID, which is not used. The value should be set to zero.

Returns

This UDF returns the encrypted data as VARCHAR(34) FOR BIT DATA.

> **Note:** If the function call fails, this UDF throws an exception with an appropriate message.

Example

```
UPDATE table-name SET (REAL1) =
   PTY.upd_enc_real(8,'data-element', REAL(298.981),0);
```

5.12.3 pty.sel_dec_real

This function is used to decrypt data with a data element.

The external name is PDSREAL.

pty.sel_dec_real(communicationid INTEGER, dataelement VARCHAR, input_data VARCHAR FOR BIT DATA, scid INTEGER)

Parameters

Name	Type	Description
communicationid	INTEGER	Specifies the location where policy can be found. Must be the same as configured in `pepserver.cfg`.
dataelement	VARCHAR(64)	Specifies the name of data element.
input_data	VARCHAR(34) FOR BIT DATA	Specifies the input data for UDF.
scid	INTEGER	Specifies the security co-ordinate ID, which is not used. The value should be set to zero.

Returns

This UDF returns the decrypted data as REAL.

> **Note:** If the function call fails, this UDF throws an exception with an appropriate message.

Example

```
SELECT pty.sel_dec_real(8,'data-element',REAL1,0) as REAL1
from table-name;
```

5.12.4 pty.ins_real

This function is used for no encryption with a data element as well as for tokenization.

The external name is PDIREAL2.

pty.ins_real(communicationid INTEGER, dataelement VARCHAR, input_data REAL, scid INTEGER)

Parameters

Name	Type	Description
communicationid	INTEGER	Specifies the location where policy can be found. Must be the same as configured in `pepserver.cfg`.
dataelement	VARCHAR(64)	Specifies the name of data element.
input_data	REAL	Specifies the input data for UDF.
scid	INTEGER	Specifies the security co-ordinate ID, which is not used. The value should be set to zero.

Returns

This UDF returns the encrypted data as REAL.

> **Note:** If the function call fails, this UDF throws an exception with an appropriate message.

Example

```
CREATE TABLE table-name (REAL1 REAL)
CCSID EBCDIC IN DATABASE DSNDB04;
INSERT into table-name (REAL1)
   VALUES (PTY.ins_real(8,'NOENC',REAL(100.001),0));
```

5.12.5 pty.upd_real

This function is used for no encryption with a data element as well as for tokenization.

The external name is PDUREAL2.

pty.upd_real(communicationid INTEGER, dataelement VARCHAR, input_data REAL, scid INTEGER)

Parameters

Name	Type	Description
communicationid	INTEGER	Specifies the location where policy can be found. Must be the same as configured in `pepserver.cfg`.
dataelement	VARCHAR(64)	Specifies the name of data element.

Name	Type	Description
input_data	REAL	Specifies the input data for UDF.
scid	INTEGER	Specifies the security co-ordinate ID, which is not used. The value should be set to zero.

Returns

This UDF returns the encrypted data as REAL.

> **Note:** If the function call fails, this UDF throws an exception with an appropriate message.

Example

```
UPDATE table-name SET (REAL1) =
   PTY.upd_real(8,'NOENC',REAL(298.981),0);
```

5.12.6 pty.sel_real

This function is used for no encryption with a data element as well as for de-tokenization.

The external name is PDSREAL2.

pty.sel_real(communicationid INTEGER, dataelement VARCHAR, input_data REAL, scid INTEGER)

Parameters

Name	Type	Description
communicationid	INTEGER	Specifies the location where policy can be found. Must be the same as configured in `pepserver.cfg`.
dataelement	VARCHAR(64)	Specifies the name of data element.
input_data	REAL	Specifies the input data for UDF.
scid	INTEGER	Specifies the security co-ordinate ID, which is not used. The value should be set to zero.

Returns

This UDF returns the decrypted data as REAL.

> **Note:** If the function call fails, this UDF throws an exception with an appropriate message.

Example

```
SELECT pty.sel_real(8,'NOENC',REAL1,0) as REAL1
from table-name;
```

5.13 DOUBLE UDFs

These UDFs can be used to encrypt and decrypt DOUBLE data.

5.13.1 pty.ins_enc_double

This function is used to encrypt data with a data element.

The external name is PDIDBLE.

pty.ins_enc_double(communicationid INTEGER, dataelement VARCHAR, input_data DOUBLE, scid INTEGER)

Parameters

Name	Type	Description
communicationid	INTEGER	Specifies the location where policy can be found. Must be the same as configured in `pepserver.cfg`.
dataelement	VARCHAR(64)	Specifies the name of data element.
input_data	DOUBLE	Specifies the input data for UDF.
scid	INTEGER	Specifies the security co-ordinate ID, which is not used. The value should be set to zero.

Returns

This UDF returns the encrypted data as VARCHAR(34) FOR BIT DATA.

> **Note:** If the function call fails, this UDF throws an exception with an appropriate message.

Example

```
CREATE TABLE table-name (DOUBLE1 VARCHAR(34) FOR BIT DATA)
CCSID EBCDIC IN DATABASE DSNDB04;
INSERT into table-name (DOUBLE1)
   VALUES (PTY.ins_enc_double(8,'data-element', DOUBLE(100.001),0));
```

5.13.2 pty.upd_enc_double

This function can be used to encrypt data with a data element.

The external name is PDUDBLE.

pty.upd_enc_double(communicationid INTEGER, dataelement VARCHAR, input_data DOUBLE, scid INTEGER)

Parameters

Name	Type	Description
communicationid	INTEGER	Specifies the location where policy can be found. Must be the same as configured in `pepserver.cfg`.
dataelement	VARCHAR(64)	Specifies the name of data element.
input_data	DOUBLE	Specifies the input data for UDF.

Name	Type	Description
scid	INTEGER	Specifies the security co-ordinate ID, which is not used. The value should be set to zero.

Returns

This UDF returns the encrypted data as VARCHAR(34) FOR BIT DATA.

> **Note:** If the function call fails, this UDF throws an exception with an appropriate message.

Example

```
UPDATE table-name SET (DOUBLE1) =
   PTY.upd_enc_double(8,'data-element', DOUBLE(298.981),0);
```

5.13.3 pty.sel_dec_double

This function is used to decrypt data with a data element.

The external name is PDSDBLE.

pty.sel_dec_double(communicationid INTEGER, dataelement VARCHAR, input_data VARCHAR FOR BIT DATA, scid INTEGER)

Parameters

Name	Type	Description
communicationid	INTEGER	Specifies the location where policy can be found. Must be the same as configured in `pepserver.cfg`.
dataelement	VARCHAR(64)	Specifies the name of data element.
input_data	VARCHAR(34) FOR BIT DATA	Specifies the input data for UDF.
scid	INTEGER	Specifies the security co-ordinate ID, which is not used. The value should be set to zero.

Returns

This UDF returns the decrypted data as DOUBLE.

> **Note:** If the function call fails, this UDF throws an exception with an appropriate message.

Example

```
SELECT pty.sel_dec_double(8,'data-element',DOUBLE1,0) as DOUBLE1
from table-name;
```

5.13.4 pty.ins_double

This function is used for no encryption with a data element as well as for tokenization.

The external name is PDIDBLE2.

pty.ins_double(communicationid INTEGER, dataelement VARCHAR, input_data DOUBLE, scid INTEGER)

Parameters

Name	Type	Description
communicationid	INTEGER	Specifies the location where policy can be found. Must be the same as configured in `pepserver.cfg`.
dataelement	VARCHAR(64)	Specifies the name of data element.
input_data	DOUBLE	Specifies the input data for UDF.
scid	INTEGER	Specifies the security co-ordinate ID, which is not used. The value should be set to zero.

Returns

This UDF returns the encrypted data as DOUBLE.

> **Note:** If the function call fails, this UDF throws an exception with an appropriate message.

Example

```
CREATE TABLE table-name (DOUBLE1 DOUBLE)
CCSID EBCDIC IN DATABASE DSNDB04;
INSERT into table-name (DOUBLE1)
   VALUES (PTY.ins_double(8,'NOENC',DOUBLE(100.001),0));
```

5.13.5 pty.upd_double

This function is used for no encryption with a data element as well as for tokenization.

The external name is PDUDBLE2.

pty.upd_double(communicationid INTEGER, dataelement VARCHAR, input_data DOUBLE, scid INTEGER)

Parameters

Name	Type	Description
communicationid	INTEGER	Specifies location where policy can be found. Must be the same as configured in `pepserver.cfg`
dataelement	VARCHAR(64)	Specifies the name of data element.
input_data	DOUBLE	Specifies the input data for UDF.
scid	INTEGER	Specifies the security co-ordinate ID, which is not used. The value should be set to zero.

Returns

This UDF returns the encrypted data as DOUBLE.

> **Note:** If the function call fails, this UDF throws an exception with an appropriate message.

Example

```
UPDATE table-name SET (DOUBLE1) =
   PTY.upd_double(8,'NOENC',DOUBLE(298.981),0);
```

5.13.6 pty.sel_double

This function is used for no encryption with a data element as well as for de-tokenization.

The external name is PDSDBLE2.

pty.sel_double(communicationid INTEGER, dataelement VARCHAR, input_data DOUBLE, scid INTEGER)

Parameters

Name	Type	Description
communicationid	INTEGER	Specifies the location where policy can be found. Must be the same as configured in `pepserver.cfg`.
dataelement	VARCHAR(64)	Specifies the name of data element.
input_data	DOUBLE	Specifies the input data for UDF.
scid	INTEGER	Specifies the security co-ordinate ID, which is not used. The value should be set to zero.

Returns

This UDF returns the decrypted data as DOUBLE.

> **Note:** If the function call fails, this UDF throws an exception with an appropriate message.

Example

```
SELECT pty.sel_double(8,'NOENC',DOUBLE1,0)
    as DOUBLE1 from table-name;
```

Chapter 6

Appendix A: DevOps REST APIs

The DevOps REST APIs include the Policy Management REST APIs and appliance-specific information APIs.

The Policy Management REST APIs are used to create or manage the policies. The policy management functions performed from the ESA Web UI can also be performed using the REST APIs. In addition, the read-only information about the appliance is also available using the REST API.

6.1 Getting Started

This section describes the steps that you can perform to access and view the DevOps API specification document.

Before you begin

The steps mentioned in this section contains the usage of Docker containers and services to download and launch the images for Swagger Editor within a Docker container.

For more information about Docker, refer to the Docker documentation.

1. Install and start the Swagger Editor.
2. Download the Swagger Editor image within a Docker container using the following command.

   ```
   docker pull swaggerapi/swagger-editor
   ```

3. Launch the Docker container using the following command.

   ```
   docker run -d -p 8888:8080 swaggerapi/swagger-editor
   ```

4. Paste the following address on a browser window to access the Swagger Editor using the specified host port.

   ```
   http://localhost:8888/
   ```

5. Download the DevOps API specification document using the following command.

   The DevOps API specification document, version v1 can be downloaded using the following command:

   ```
   curl --insecure --user <username>:<password> "https://ip-address/pty/v1/pim/doc" -H
   "accept: application/x-yaml" --output pim-devops.yaml
   ```

The DevOps API specification document, version v2 can be downloaded using the following command.

```
curl --insecure --user <username>:<password> "https://ip-address/pty/v2/pim/doc" -H
"accept: application/x-yaml" --output pim-devops.yaml
```

> **Note:**
> As the DevOps API specification document, version v1 is deprecated, download and refer to the latest version of the DevOps API specification document, version v2.

6. Drag and drop the downloaded *pim-devops.yaml* file into a browser window.

6.2 Accessing the ESA using the DevOps REST APIs

The following authentication mechanisms can be used to access the ESA.

- Basic authentication with user name and password
- Client Certificate-based authentication
- Token-based authentication

For more information about accessing the ESA using these authentication mechanisms, refer to the *Appliance Overview Guide 8.0.0*.

6.3 Using the DevOps REST APIs

This section explains the usage of the DevOps APIs with some generic samples.

The following table provides section references that explain usage of some of the Policy Management REST APIs. It includes an example workflow to work with the Policy Management functions.

Table 6-1: REST API Section Reference

REST API	Section Reference
Get service version information	*Getting the Service Version Information*
Policy Management initialization	*Initializing the Policy Management*
Creating an empty manual role that will accept all users	*Creating a Manual Role*
Create data elements	*Create Data Elements*
Create policy	*Create Policy*
Add roles and data elements to the policy	*Adding roles and data elements to the policy*
Create a default data store	*Creating a default datastore*
Deploy the data store	*Deploying the Data Store*
Get the deployment information	*Getting the Deployment Information*

6.3.1 Getting the Service Version Information

This section explains how you can get the service version information.

Base URL

https://{Appliance IP address}/pty/v2

In the base URL, the Appliance IP address specifies the IP address of the required appliance.

Path

/pim/version

Method

GET

Sample Request

```
curl --insecure --user <username>:<password> -X GET "https://{Appliance IP address}/pty/v2/pim/version" -H "accept: application/json"
```

6.3.2 Initializing the Policy Management

This section explains how you can initialize the Policy Management to create the keys-related data and the policy repository. If you are initializing the Policy Management from the ESA Web UI, then the execution of this service is not required.

For more information about initializing the Policy Management from the ESA Web UI, refer to the *Policy Management Guide 8.0.0*.

Base URL

```
https://{Appliance IP address}/pty/v2
```

In the base URL, the Appliance IP address specifies the IP address of the required appliance.

Path

/pim/init

Method

POST

Sample Request

```
curl -k --user <username>:<password> -X POST "https://{Appliance IP address}:443/pty/v2/pim/init" -H "accept: application/json"
```

6.3.3 Creating a Manual Role

This section explains how you can create a manual role that accepts all the users.

For more information about working with roles, refer to the *Policy Management Guide 8.0.0*.

Base URL

```
https://{Appliance IP address}/pty/v2
```

In the base URL, the Appliance IP address specifies the IP address of the required appliance.

Path

/pim/roles

Method

POST

Sample Request

```
curl --insecure --user <username>:<password> -X POST "https://{Appliance IP address}:443/pty/v2/pim/roles" -H "accept: application/json" -H "Content-Type: application/json" -d "{\"name\":\"ROLE\",\"mode\":\"MANUAL\",\"allowAll\": true}"
```

6.3.4 Creating Data Elements

This section explains how you can create the data elements.

For more information about working with data elements, refer to the *Policy Management Guide 8.0.0*.

Base URL

https://{Appliance IP address}/pty/v2

In the base URL, the Appliance IP address specifies the IP address of the required appliance.

Path

/pim/roles

Method

POST

Sample Request

```
curl --insecure --user <username>:<password> -X POST "https://{Appliance IP
address}:443/pty/v2/pim/dataelements" -H "accept: application/json" -H "Content-Type: application/
json" -d "{\"name\": \"DE_ALPHANUM\",\"description\": \"DE_ALPHANUM\",\"alphaNumericToken\":
{\"tokenizer\":\"SLT_1_3\",\"fromLeft\": 0,\"fromRight\": 0,\"lengthPreserving\": true, \"allowShort\": \"YES\"}}"
```

6.3.5 Creating Policy

This section explains how you can create a policy.

For more information about working with policies, refer to the *Policy Management Guide 8.0.0*.

Base URL

https://{Appliance IP address}/pty/v2

In the base URL, the Appliance IP address specifies the IP address of the required appliance.

Path

/pim/policies

Method

POST

Sample Request

```
curl --insecure --user <username>:<password> -X POST "https://{Appliance IP address}:443/pty/v2/pim/policies"
-H "accept: application/json" -H "Content-Type: application/json" -d "{\"name\":\"POLICY\",\"description\":
\"POLICY\", \"template\":{\"access\":{\"protect\":true,\"reProtect\":true,\"unProtect\":true},\"audit\":
{\"success\":{\"protect\":false,\"reProtect\":false,\"unProtect\":false},\"failed\":
{\"protect\":false,\"reProtect\":false,\"unProtect\":false}}}}"
```

6.3.6 Adding Roles and Data Elements to a Policy

This section explains how you can add roles and data elements to a policy.

For more information about adding roles and data elements to a policy, refer to the *Policy Management Guide 8.0.0*.

Base URL

https://{Appliance IP address}/pty/v2

In the base URL, the Appliance IP address specifies the IP address of the required appliance.

Path

/pim/policies/1/rules

Method

POST

Sample Request

```
curl --insecure --user <username>:<password> -X POST "https://{Appliance IP
address}:443/pty/v2/pim/policies/1/rules" -H "accept: application/json" -H "Content-Type:
application/json" -d "{\"role\":\"1\",\"dataElement\":\"1\",\"noAccessOperation\":\"EXCEPTION\",\"permission\":
{\"access\":{\"protect\":true,\"reProtect\":true,\"unProtect\":true},\"audit\":
{\"success\":{\"protect\":false,\"reProtect\":false,\"unProtect\":false},\"failed\":
{\"protect\":false,\"reProtect\":false,\"unProtect\":false}}}}"
```

6.3.7 Creating a Default Data Store

This section explains how you can create a default data store.

For more information about working with data stores, refer to the *Policy Management Guide 8.0.0*.

Base URL

```
https://{Appliance IP address}/pty/v2
```

In the base URL, the Appliance IP address specifies the IP address of the required appliance.

Path

/pim/datastores

Method

POST

Sample Request

```
curl --insecure --user <username>:<password> -X POST "https://{Appliance IP address}:443/pty/v2/pim/datastores"
-H "accept: application/json" -H "Content-Type: application/json" -d "{\"name\":\"DS\",\"description\": \"DS\",
\"default\":true}"
```

6.3.8 Deploying the Data Store

This section explains how you can deploy policies or trusted applications linked to a specific data store or multiple data stores.

For more information about deployment, refer to the *Policy Management Guide 8.0.0*.

6.3.8.1 Deploying a Specific Data Store

This section explains how you can deploy policies and trusted applications linked to a specific data store. The specifications provided for the specific data store are applied and becomes the end-result.

> **Note:** If you deploy an array with empty policies or trusted applications, or both, then the connected PEP servers contain empty definitions for these respective items.

Base URL

```
https://{Appliance IP address}/pty/v2
```

In the base URL, the Appliance IP address specifies the IP address of the required appliance.

Path

/pim/datastores/{dataStoreUid}/deploy

Method

POST

Sample Request

```
curl --insecure --user <username>:<password> -X POST "https://{Appliance IP address}:443/pty/v2/pim/datastores/{dataStoreUid}/deploy" -H "accept: application/json" -H "Content-Type: application/json" -d "{\"policies\":[\"1\"],\"applications\":[\"1\"]}"
```

6.3.8.2 Deploying Data Stores

This section explains how you can deploy data stores, which can contain the linking of either the policies or trusted applications, or both for the deployment.

> **Note:** If you deploy a data store containing an array with empty policies or trusted applications, or both, then the connected PEP servers contain empty definitions for these respective items.

Base URL

```
https://{Appliance IP address}/pty/v2
```

In the base URL, the Appliance IP address specifies the IP address of the required appliance.

Path

/pim/deploy

Method

POST

Sample Request

```
curl --insecure --user <username>:<password> -X POST "https://{Appliance IP address}:443/pty/v2/pim/deploy" -H "accept: application/json" -H "Content-Type: application/json" -d "{\"dataStores\":[{\"uid\":\"1\",\"policies\":[\"1\"],\"applications\":[\"1\"]},{\"uid\":\"2\",\"policies\":[\"2\"],\"applications\":[\"2\"]}]}"
```

6.3.9 Getting the Deployment Information

This section explains how you can check the complete deployment information. This service returns the list of the data stores with the connected policies and trusted applications.

> **Note:** The result might contain data store information that is pending deployment after combining the Policy Management operations performed through the ESA Web UI and DevOps API.

Base URL

```
https://{Appliance IP address}/pty/v2
```

In the base URL, the Appliance IP address specifies the IP address of the required appliance.

Path

/pim/deploy

Method

GET

Sample Request

```
curl --insecure --user <username>:<password> -X GET "https://{Appliance IP address}:443/pty/v2/pim/deploy" -H "accept: application/json"
```

6.4 Generating the DevOps REST API Samples

If you are working with Swagger Editor to access the *pim-devops.yaml* API specification document, then you can generate Curl samples using the Swagger Editor.

For more information about getting started with Swagger Editor, refer to the section *Getting Started*.

Perform the following steps to generate samples using the Swagger Editor.

1. On the Swagger Editor UI, click on the required API request.
2. Click **Try it out**.
3. Enter the parameters for the API request.
4. Click **Execute**.

   The generated Curl command and the URL for the request appears in the *Responses* section.

Chapter 7

Appendix B: APIs for Immutable Protectors

This section provides information about APIs that can be used to get the supported PEP server (pepserver) versions, log levels set, IMPS service health status, and the API document. The APIs also include the *export* API that is used to export the immutable policy, which is used by the Immutable Gen 2 protectors.

Important:
The permission to export the IMP package needs to be enabled through the ESA Web UI.

Perform the following steps to enable the Export IMP permission for a user.

1. On the ESA Web UI, navigate to the **Settings** > **Users** > **Roles** Tab.
2. Click on the **Role** for which you need to enable the permission to Export IMP.
   The selected role screen appears.
3. Select the **Export IMP** check box and Click **Save**.

Before you begin:

- Ensure that the concept of immutable protectors and necessity of an immutable policy is understood.
  For more information about the immutable policy concept, refer to the section *Application Protector (AP) Java Immutable Policy User Guide*.
- Ensure that the **IMPS** service is running on the ESA.

7.1 Viewing the Immutable Service API Specification with Swagger UI

This section describes the steps that you can perform to access and view the Immutable Service API specification document. Docker containers and services are used to download and launch the images for Swagger editor within a Docker container.

Before you begin

- Ensure that concepts of Docker and containers is understood.
  For more information about Docker, refer to the Docker documentation.

- Docker is installed on the machine where the Swagger editor is launched.

1. Install and start the Swagger Editor.
2. Download the Swagger Editor image within a Docker container using the following command.

```
docker pull swaggerapi/swagger-editor
```

3. Launch the Docker container using the following command.

```
docker run -d -p 8888:8080 swaggerapi/swagger-editor
```

4. Paste the following address on a browser window to access the Swagger Editor using the specified host port.

```
http://localhost:8888/
```

5. Download the Immutable Service API specification document using the following command.

```
curl --user <ESA username>:<ESA password> "https://ESA-ip-address/pty/v1/imp/doc" -H
"accept: application/x-yaml" --output imps.yaml
```

6. Drag and drop the downloaded *imps.yaml* file into a browser window.

7.2 Supported Authentication Methods for Immutable Service APIs

The following authentication methods can be used to establish a connection with the ESA:

- Basic authentication with the ESA user name and password
- Client Certificate-based authentication
- Token-based authentication

For more information about accessing the ESA using these authentication mechanisms, refer to the section *Appliance Overview Guide 9.0.0.0*.

7.3 Using the Immutable Service APIs

This section lists the Immutable Service APIs along with a sample API.

The following table lists the Immutable Service IMP REST APIs.

Table 7-1: List of Immutable Service REST APIs

REST API	Description
Retrieve the supported PEP server versions	This API retrieves the Immutable service version, Immutable service build version, and the supported PEP server versions.
Retrieve the Immutable Service health information	This API request retrieves the Immutable Service health information and identifies whether the service is running.
Retrieve the API specification document	This API request retrieves the API specification document.
Retrieve the Log Level	This API request retrieves current log level set for the Immutable Service logs.
Set log level for the Immutable Service log	This API request changes the Immutable Service log level during run-time. The level set through this resource is persisted until the IMP service is restarted. This log level overrides the log level defined in the configuration.
Export policy from a PEP server version	This API request exports the immutable policy that can be used with Immutable Gen 2 protectors.

7.3.1 Retrieving the Supported PEP Server Versions

This API retrieves the PEP server versions that are supported by the IMP service on the ESA.

Base URL

https://{ESA IP address}/pty/v1/imp

Path

/version

Method

GET

CURL request syntax

curl -k --user <username>:<password> -X GET "https://{ESA IP address}/pty/v1/imp/version"

Request parameters

username

ESA user name

password

ESA password

Sample CURL request

curl -k --user user:user1234 -X GET "https://10.10.101.43/pty/v1/imp/version"

Sample CURL response

```
{
  "version" : "1.1.0",
  "buildVersion" : "1.1.0+17.g1234.1.1",
  "pepVersions" : [  "1.1.0+85, 1.2.0+17" ]
}
```

7.3.2 Retrieving the IMPS Service Health Information

This API request retrieves the IMPS service health information and identifies whether the service is running.

Base URL

https://{ESA IP address}/pty/v1/imp

Path

/health

Method

GET

CURL request syntax

curl -k --user <username>:<password> -X GET "https://{ESA IP address}/pty/v1/imp/health"

Request parameters

username

ESA user name

password

ESA password

Sample CURL request

curl -k --user user:user1234 -X GET "https://10.10.101.43/pty/v1/imp/health"

Sample CURL response

```
{
  "isHealthy" : true
}
```

where,

- **isHealthy: true** - Indicates that the service is up and running.
- **isHealthy: false** - Indicates that the service is down.

7.3.3 Retrieving the API Specification Document

This API request retrieves the API specification document.

Base URL

https://{ESA IP address}/pty/v1/imp

Path

/doc

Method

GET

CURL request syntax

curl -k --user <username>:<password> -X GET "https://{ESA IP address}/pty/v1/imp/doc"

Request parameters

username

ESA user name

password

ESA password

Sample CURL request

curl -k --user user:user1234 -X GET "https://10.10.101.43/pty/v1/imp/doc"

Sample CURL response

The IMP API specification document is displayed as a response.

7.3.4 Retrieving the Log Level

This API request retrieves current log level set for the IMPS service logs.

Base URL

https://{ESA IP address}/pty/v1/imp

Path

/log

Method

GET

CURL request syntax

curl -k --user <username>:<password> -X GET "https://{ESA IP address}/pty/v1/imp/log"

Request parameters

username

ESA user name

password

ESA password

Sample CURL request

curl -k --user user:user1234 -X GET "https://10.10.101.43/pty/v1/imp/log"

Sample CURL response

```
{
  "level": "FINE"
}
```

7.3.5 Setting Log Level for the IMPS Service Log

This API request changes the IMPS service log level during run-time. The level set through this resource is persisted until the IMP service is restarted. This log level overrides the log level defined in the configuration.

Base URL

https://{ESA IP address}/pty/v1/imp

Path

/log

Method

POST

CURL request syntax

curl -X POST "https://{ESA IP Address}/pty/v1/imp/log" --user "{ username:password}" -H "accept: application/json" -H "Content-Type: application/json" -d "{\"level\":\"log level\"}"

Request body elements

username

ESA administrator user name

password

ESA administrator password

log level

Set the log level. The log level can be set to *SEVERE*, *WARNING*, *INFO*, *CONFIG*, *FINE*, *FINER*, or *FINEST*.

Sample CURL request

curl -X POST "https://{ESA IP Address}/pty/v1/imp/log" --user "{ username:password}" -H "accept: application/json" -H "Content-Type: application/json" -d "{\"level\":\"SEVERE\"}"

Sample response

The log level is set successfully.

7.4 Sample Immutable Service API - Exporting Policy from a PEP Server Version

This API request exports the immutable policy that can be used with Immutable Gen 2 protectors.

> **Note:** Ensure that the *EXPORT IMP* permission is granted to the role that is assigned to the user exporting the policy from the ESA.
> For more information about how permissions and roles are related, refer to section *Managing Roles* in the *Appliance Overview Guide 9.0.0.0*.

> **Warning:** Do not modify the policy that has been exported using the Immutable Service API.

If you try to modify the exported policy, then the policy might get corrupted.

Base URL

https://{ESA IP address}/pty/v1/imp

Path

/export

Method

POST

CURL request syntax

Export API - PKCS5

```
curl -u "{username}:{password}" -X POST https://{ESA_IP_address}/pty/v1/imp/export -H
"Content-Type: application/json" --data '{
    "name": "{name}",
    "pepVersion": "{pepVersion}",
    "dataStoreName": "{dataStoreName}",
    "kek":{
        "pkcs5":{
            "salt":"{salt}",
            "password":"{password}"
        }
    }
}' -o imps.json
```

Authentication credentials

username

ESA user name

password

ESA password

Request body elements

name

Set the name for the exported policy.

pepVersion

Set the PEP version. Use the */version* API to retrieve the supported PEP versions.

dataStoreName

Set the Data Store name where the policy is deployed on ESA. Login to the ESA and retrieve the name of the data store.

Encryption Method

The **pkcs5** encryption can be used to protect the exported file.

Encryption Method	Sub-elements	Description
pkcs5	**salt**	Set the salt that will be used to encrypt the exported policy.
	password	Set the password that will be used to encrypt the exported policy.

Sample CURL request

Export API - PKCS5

```
curl -u "admin:admin123" -X POST https://10.39.1.144/pty/v1/imp/export -H "Content-
Type: application/json" --data '{
    "name": "exportimps",
    "pepVersion": "1.2.0+17",
    "dataStoreName": "Datastore",
    "kek":{
```

```
        "pkcs5":{
            "salt":"GtYhY",
            "password":"protect123"
        }
    }
}' -o imps.json
```

Sample response

The ***imp.json*** file is exported.

> **Note:**
> After the policy is exported, you must store it in an object storage.
>
> For more information about importing the policy, refer to the respective Immutable Protector documentation.

> **Important:** The exported policy can only be used with 64-bit Linux protectors.