Protegrity Protection Method Reference Guide 9.1.0.4

Created on: Nov 19, 2024

Copyright

Table of Contents

Chapter 1

Introduction to This Guide

This guide provides an overview of protection methods supported by Protegrity products. It explains properties of protection methods, and provides examples of their usage.

The document is intended for Security Officers who will create and manage data security policies. The document guides you through Protegrity protection methods, providing a comparison of all of the methods, and helping you to select the method that will fit your enterprise and specific business requirements appropriately.

This guide should be used along with the *Protegrity Enterprise Security Administrator Guide 9.1.0.0*, which explains the mechanism of managing data security policy.

It is recommended that you first read the sections explaining the properties of protection methods. It will give you better understanding of the properties applicable to each specific protection method.

1.1 Sections contained in this Guide

The guide is broadly divided into the following sections:

- Section 1 *Introduction to This Guide* defines the purpose and scope for this guide. In addition, it explains how information is organized in this guide.
- Section 2 *Protegrity Protection Methods Overview*provides an overview about the Protegrity Protection Methods.
- Section 3 *Protegrity Tokenization* provides information about Tokenization and tokenization types.
- Section 4 *Protegrity Format Preserving Encryption* provides information about a data encryption technique that preserves the ciphertext format using FF1 mode of operation for AES-256 block cipher algorithm.
- Section 5 *Protegrity Encryption* provides information about the Protegrity Encryption algorithms.
- Section 6 *Monitoring and Blocking (No Encryption)* provides information about monitoring and Blocking data where encryption is not used for data protection.
- Section 7 *Hashing* provides information about the Hashing data protection algorithm.
- Section 8 *Masking* provides information about the masking method.
- *Appendix A: ASCII Character Codes* provides a table that lists the ASCII Character Codes.
- *Appendix B: Examples of Column Sizes Calculation for Encryption* provides a table that contains the Column Sizes Calculation for 3DES encryption.
- *Appendix C: Empty String Handling by Protectors* provides information about how Protectors provide Empty String Handling support on different systems.
- *Appendix D: NULL Handling by Protectors* provides information about the behavior of Protectors on different systems when NULL value is the input.

- *Appendix E: Hashing Functions and Examples* provides information about the Hashing functions and examples.
- *Appendix F: Codebook Re-shuffling in the Data Security Gateway (DSG)* provides information about the Codebook Re-shuffling feature.

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

Protegrity Protection Methods Overview

Protegrity products can protect sensitive data with the following protection methods:

- Tokenization
- Format Preserving Encryption (FPE)
- Encryption
- Hashing
- No Encryption
- Monitoring
- Masking

The following table describes the protection methods available for each data security policy type (structured and unstructured protection). For more information, refer to *Protegrity Enterprise Security Administrator Guide 9.1.0.0*.

Table 2-1: Protection Methods by Data Security Policy Type

Protection Method	***Description***	***Structured***	***Unstructured***
Tokenization (all types)	Bound to the token element created on ESA. For more information, refer to section 3 *Protegrity Tokenization*.	√	
Format Preserving Encryption (FPE)	A data encryption technique that preserves the ciphertext format using FF1 mode of operation for AES-256 block cipher algorithm. For more information, refer to section 4 *Protegrity Format Preserving Encryption*.	√	
3DES	A block cipher with 168 bit encryption keys.	√	√
AES-128	A block cipher with 128 bit encryption keys.	√	√
AES-256	A block cipher with 256 bit encryption keys.	√	√
CUSP 3DES, CUSP AES-128, CUSP AES-256	A modified block algorithm mainly used in environments where an IBM mainframe is present.	√	
No Encryption	Does not protect data at rest by changing it. Protection comes from monitoring and masking.	√	
Monitoring	Does not protect data at rest by changing it. Used for monitoring and auditing.	√	
Masking	Does not protect data at rest by changing it. Protection comes from masking.	√	
Hashing (HMAC-SHA1)	A Keyed-Hash Message Authentication Code. Used only for protection of data. Since hashing is a one-way function, the original data cannot be restored.	√	

Protegrity protection methods (tokenization, encryption, no encryption, monitoring, masking, and hashing) support a number of input formats so that you can protect sensitive data such as:

- Social Security Numbers (SSNs)
- Credit Card Numbers (CCNs)
- Electronic Personal Health Information (ePHI), which is controlled by Health Insurance Portability and Accountability Act (HIPPA) and Health Information Technology for Economic and Clinical Health (HITECH)

- Personally identifiable information (PII)

The following table shows different types of sensitive data that can be protected with the Protegrity platform and demonstrates input values and their corresponding protected values.

Table 2-2: Examples of Protected Data

#	*Type of Data*	*Input*	*Protected Value*	*Comment*
1	SSN delimiters	075-67-2278	287-38-2567	Numeric token, delimiters in input
2	SSN delimiters	075-67-2278	731-80-3403	delimiters in input
3	Credit Card	5511 3092 3993 4975	8278 2789 2990 2789	Numeric token
4	Credit Card	5511 3092 3993 4975	5097 4431 6333 9030	Numeric encryption
5	Credit Card	5511 3092 3993 4975	8278 2789 2990 **4975**	Numeric token, last 4 digits in clear
6	Credit Card	5511309239934975	551130##########	No Encryption with mask exposing the first 6 digits. A mask is applied by the data security policy when a user tries to view the protected value.
7	Credit Card	5511309239934975	1437623387940746	Credit Card token with invalid Luhn digit property. Tokenized value has invalid Luhn checksum.
8	Credit Card	5511309239934975	**8**313123036143103	Credit Card token with invalid card type identification. The first digit in tokenized value is not a valid card type.
9	Credit Card	5511309239934975	1854817**J**97347370	Credit Card token with alphabetic indicator on 8th position
10	Phone/Fax number	1 888 397 8192	9 853 888 8435	Numeric token
11	Phone/Fax number	1 888 397 8192	4 244 020 3311	Numeric encryption
12	Medical ID	29M2009ID	iA6wx0Mw1	Alpha-Numeric token
13	Medical ID	29M2009ID	vOjaewxdO	Alpha-Numeric encryption
14	Date	10/30/1955	12/25/2034	Date token
15	Date with month in clear	2009.04.12	1595.**04**.19	Datetime token, month is not tokenized
16	Date and Time	2012.12.31 12:23:34	1816.07.22 **14:31:51**	Datetime token, date and time parts are tokenized
17	Date which should be distinguishable	2012.12.31	**7867**.03.12	Datetime token, distinguishable year
18	Proper names	Alfred Hitchcock	uRLzbg cvofdBFJh	Alpha token
19	Proper names	Alfred Hitchcock	UMWLDq WkXOEeDEK	Alpha encryption
20	Short names	Al	kKX	Alpha token non-length preserving
21	Abbreviations	CXR	GTP	Upper-case Alpha token
22	License plates	583-LBE	44J-KLT	Upper Alpha-Numeric token
23	Addresses	5 High Ridge Park, Stamford	5 hcY2 k9rLp Z0uA, KunZYNEM	Alpha-Numeric token. Punctuation marks and spaces are treated as delimiters.
24	Addresses	5 High Ridge Park, Stamford	T 0xqm BoUXY 7Vva, Eckv3Bc1	Alpha-Numeric encryption. Punctuation marks and spaces are treated as delimiters.
25	E-mail Address	a@gmail.com	h@MSGXJ**.com**	Alpha-Numeric token, delimiters in input, last 3 characters in clear
26	E-mail Address	a@gmail.com	y@7FD5B.acb	Alpha-Numeric encryption, delimiters in input
27	Strong passwords	2$trongPa$$	KÒý¿ú!â¥%¸_	Printable token

#	Type of Data	Input	Protected Value	Comment
28	Strong passwords	2$trongPa$$	A[ZæJß÷ý:·	Printable encryption
29	Fuzzy times	1994-01-01_00.00.00	Û-TIÇLæpj\aDv4ë}{9	Printable token
30	Fuzzy times	1994-01-01_00.00.00	UÚ8Ã¯7·äMÊô÷Àzµx|°l	Printable encryption
31	Unicode text	€¥©ă÷	CaqRppjFPwY22w5fDhUkxNG	Unicode token
32	Unicode text	Протегрити	mbiVtZxJonVWS7IPB6yz6tztblYBfkj	Unicode token
33	Unicode Base64	Москва	BftgxVX0t+O+I8v	Unicode Base64 token
34	Unicode Gen2	МОСКВб	Ууыш4х	Unicode Gen2 token with custom alphabets as defined in the data element.
35	Financial data	-3015.039	-4416.646	Decimal token. Protected value will never contain any zeroes.
36	Photographic images, media files	Media stored as BLOB type	Encrypted BLOB	Encryption (AES-256, AES-128, 3DES) or hashing (HMAC-SHA1)
37	Irreversible data to be destroyed	AnyDataTo Destroy	Q2LKa2UhIhMTiRsi0l8BUF5xVag=	Hashing (HMAC-SHA1), data cannot be decrypted

You can combine Protegrity protection methods to obtain the required level of data access control within the enterprise.

For example, a Security Officer can use a data security policy to control what is delivered to different roles in the policy. The following figure shows how Social Security Number access can vary by different users and applications.

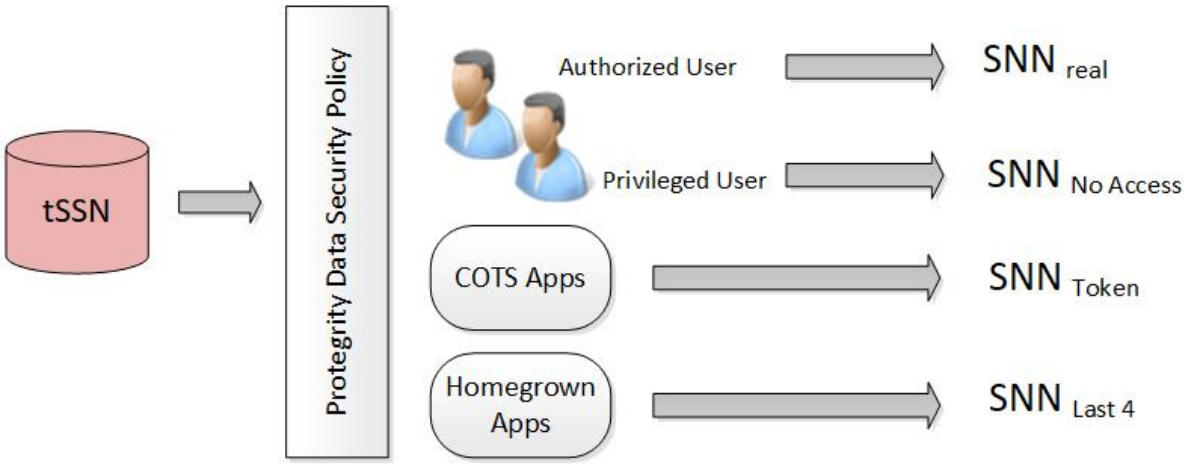

Figure 2-1: SSN Access

In the previous figure, the tokenized SSN is stored in the database. However, there are four roles defined in the policy: authorized users, privileged users, off-the-shelf application users, and homegrown application users. Each role can receive a different form of the SSN based on its need. The Security Officer determines the SSN form by role.

Protegrity tokenization maintains a separation of duties by way of the data security policy. The DBA, Developers, and System Administrators do not have access to the data, even if they manage the system, because everything goes through the data security policy.

For more information about data security policies, refer to the Managing Security Policies section in the *Enterprise Security Administrator Guide 9.1.0.0.*

Chapter 3

Protegrity Tokenization

Tokenization is the process of replacing sensitive data with an inert value that has no worth to someone who gains unauthorized access to the data. With tokenization, specific pieces of original data can be preserved, while the system tokenizes data according to design. Once tokenization is deployed, operational systems continually work with the tokens. If the operational systems experience a security breach, then only the tokens are at risk of being compromised.

Protegrity tokenization is a tokenization method that is optimized to meet the performance, scalability, and manageability requirements of large and complex environments. Tokens can be set up and deployed directly on the protection point, depending on your enterprise configuration and data security needs. Protegrity tokenization is transparent to end-users. Data integrity is strongly enforced by way of the data security policy.

Protegrity tokenization can protect many different types of data using Numeric, Integer, Alpha, Upper-case Alpha, Alpha-Numeric, Upper Alpha-Numeric, Lower ASCII, Printable, Date, Datetime, Credit Card, Decimal, Unicode, Unicode Base64, Unicode Gen2, Binary, and Email token types. The token specification can also be used to preserve different parts of the original value in the token, such as the last 4 digits. Protegrity tokenization also recognizes and preserves delimiters, which are often used in SSNs, dates, etc.

Protegrity tokenization allows you to tokenize payment card industry (PCI) data and a variety of input data types, such as personally identifiable information (PII), and protected health information (PHI) to comply with industry regulations.

With Protegrity tokenization, there is a 1:1 relation between the real data value and its token value. This enables token values to be used as an alternative unique ID that can be used for joining related information.

The following table describes the token types supported by Protegrity tokenization.

Table 3-1: Tokenization Types

Tokenization Type	***Alphabet Characters***	***Comment***
Numeric (0-9)	Digits 0 through 9	
Integer	Digits 0 through 9	Data length: 2 bytes, 4 bytes, and 8 bytes
Credit Card	Digits 0 through 9	Special settings: Invalid LUHN digit, invalid card type, alphabetic indicator
Alpha (a-z, A-Z)	Lowercase letters a through z Uppercase letters A through Z	
Upper-case Alpha (A-Z)	Uppercase letters A through Z	Lower case characters will be converted to upper-case in output value

Tokenization Type	*Alphabet Characters*	*Comment*
Alpha-Numeric (0-9, a-z, A-Z)	Digits 0 through 9 Lowercase letters a through z Uppercase letters A through Z	
Upper Alpha-Numeric (0-9, A-Z)	Digits 0 through 9 Uppercase letters A through Z	Lower case characters will be converted to upper-case in output value
Lower ASCII	The lower part of ASCII table. Hex character codes from 0x21 to 0x7E	Support of 94 printable characters (ASCII from 33 (!) to 126(~)), the rest are treated as delimiters
Printable	ASCII printable characters, which include letters, digits, punctuation marks, and miscellaneous symbols. Hex character codes from 0x20 to 0x7E, and from 0xA0 to 0xFF	ISO 8859-15 Latin alphabet no. 9
Date YYYY-MM-DD	Date in big endian form, starting with the year. The following separators are supported: .(dot), / (slash), - (dash).	
Date DD/MM/YYYY	Date in little endian form, starting with the day. The following separators are supported: . (dot), / (slash), - (dash).	
Date MM.DD.YYYY	Date in middle endian form, starting with the month. The following separators are supported: . (dot), / (slash), - (dash) supported.	
Datetime	YYYY-MM-DD HH:MM:SS	Special settings : Tokenize time, Distinguishable date, Date in clear
Decimal	Digits 0 through 9 sign and .(decimal delimiter)	Numeric data with precision and scale. The token will not contain any zeros.
Unicode	UTF-8 text. Hex character codes from 0x00 to 0xFF	Result is Alpha-Numeric. Supported by Application protectors, Big Data protector, and Teradata Database protector.
Unicode Base64	UTF-8 text. Hex character codes from 0x00 to 0xFF	Result is Alpha-Numeric, +, /, and =. Supported by Application protectors, Big Data protector, and Teradata Database protector.
Unicode Gen2	Unicode code points between U+0020 and U+3FFFF	Result is based on the alphabet selected while creating the token.
Binary	Hex character codes from 0x00 to 0xFF	Supported by Application protectors
Email	Digits 0 through 9 Lowercase letters a through z Uppercase letters A through Z Special characters with restrictions @ sign and .(dot) are delimiters	Domain part after @ sign will not be tokenized

3.1 Delimiters

Protegrity tokenization can generate the same token regardless of how the data is formatted. Any input that does not comply with the token types in the previous table is treated as a delimiter and remains unchanged during tokenization.

The following table shows how the system handles delimiters and spaces as compared to plain numerical data.

Table 3-2: Tokenization with Delimiters

Input	*CCN*	*Token*	*Value returned by Protegrity tokenization*
5332711989955364	5332711989955364	8344588301109112	8344588301109112
5332-7119-8995-5364	5332711989955364	8344588301109112	8344-5883-0110-9112
5332 7119 8995 5364	5332711989955364	8344588301109112	8344 5883 0110 9112

3.2 Support by Protegrity Products

Tokenization is supported on protectors supporting structured policies.

Application protectors support all types of tokens. Database protectors have limitations on support of Unicode and Binary token types.

The following four tables list the Data Types to be used with different tokenization types across Application, Databases, and Big Data Protectors.

Table 3-3: Supported Tokenization Types by Application Protector

Tokenization Type[*1]	*Application Protector*					
	AP Python	**AP Java**	**AP C**	**AP Go**	**AP .Net**	**AP NodeJS**
Credit Card Numeric Alpha Upper-case Alpha Alpha-Numeric Upper Alpha-Numeric Printable[*2] Lower ASCII Email	STRING BYTES	STRING CHAR[] BYTE[]	BYTE[]	STRING []BYTE	STRING BYTE[]	STRING BYTE[]
Integer	INT: 4 bytes and 8 bytes	SHORT: 2 bytes INT: 4 bytes LONG: 8 bytes	BYTE[]	SHORT: 2 bytes INT: 4 bytes LONG: 8 bytes	STRING BYTE[]	STRING BYTE[]

Tokenization Type[*1]	*Application Protector*					
	AP Python	**AP Java**	**AP C**	**AP Go**	**AP .Net**	**AP NodeJS**
Date Datetime	DATE STRING BYTES	DATE STRING CHAR[] BYTE[]	BYTE[]	STRING []BYTE	STRING BYTE[]	STRING BYTE[]
Decimal	STRING BYTES	STRING CHAR[] BYTE[]	BYTE[]	STRING []BYTE	STRING BYTE[]	STRING BYTE[]
Unicode	STRING BYTES	STRING CHAR[] BYTE[]	BYTE[]	STRING []BYTE	STRING BYTE[]	STRING BYTE[]
Unicode Base64	STRING BYTES	STRING CHAR[] BYTE[]	BYTE[]	STRING []BYTE	STRING BYTE[]	STRING BYTE[]
Unicode Gen2	STRING BYTES	STRING CHAR[] BYTE[]	BYTE[]	STRING []BYTE	STRING BYTE[]	STRING BYTE[]
Binary	BYTES	BYTE[]	BYTE[]	BYTE[]	BYTE[]	BYTE[]

Note:

[*1]- If the input and output types of the API are BYTE[], then the customer application should convert the input to and output from the byte array, before calling the API.

[*2]- Printable token data element is unsupported by the AP .Net and AP NodeJS.

Table 3-4: Supported Tokenization Types by Database Protectors

Tokenization Type	*Database*		
	MSSQL Server	**Oracle**	**DB/2**
Credit Card Numeric	VARCHAR CHAR	VARCHAR2 CHAR	VARCHAR CHAR

Tokenization Type	*Database*		
	MSSQL Server	**Oracle**	**DB/2**
Alpha Upper-case Alpha Alpha-Numeric Upper Alpha-Numeric Printable Lower ASCII Email			
Integer	INTEGER	INTEGER	INTEGER
Date Datetime	VARCHAR CHAR	DATE VARCHAR2 CHAR	DATE VARCHAR CHAR
Decimal	VARCHAR CHAR	NUMBER VARCHAR2 CHAR	VARCHAR2 CHAR
Unicode	Not supported	Not supported	VARCHAR
Unicode Base64	Not supported	VARCHAR2 NVARCHAR2	VARCHAR
Unicode Gen2	Not supported	VARCHAR2 NVARCHAR2	VARCHAR
Binary	Not supported	Not supported	Not supported

Table 3-5: Supported Tokenization Types for MPP Database Protectors

Tokenization Type	*MPP Database*		
	GPDB	**Teradata**	**IBM Netezza**
Credit Card Numeric Alpha Upper-case Alpha Alpha-Numeric	VARCHAR	VARCHAR LATIN CHAR LATIN	VARCHAR

Tokenization Type	*MPP Database*		
	GPDB	**Teradata**	**IBM Netezza**
Upper Alpha-Numeric Printable Lower ASCII Email			
Integer	INTEGER	INTEGER	INTEGER
Date Datetime	DATE VARCHAR	VARCHAR LATIN CHAR LATIN	DATE VARCHAR
Decimal	VARCHAR	VARCHAR LATIN CHAR LATIN	VARCHAR
Unicode	Not supported	VARCHAR UNICODE CHAR UNICODE	Not supported
Unicode Base64	Not supported	Not supported	Not supported
Unicode Gen2	Not supported	Not supported	Not supported
Binary	Not supported	Not supported	Not supported

Table 3-6: Supported Tokenization Types for Big Data Protectors

Tokenization Type[*1]	*Big Data*							
	MapReduce	**Hive**	**Pig**	**HBase**	**Impala**	**Spark**	**Spark SQL**	**Presto**
Credit Card Numeric Alpha Upper-case Alpha Alpha-Numeric Upper Alpha-Numeric Lower ASCII Email	BYTE[]	STRING	CHARARRAY	BYTE[]	STRING	VARCHAR STRING	STRING	VARCHAR
Integer	INT: 4 bytes	INT: 4 bytes	INT: 4 bytes	BYTE[]	INT: 4 bytes	SHORT: 2 bytes INT: 4 bytes	SHORT: 2 bytes INT: 4 bytes	SMALL INT: 2 bytes INT: 4 bytes

Tokenization Type[*1]	*Big Data*							
	MapReduce	**Hive**	**Pig**	**HBase**	**Impala**	**Spark**	**Spark SQL**	**Presto**
	LONG: 8 bytes	BIGINT: 8 bytes				LONG: 8 bytes	LONG: 8 bytes	BIGINT: 8 bytes
Date Datetime [*2]	BYTE[]	STRING DATE DATETIME	CHARARRAY	BYTE[]	STRING	BYTE[] STRING	STRING DATE DATETIME	VARCH AR DATE TIMES TAMP
Decimal	BYTE[]	STRING	CHARARRAY	BYTE[]	STRING	BYTE[] STRING	STRING	VARCHAR
Printable	BYTE[]	Not supported	Not supported	BYTE[]	STRING	BYTE[]	Not supported	Not supported
Unicode	BYTE[]	STRING	Not supported	BYTE[]	STRING	BYTE[] STRING	STRING	VARCHAR
Unicode Base64	BYTE[]	STRING	Not supported	BYTE[]	STRING	BYTE[] STRING	STRING	VARCHAR
Unicode Gen2	BYTE[]	STRING	Not supported	BYTE[]	STRING	BYTE[] STRING	STRING	VARCHAR
Binary	BYTE[]	Not supported	Not supported	BYTE[]	Not supported	BYTE[]	Not supported	Not supported

Note:

[*1] - The customer application should convert the input to and output from byte array.

[*2] - The Datetime tokenization will only work with VARCHAR data type.

[*3] - The Char tokenization UDFs only support Numeric, Alpha, Alpha Numeric, Upper-case Alpha, Upper Alpha-Numeric, and Email data elements, and with length preservation selected. Using any other data elements with Char tokenization UDFs is not supported. Using non-length preserving data elements with Char tokenization UDFs is not supported.

Note:

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.

For more information, refer to section *Truncating White Spaces*.

Note: For zOS databases like DB2/IMS, data including the KEY fields or SSAs can be protected. However, after data protection, the collating order of the data will be impacted if the search criteria includes the KEY field or SSA.

3.3 Tokenization Properties

The properties for a token element are specified when the token element is created. Some are input by the end-user, and some are calculated.

Table 3-7: Common Tokenization Properties

Token Property	***Description***
User configured token properties	
Name	Unique name identifying the token element. Maximum length is 56 characters.
Token type and Format	Type of data to tokenize. Name of the alphabet, which indicates the specific characters to tokenize.
Tokenizer	How many and what kind of lookup tables to generate (SLT_1_3, SLT_1_6, SLT_2_3, SLT_2_6, SLT_6_DECIMAL, SLT_DATETIME, and SLT_X_1). **Note:** The newly created data elements using the SLT_2_6 tokenizer from v7.1 Maintenance Release 1 (MR1) onwards are deployable to protectors with versions 7.1 MR1 and higher. **Note:** The *SLT_X_1* tokenizer can only be used to create the data elements for protectors with version 9.1.0.0 and higher and with the *Unicode Gen2* token type.
Preserve Case	Whether the case of the alphabets and position of the alphabets and numbers must be preserved when tokenizing the value. This is applicable when using the *Alpha-Numeric (0-9, a-z, A-Z)* token type and the *SLT_2_3* tokenizer only.
Preserve Position	Whether the position of the alphabets and numbers must be preserved when tokenizing the value. This is applicable when using the *Alpha-Numeric (0-9, a-z, A-Z)* token type and the *SLT_2_3* tokenizer only.
Preserve length	Whether tokens will be the same length as the input or not.
Allow Short Tokens	Whether short tokens will be enabled or not (Possible options are Yes, No, generate error, or No, return input as it is).
Left	Number of characters from left to keep in clear.
Right	Number of characters from right to keep in clear.
Minimum input length	Shortest length that can be tokenized.
Maximum input length	Longest length that can be tokenized.
Automatically calculated token properties	
External IV	Whether external initialization vector (IV) will be used or not.

Token Property	*Description*
Other token properties	
Internal IV	Whether internal initialization vector (IV) will be used or not.

All the options cannot be combined with all types of tokens. The following table shows what properties can be set for the various types.

Table 3-8: Tokenization Properties by Token Types

Tokenization Type	*Tokenizer*	*Preserve length*	*Preserve Case/ Preserve Position*	*Allow Short Tokens*	*Left, Right*	*Minimum/ Maximum length*	*External IV*	*Internal IV*
Numeric	SLT_1_3, SLT_2_3, SLT_1_6, SLT_2_6[*1]	√	X	√	√	X	√	√
Integer	SLT_1_3	√	X	X	X	X	X	X
Credit Card	SLT_1_3, SLT_2_3, SLT_1_6, SLT_2_6[*1]	√ (always yes)	X	X	√	X	√	√
Alpha	SLT_1_3, SLT_2_3	√	X	√	√	X	√	√
Upper-case Alpha	SLT_1_3, SLT_2_3	√	X	√	√	X	√	√
Alpha-Numeric	SLT_1_3, SLT_2_3	√	X	√	√	X	√	√
	SLT_2_3	√	√	√	√	X	√	√
Upper Alpha-Numeric	SLT_1_3, SLT_2_3	√	X	√	√	X	√	√
Lower ASCII	SLT_1_3	√	X	√	√	X	√	√
Printable	SLT_1_3	√	X	√	√	X	√	√
Date YYYY-MM-DD	SLT_1_3, SLT_2_3, SLT_1_6, SLT_2_6[*1]	√ (always yes)	X	X	X (0, 0)	X	X	X
Date DD/MM/YYYY	SLT_1_3, SLT_2_3, SLT_1_6, SLT_2_6[*1]	√ (always yes)	X	X	X (0, 0)	X	X	X
Date MM.DD.YYYY	SLT_1_3, SLT_2_3, SLT_1_6, SLT_2_6[*1]	√ (always yes)	X	X	X (0, 0)	X	X	X
Datetime	SLT_DATETIME	√ (always yes)	X	X	X (0, 0)	X	X	X
Decimal	SLT_6_DECIMAL	X (always no)	X	X	X (0, 0)	√	X	X
Unicode	SLT_1_3, SLT_2_3	X (always no)	X	√	X (0, 0)	X	√	X
Unicode Base64	SLT_1_3, SLT_2_3	X (always no)	X	√	X (0, 0)	√	√	X
Unicode Gen2	SLT_1_3	√	X	√	√	√	√	√

Tokenization Type	*Tokenizer*	*Preserve length*	*Preserve Case/ Preserve Position*	*Allow Short Tokens*	*Left, Right*	*Minimum/ Maximum length*	*External IV*	*Internal IV*
	SLT_X_1							
Binary	SLT_1_3, SLT_2_3	X (always no)	X	X	√	X	√	√
Email	SLT_1_3, SLT_2_3	√	X	√	X (0, 0)	X	√	X

X - means that Property is disabled and cannot be specified

√ - means that Property is enabled or can be specified

> **Note:** *1The newly created data elements using the SLT_2_6 tokenizer from v7.1 Maintenance Release 1 (MR1) onwards are deployable to protectors with versions 7.1 MR1 and higher.

3.3.1 Token Type and Format

The token type specifies the data that should be tokenized, for instance with the characters to expect as input and the output to generate.

The format is a name of an alphabet that contains all characters considered for tokenization, it is derived from the Token Type property. Characters outside the alphabet are considered to be delimiters.

Refer to the table *Tokenization Types* for the full list of supported token types.

3.3.2 Static Lookup Table (SLT) Tokenizers

A static lookup table (SLT) contains a pre-generated list of all possible values from a given set of characters. An alphabetic lookup table for instance might contain all values from ‘Aa’ to ‘Zz’. All entries are then shuffled so that they are in random order.

SLT tokenizer represents a method that uses multiple SLTs to generate tokens. This is done by first dividing the input value into smaller pieces, called token blocks, which correspond to entries in the lookup tables. The token blocks are then substituted with values from the SLTs and chained together to form the final token value. This means that the token is a result of multiple lookups in multiple SLTs.

Another benefit of SLT tokenizers is that tokenization can be done locally on the protection point. With this solution, tokenization is performed locally within the Protection Enforcement Point (PEP) environment.

Tokenization within the PEP environment is the main tokenization mode from the SP2 Release onwards.

For more information, refer to the Data Elements Deployment section in the *Protegrity Enterprise Security Administrator Guide 9.1.0.0*.

There are several types of SLT tokenizers from which you can choose. They are distinguished by their block size and the number of lookup tables.

Table 3-9: SLT Tokenizer with block size and lookup tables

Tokenizer	*Allow Short Tokens*	*No. of lookup tables*	*Block size*
SLT_1_3	Yes	1	1
		1	2

Tokenizer	*Allow Short Tokens*	*No. of lookup tables*	*Block size*
		1	3
	No, return input as it is No, generate error	1	3
SLT_2_3	Yes	2	1
		2	2
		2	3
	No, return input as it is No, generate error	2	3
SLT_1_6	Yes	1	1
		1	2
		1	3
		1	6
	No, return input as it is No, generate error	1	6
SLT_2_6*1	Yes	2	1
		2	2
		2	3
		2	6
	No, return input as it is No, generate error	2	6
SLT_6_DECIMAL	NA	Multiple lookup tables: One for each input length in the range 1 to 5 One for input lengths >= 6	
SLT_DATETIME	NA	Multiple lookup tables	
SLT_X_1	Yes	5-98*2	1
	No, return input as it is No, generate error	3-96*2	1

Note: *1 - The data elements created using the *SLT_2_6* tokenizer are not deployable to protectors lower than versions 7.1.

Note: *2 - For the *SLT_X_1* tokenizer, the number of lookup tables used for the security operations is determined during the creation of the data elements.

The following table describes the types of SLT tokenizers and compares their characteristics.

Table 3-10: SLT Tokenizer Characteristics

Token Type	*Tokenizer*	*Allow Short Tokens*	*Maximum Size of Token Tables (number of entries)*	*Size of Token Tables (kB)*	*Amount of Memory used in the Protector (kB)*	*Comments*
Numeric	SLT_1_3 SLT_2_3 SLT_1_6 SLT_2_6	No, generate error No, return input as it is	1,000 2*1,000 1,000,000 2*1,000,000	4 8 3,906 7,812	8 16 7,812 15,624	
		Yes	1,110 2*1,110 1,001,110 2*1,001,110	4.33 8.66 3,910.58 7,821.17	8.66 17.32 7,821.17 15,642.34	
Integer	SLT_1_3	NA	4096	16	32	
Credit Card	SLT 1_3 SLT 2_3 SLT 1_6 SLT 2_6	NA	1,000 2*1,000 1,000,000 2*1,000,000	4 8 3,906 7,812	8 16 7,812 15,624	
Alpha	SLT 1_3 SLT 2_3	No, generate error No, return input as it is	140, 608 2*140,608	549 1,098	1,098 2,196	
		Yes	143,364 2*143,364	560.01 1,120.02	1,120.02 2,240.04	
Upper-case Alpha	SLT 1_3 SLT 2_3	No, generate error No, return input as it is	17,576 2*17,576	69 138	138 276	
		Yes	18,278 2*18,278	71.39 142.79	142.79 285.59	
Alpha-Numeric	SLT 1_3 SLT 2_3	No, generate error No, return input as it is	238,328 2*238,328	931 1,862	1,862 3,724	

Token Type	*Tokenizer*	*Allow Short Tokens*	*Maximum Size of Token Tables (number of entries)*	*Size of Token Tables (kB)*	*Amount of Memory used in the Protector (kB)*	*Comments*
		Yes	242,234 2*242,234	946.22 1,892.45	1,892.45 3,784.90	
Upper Alpha-Numeric	SLT 1_3 SLT 2_3	No, generate error No, return input as it is	46,656 2*46,656	182 364	364 728	
		Yes	47,988 2*47,988	187.45 374.90	374.90 749.81	
Lower ASCII	SLT 1_3	No, generate error No, return input as it is	830,584	3,244	6,488	
		Yes	839,514	3,279.35	6,558.70	
Printable	SLT 1_3	No, generate error No, return input as it is	6,967,871	27,218	54,436	
		Yes	7,004,543	27,361.49	54,722.99	
Date YYYY-MM-DD	SLT_1_3 SLT_2_3 SLT_1_6 SLT_2_6	NA	1,000 2*1,000 1,000,000 2*1,000,000	4 8 3,906 7,812	8 16 7,812 15,624	
Date DD/MM/YYYY	SLT_1_3 SLT_2_3 SLT_1_6 SLT_2_6	NA	1,000 2*1,000 1,000,000 2*1,000,000	4 8 3,906 7,812	8 16 7,812 15,624	
Date MM.DD.YYYY	SLT_1_3 SLT_2_3 SLT_1_6	NA	1,000 2*1,000 1,000,000	4 8 3,906	8 16 7,812	

Token Type	*Tokenizer*	*Allow Short Tokens*	*Maximum Size of Token Tables (number of entries)*	*Size of Token Tables (kB)*	*Amount of Memory used in the Protector (kB)*	*Comments*
	SLT_2_6		2*1,000,000	7,812	15,624	
Datetime	SLT_DATETIME	NA	1,000,000 + 86,400	4,244	8,488	Maximum memory is used when both date part and time part will be tokenized
Decimal	SLT_6_DECIMAL	NA	597,870	2,335	4,670	
Unicode	SLT_1_3 SLT_2_3	No, generate error No, return input as it is	238,328 2*238,328	931 1,862	1,862 3,724	Same tokenizers and other values as for Alpha-Numeric token element
		Yes				
Unicode Base64	SLT_1_3 SLT_2_3	No, generate error No, return input as it is	274,625 2*274,625	1,073 2,146	2,146 4,292	Same tokenizers and other values as for Alpha-Numeric token element
		Yes				
Unicode Gen2	SLT_1_3 SLT_X_1	No, generate error No, generate error No, return input as it is	4,096,000 359,994[*1]	16,384 1,440[*1]	32,768 2,880[*1]	
	SLT_1_3 SLT_X_1	Yes Yes	4,121,760 500,000[*2]	16,488 2,000[*2]	32,975 4,000[*2]	
Binary	SLT_1_3 SLT_2_3	NA	238,328 2*238,328	931 1,862	1,862 3,724	Same tokenizers and other values as for Alpha-Numeric token element
Email	SLT_1_3 SLT_2_3	No, generate error No, return input as it is	238,328 2*238,328	931 1,862	1,862 3,724	Same tokenizers and other values as for Alpha-Numeric token element
		Yes	242,234	946.22	1,892.45	

Token Type	*Tokenizer*	*Allow Short Tokens*	*Maximum Size of Token Tables (number of entries)*	*Size of Token Tables (kB)*	*Amount of Memory used in the Protector (kB)*	*Comments*
			2*242,234	1,892.45	3,784.90	

Note: The data elements created using the *SLT_2_6* tokenizer are not deployable to protectors lower than versions 7.1.

Important:

The *alphabet size* denotes the number of code points present in an alphabet.

Note:

*1 - The characteristic values of the *SLT_X_1* tokenizer are applicable if the alphabet size is 59,999.

Note:

*2 - The characteristic values of the *SLT_X_1* tokenizer are applicable if the alphabet size is 100,000.

The amount of memory used in the protector is twice the size of the token tables (kB) because an inverted SLT is stored in shared memory, in addition to the original SLT.

3.3.3 Left and Right Settings

This property indicates the number of characters from left and right that will remain in the clear and hence be excluded from tokenization. Not all token types will allow the end-user to specify these values. The *Left and Right* settings can be configured in the **Tokenize Options** on the ESA Web UI

When processing input data where both *Left and Right* settings and *Allow Short Data* settings are applied, the input is validated for the the *Left and Right* settings before the *Allow Short Data* settings are applied.

For more information about how left and right settings work together with short data settings, refer to the section *Calculating Token Length (Zero-Length Tokens)*.

3.3.4 Internal Initialization Vector (IV)

An Internal IV is used during the tokenization process to make it more difficult to detect patterns in multiple tokenized values and thereby provide additional security.

Internal IV is automatically applied to the input value when the token element’s left and/or right properties are non-zero, designating some characters to remain in the clear.

Data to tokenize can be logically divided into three components: left, middle, and right. If an IV is used, then the left and right components are concatenated to form the IV. This IV is then added to the middle component before the value is tokenized.

Table 3-11: Examples of Tokenization with Internal IV

Token Properties	*Input Value*	*Output Value*	*Comments*
Alpha Token	1Protegrity	1aOkCUXmhXC	Left=1 thus the first character in the input value is not tokenized

Token Properties	*Input Value*	*Output Value*	*Comments*
Left=1 Right=0	2Protegrity 3Protegrity	2DeKeldVpKj 3hASBMvvfuL	but used as internal IV. For each of three input values the value ‘Protegrity’ is tokenized, with internal IVs ‘1’, ‘2’ and ‘3’ respectively. Tokenized value is different for all three cases.
Alpha Token Left=2 Right=4	W2Protegrity2012 W2Protegrity2013 Q2Protegrity2013	W2NXgfOdLQEy2012 W2XdjFTIFQNC2013 Q2gWjpyMwvDJ2013	Left=2, Right=4 thus the first 2 and the last 4 characters in the input value are not tokenized but used as internal IV. For each of three input values the value ‘Protegrity’ is tokenized, with internal IVs ‘W22012’, ‘W22013’ and ‘Q22013’ respectively. Tokenized value is different for all three cases.
Alpha Token Left=0 Right=0	Protegrity	RlfZVOmhQD	Left and Right are undefined thus the internal IV is not used.

3.3.5 Minimum and Maximum Input Length

The minimum and maximum input lengths are the boundaries that are used in input validation.

In Protegrity tokenization only one token type (Decimal) allows defining the Minimum and Maximum length of the token element upon its creation. Some token types (Date and Datetime) have a fixed length. For the remainder, Minimum and Maximum length depends on token type, tokenizer, length preservation, and short token setting.

The following table illustrates length settings by token type.

Important: For Protegrity protectors on z/OS, the maximum length for all the tokens are limited to 256 characters except the following:

- Integer
- Date
- Datetime
- Decimal

Table 3-12: Minimum and Maximum Input Length by Token Types

Token Type	Tokenizer	Preserves Length	Allow Short Data	Minimum Length	Maximum Length
Numeric	SLT_1_3 SLT_2_3	Yes	Yes	1	4096
			No, return input as it is	3	
			No, generate error		

Token Type	Tokenizer	Preserves Length	Allow Short Data	Minimum Length	Maximum Length
		No	NA	1	3933
	SLT_1_6 SLT_2_6[*1]	Yes	Yes	1	4096
			No, return input as it is	6	
			No, generate error		
		No	NA	1	3933
Integer	SLT_1_3	Yes	NA	2	8
Credit Card	SLT_1_3 SLT_2_3	Yes	NA	3	4096
	SLT_1_6 SLT_2_6[*1]	Yes	NA	6	4096
Alpha	SLT_1_3 SLT_2_3	Yes	Yes	1	4096
			No, return input as it is	3	
			No, generate error		
		No	NA	1	4076[1]
Upper-case Alpha	SLT_1_3 SLT_2_3	Yes	Yes	1	4096
			No, return input as it is	3	
			No, generate error		
		No	NA	1	4049
Alpha-Numeric	SLT_1_3 SLT_2_3	Yes	Yes	1	4096
			No, return input as it is	3	
			No, generate error		
		No	NA	1	4080

Token Type	Tokenizer	Preserves Length	Allow Short Data	Minimum Length	Maximum Length
Upper Alpha-Numeric	SLT_1_3 SLT_2_3	Yes	Yes	1	4096
			No, return input as it is	3	
			No, generate error		
		No	NA	1	4064
Lower ASCII	SLT_1_3	Yes	Yes	1	4096
			No, return input as it is	3	
			No, generate error		
		No	NA	1	4086
Printable	SLT_1_3	Yes	Yes	1	4096
			No, return input as it is	3	
			No, generate error		
		No	NA	1	4091
Date YYYY-MM-DD Date DD/MM/YYYY Date MM/DD/YYYY	SLT_1_3 SLT_2_3 SLT_1_6 SLT_2_6*	Yes	NA	10	10
Datetime	SLT_DATETIME	Yes	NA	10	23
Decimal	SLT_6_DECIMAL	No	NA	1	36
Unicode	SLT_1_3 SLT_2_3	No	Yes	1 byte	4096 bytes
			No, return input as it is	3 bytes	
			No, generate error		
Unicode Base64	SLT_1_3	No	Yes*2	1 byte	4096 bytes
	SLT_2_3		No, return input as it is	3 byte	

Token Type	Tokenizer	Preserves Length	Allow Short Data	Minimum Length	Maximum Length
			No, generate error		
Unicode Gen2	SLT_1_3 SLT_X_1	Yes	Yes	1 Code Points	4096 Code Points
			No, return input as it is	3 Code Points	
			No, generate error		
Binary	SLT_1_3 SLT_2_3	No	NA	3	4095
Email	SLT_1_3 SLT_2_3	Yes	Yes	3	256
			No, return input as it is	5	
			No, generate error		
		No	NA	3	256

Note:

*2 - If the input value for the protect or unprotect operation is between 1 character to 3 characters and the Allow Short Data setting is set to Yes, then the protect or unprotect operation fails with an error message indicating the input data is too short.

Note:

- The minimum/maximum length means number of characters of a supported alphabet (thus, for alphanumeric tokens, an input value consisting of one printable character of a non-supported alphabet, will not be tokenized).
- The minimum and maximum length for an integer is between 2 bytes to 8 bytes.
- The minimum and maximum lengths supported by the Integer token type for the Impala protector in the Big Data Protector are 1 and 10 respectively. The Impala protector supports only 4-byte integers.
- The minimum/maximum length validation on input data is done on the characters to tokenize.
- Left and right clear characters are not counted as well as any characters outside of the alphabet for the selected token type.
- NULL values are accepted but not tokenized.
- Email token minimum length in the table above means the length of the entire email.
- Maximum lengths of all tokens are limited to 256 characters for Protegrity protectors on z/OS.
- The newly created data elements using the SLT_2_6 tokenizer from v7.1 MR1 onwards are deployable to protectors with versions 7.1 MR1 and higher.

3.3.5.1 Calculating Token Length (Zero-Length Tokens)

If the input value does not contain characters that can be tokenized with the selected token type, then the characters not supported by the token type will be treated as delimiters and left un-tokenized.

The number of characters to tokenize is calculated as described on the following image:

Figure 3-1: Number of characters to tokenize

If the input value does not contain characters to tokenize, then it is considered a **zero-length token**. The tokenization of an empty input value will not produce an error during the tokenization, and input value will be returned as output.

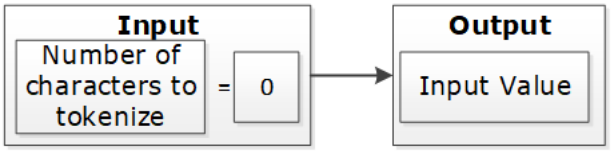

Figure 3-2: Input value returned as a result of tokenization with zero-length token

If the input value has at least one character and short data tokenization is enabled, then the source data can be tokenized. If short data tokenization is not enabled then as per the settings, either the source data can be returned as it is or an appropriate error appears as a result of tokenization.

For more information on short data tokenization, refer to the section *Short Data Tokenization*.

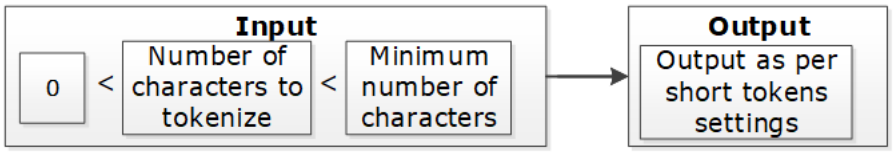

Figure 3-3: Output returned when the input is too short

If the input value contains more than the maximum number of characters to tokenize, then the value to tokenize is considered too long, and the appropriate error will appear as a result of tokenization.

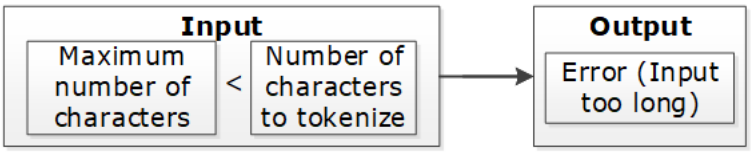

Figure 3-4: Error returned when the input is too long

If the input value contains enough characters to tokenize (between minimum and maximum settings), then the tokenization is successful.

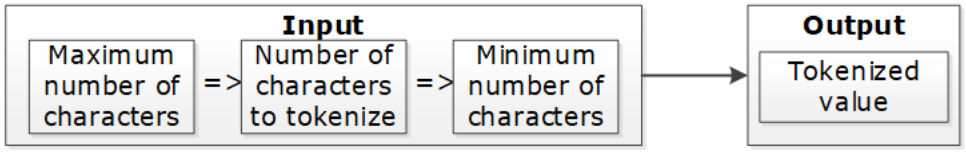

Figure 3-5: Tokenized value returned when the input is enough for tokenization

Table 3-13: Token Length Examples

Token Properties	Input Value	Output Value	Comments
Numeric Token	ab1cd	ab6cd	Non-numeric values are considered as delimiters. Input

Token Properties	Input Value	Output Value	Comments
Left/Right undefined Allow Short Tokens=Yes			is tokenized as short data is enabled and minimum length is 1 character.
Numeric Token Left/Right undefined Allow Short Tokens= No, generate an error	ab1cd	Error. Input too short.	Non-numeric values are considered as delimiters. Input is short since short data is not enabled and the minimum number of characters to tokenize for this token type is 3 characters.
Numeric Token Left/Right undefined Allow Short Tokens= No, return input as it is	12	12	Input is returned as is as per the settings for short data.
Numeric Token Left=2 Right=2	48ghdg83	48ghdg83	The input value is left unchanged by the tokenization since it is an empty value for tokenization (left and right settings remove all numeric characters).
Numeric Token Left=2 Right=2	4568	4568	The input value is left unchanged by the tokenization since it is an empty value for tokenization.
Numeric Token Left/Right undefined	ab123cd	ab857cd	Input value has enough characters for tokenization, only supported by numeric token type values are tokenized.
Alpha Numeric Token Left=5, Right=0 Allow Short Tokens= yes	345465	34546c	Input is evaluated first for left and right settings. Since left settings are set to 5, the first five digits are excluded and the sixth digit can be tokenized. As the *Allow Short Tokens* is set as **yes**, the sixth digit is tokenized.
Alpha Numeric Token Left=5, Right=0 Allow Short Tokens= no, generate error	345465	error	Input is evaluated first for left and right settings. Since left settings are set to 5, the first five digits are excluded and the sixth digit can be tokenized. As the *Allow Short Tokens* is set as **no, generate error** and the length of data to be tokenized is less than 3, an *Input too short* error is generated.
Alpha Numeric Token Left=5, Right=0 Allow Short Tokens= No, return input as it is	345465	345465	Input is evaluated first for left and right settings. Since left settings are set to 5, the first five digits are excluded and the sixth digit can be tokenized. As the *Allow Short Tokens* is set as **No, return input as it is** and the length of data to be tokenized is less than 3, the data is passed as is.

Token Properties	Input Value	Output Value	Comments
Alpha Numeric Token Left=5, Right=0 Allow Short Tokens= yes	34546	34546	Input is evaluated first for left and right settings. Since left settings are set to 5 and the input is five digits, no data exists to be tokenized. As no data exists, it is considered as a zero length token and the input is passed as is.
Alpha Numeric Token Left=5, Right=0 Allow Short Tokens= no, generate error	34546	34546	
Alpha Numeric Token Left=5, Right=0 Allow Short Tokens= No, return input as it is	34546	34546	
Alpha Numeric Token Left=5, Right=0 Allow Short Tokens= yes	3454	error	Input is evaluated first for left and right settings. Since left settings are set to 5 and the input is four digits, the left and right settings condition is not met. This results in an *Input too short* error.
Alpha Numeric Token Left=5, Right=0 Allow Short Tokens= no, generate error	3454	error	
Alpha Numeric Token Left=5, Right=0 Allow Short Tokens= No, return input as it is	3454	error	

3.3.6 Length Preserving

With **Preserve Length** flag enabled, the length of the input data and protected token value is the same.

For data elements with **Preserve Length** flag available, you have an option to generate token values that are of the same length as the input data.

> **Note:** The Unicode Gen2 token element is Code Point length preserving based on the alphabet selected during data element creation.

As an extension to this flag, the **Allow Short Data** flag provides multiple options to manage short input data handling. If the **Preserve Length** property is not selected, then short input can be extended up to the minimum length.

For more information about short data tokenization, refer to the section *Short Data Tokenization*.

The system always enforces a maximum length, whether length preservation is on or off.

If **Preserve Length** is not selected, then tokenized data may be longer than the input value up to +5%, or at least +1 symbol on a very small initial value (1-2 symbols).

If **Preserve Length** is not selected, then:

- Out-of-alphabet characters could cause an error because they are not tokenized and they need to be put back into the correct position when the data is detokenized. When length preservation is not selected, position information is lost.

- For database protection, column length of the resulting protected table should be bigger than length of the column to tokenize in the initial table. This will allow inserting tokenized data during protection when tokenized data is longer than the input data.

3.3.7 Short Data Tokenization

The use of short data input supports tokenization, if the source data length to be tokenized is less than the limit for tokenizable characters or bytes. When using tokenizers, such as, SLT_1_3, SLT_2_3, and SLT_X_1, the limit for tokenizable characters or bytes is three. When using tokenizers, such as, SLT_1_6 and SLT_2_6, the limit for tokenizable characters or bytes is six.

The possible flag values for short data tokenization are described in the following table.

Table 3-14: Short tokens flag values

Short Token flag value	Action
No, generate error	Do not tokenize the short input but generate an error code and an audit log.
Yes	Tokenize the data if the input is short.
No, return input as it is	Do not tokenize the short input but return the input as it is.

The following tokens support short data tokenization:

- Numeric (0-9)
- Alpha (a-z, A-Z)
- Upper-case Alpha (A-Z)
- Alpha-Numeric (0-9, a-z, A-Z)
- Upper-case Alpha-Numeric (0-9, A-Z)
- Printable
- Lower ASCII
- Email
- Unicode
- Unicode Base64
- Unicode Gen2

Important: Since there is no chaining for short input data, tokenization can be at risk. User can easily guess the lookup table and the original data by tokenizing some input data.

It is recommended that careful consideration is given to employ short data tokenization. If possible, short data input must be avoided.

For more information about the maximum length setting for non-length-preserving token elements, refer to table *Minimum and Maximum Input Length by Token Types*.

3.3.8 Case-Preserving and Position-Preserving Tokenization

If you are working with the *Alpha-Numeric (0-9, a-z, A-Z)* token type and the *SLT_2_3* tokenizer, then you can specify additional tokenization options for case preservation and position preservation. This section explains these additional tokenization options.

Warning:

Case-Preserving and Position-Preserving tokenization was designed to support very specific business requirements. There is a trade-off between those requirements and the cryptographic strength of the tokens. When preserving the case and position of Alpha-Numeric characters, some information may be leaked through the tokenized value. In addition, depending on the length of the Alpha and Numeric

substrings, tokens may suffer the same weaknesses as Short Tokens, as described in the section *Short Data Tokenization*. Generally, this method should not be used for most use cases. Before using this method, contact Protegrity to ensure that the risks are fully understood.

3.3.8.1 Case-Preserving Tokenization

When working with data that is received from multiple sources, the data can contain different casing properties. The data processing stage makes the casing consistent prior to distributing the data to additional systems.

If tokenization is performed prior to the data processing stage, then it results in tokens that differ in its casing properties as per the non-processed data.

To ensure that the data casing properties are preserved when tokenizing the non-processed data, an additional tokenization option is provided to preserve the case for the *Alpha-Numeric (0-9, a-z, A-Z)* token type. The casing of the alphabets in the tokenized value matches the casing of the alphabets in the input value.

Note:

You can specify the case-preserving tokenization option when using the *SLT_2_3* tokenizer and *Alpha-Numeric (0-9, a-z, A-Z)* token type only.

If you select the *Preserve Case* property on the ESA Web UI, then the *Preserve Position* property is also selected, by default. Hence, the position of the alphabets and numbers is preserved along with the casing of the alphabets in the output tokenized value.

If you are selecting the *Preserve Case* or *Preserve Position* property on the ESA Web UI, then the following additional properties are set:

- The *Preserve Length* property is enabled and *Allow Short Data* property is set to *Yes*, by default. These two properties are not modifiable.
- The retention of characters or digits from the left and the right are disabled, by default. The *From Left* and *From Right* properties are both set to zero.

For more information about specifying the case-preserving tokenization option for the *Alpha-Numeric (0-9, a-z, A-Z)* token type, refer to the section *Creating Case-Preserving and Position-Preserving Data Element* in the *Policy Management Guide 9.1.0.0*.

The following table provides some examples for the case-preserving tokenization option.

Table 3-15: Case-Preserving Tokenization Examples

Input Value	Tokenized Value Using the Case-Preserving Tokenization
Dan123	Abc567
DAn123	ABc567
daN123	abC567

3.3.8.2 Position-Preserving Tokenization

The position-preserving tokenization preserves the position of the alphabets and numbers when tokenizing the alpha-numeric values. The alphabetic and numeric positions in the tokenized value matches the alphabetic and numeric positions in the input value.

Note:

You can specify the position-preserving tokenization option when using the *SLT_2_3* tokenizer and *Alpha-Numeric (0-9, a-z, A-Z)* token type only.

If you are selecting the *Preserve Case* or *Preserve Position* property, then the following additional properties are set:

- The *Preserve Length* property is enabled and *Allow Short Data* property is set to *Yes*, by default. These two properties are not modifiable.
- The retention of characters or digits from the left and the right are disabled, by default. The *From Left* and *From Right* properties are both set to zero.

For more information about specifying the position-preserving tokenization option for the *Alpha-Numeric (0-9, a-z, A-Z)* token type, refer to the section *Creating Case-Preserving and Position-Preserving Data Element* in the *Policy Management Guide 9.1.0.0*.

The following table provides some examples for the position-preserving tokenization option.

Table 3-16: Position-Preserving Tokenization Examples

Input	Tokenized Value Using the Position-Preserving Tokenization
Dan123	pXz789
DAn123	Abp708
daN123	Axz642

3.3.9 External Initialization Vector (IV)

External IV feature provides an additional level of security, by adding the possibility to have different tokenized results across protectors for the same input data and token element, depending on the External IV set on each protector.

3.3.9.1 Tokenization model with External IV

The External IV value is set as a new parameter when calling Application Protector from the client application.

To enforce additional security, the External IV value is not used “as is” when applied to a token value. Instead it is being tokenized using a specific internal token element.

The following example explains how the tokenization is performed with the External IV defined. As mentioned before, the main characteristic of the External IV feature is obtaining different outputs for the same input. To have different outputs, you need to specify different IVs.

Note: External IV is used, prior to protection, as input to modify the data to protect. The external IV is ignored when using encryption.

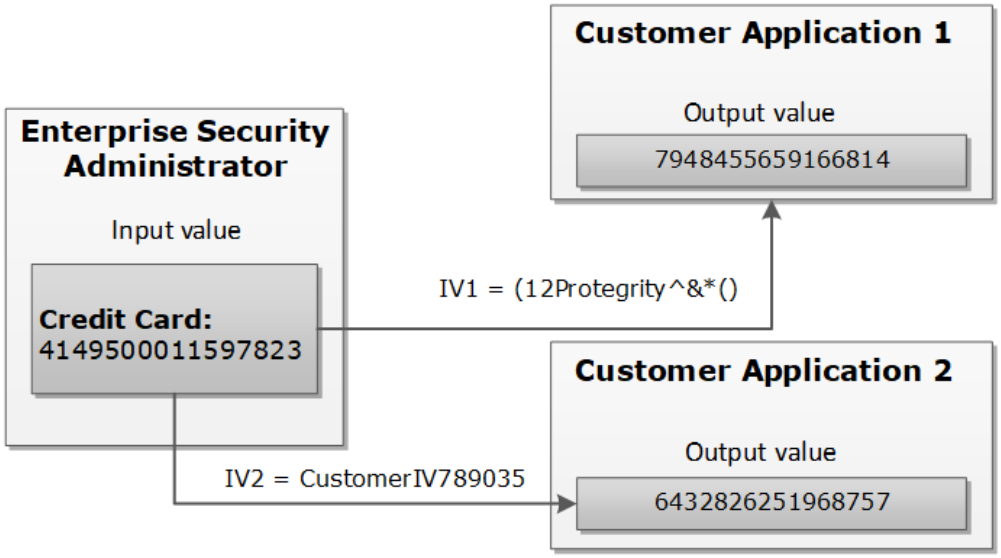

Figure 3-6: External IV in the Credit Card tokenization process

3.3.9.2 External IV Tokenization Properties

The External IV is supported by all token types, except Date, Datetime, and Decimal tokens.

The tokenization with the External IV is done only if the IV is specified during the protect operation through the end user API. When performing unprotect and re-protect operations, the same IV value used for protection must be identified.

If External IV is not provided in either protect or unprotect function call, then the input is tokenized as-is without any IV.

The External IV value has the following properties:

- Supports ASCII and Unicode characters
- Minimum 1 byte for the input, but 2 bytes is the recommended length to create unique Token elements using External IV
- Maximum 256 bytes for the input
- Empty and NULL strings are not supported as External IV value (these strings will be ignored and tokenization will be performed as if External IV was not used).

> **Note:** External IV parameter should not be used for Token elements created in 6.0 and earlier releases. On attempt to use External IV with such Token elements, an error will be displayed.

Here is an example of the tokenized input value with the External IV for a Numeric token:

Table 3-17: Example-External IV for a Numeric token

Input Value	External IV	Output Value	Comments
1234567890	None	5108318538	External IV is not applied.
1234567890	1234	0442985096	Output values differ because different external IVs were applied.
	12	1197578213	
	abc	9423146024	

3.3.10 Truncating White Spaces

If you have fixed length fields or columns and the input data is shorter than the length of the field, the data may be appended with some trailing or/and leading white spaces. In such situations, the trailing and/or leading white spaces will be considered during Tokenization and will impact the tokenization results.

For instance, consider a scenario where the name Hultgren Caylor is stored in a Hive Char(30) column.

As the length of the data is less than 30 characters, trailing white spaces are appended to it. In this case, assume that we need to protect this column with a data element that preserves the first and last character (L=1, R-1). Now with this setting, the expectation is to preserve character H at the start and the character r at the end, in the protected value output. However, the actual data has trailing white spaces, which means, the output will contain the character H at the start and character " " (whitespace) at the end, which is unnecessary. The final protected output would generate a different token due to the unnecessary trailing white space.

It is therefore recommended to truncate any unnecessary trailing or/and leading white spaces, before sending the data to the respective Protect, Unprotect, or Reprotect UDFs. This ensures that only the actual data is considered in the tokenization process, and any unnecessary trailing or/and leading white spaces are not considered.

In addition, it is important to follow a consistent approach for truncating the white spaces across all operations, such as, Protect, Unprotect, Reprotect. For instance, if we have truncated unnecessary trailing white spaces from the input before the Protect operation, then the same logic of truncating white spaces from the input, during Unprotect and Reprotect operations needs to be followed.

3.4 Tokenization Types

This section describes the tokenization type properties for different protectors. It also provides some examples for tokenized values for different token types.

3.4.1 Numeric (0-9)

The Numeric token type tokenizes digits from 0 to 9.

Table 3-18: Numeric Tokenization Type properties for different Protectors

Tokenization Type Properties	**Settings**			
Name	Numeric			
Token type and Format	Digits 0 through 9			
Tokenizer	**Length Preservation**	**Allow Short Data**	**Minimum Length**	**Maximum Length**
SLT_1_3 SLT_2_3	Yes	Yes	1	4096
		No, return input as it is	3	
		No, generate error		
	No	NA	1	3933

Tokenization Type Properties	Settings			
SLT_1_6 SLT_2_6[*4]	Yes	Yes	1	4096
		No, return input as it is	6	
		No, generate error		
	No	NA	1	3933
Possibility to set Minimum/ maximum length	No			
Left/Right settings	Yes			
Internal IV	Yes, if Left/Right settings are non-zero			
External IV	Yes			

Supported input data types (by Application Protectors) *1	**AP Python**[*3]	**AP Java** [*3]	**AP C**[*3]	**AP NodeJS**[*3]	**AP .Net**[*3]	**AP Go**[*3]
	STRING BYTES	STRING CHAR[] BYTE[]	BYTE[]	STRING BYTE[]	STRING BYTE[]	STRING []BYTE

Supported input data types (by DB Protectors)	**MSSQL Server**	**Oracle**	**DB/2**
	VARCHAR CHAR	VARCHAR 2 CHAR	VARCHAR CHAR

Supported input data types (by MPP DB Protectors)	**Teradata**	**GPDB**	**IBM Netezza**
	VARCHAR LATIN CHAR LATIN	VARCHAR	VARCHAR

Supported input data types (for Big Data Protectors) [*1]	**MapReduce** [*2]	**Hive**	**Pig**	**HBase** [*2]	**Impala**	**Spark** [*2]	**Spark SQL**	**Presto**
	BYTE[]	CHAR[*5] STRING	CHARARRAY	BYTE[]	STRING	BYTE[] STRING	STRING	VARCHAR

Tokenization Type Properties	Settings
Return of Protected value	Yes
Supported in Protegrity releases	6.6.x – 9.x.x.x
Token specific properties	None

Note:

*1– If the input and output types of the API are BYTE[], then the customer application should convert the input to and output from the byte array, before calling the API.

*2 – The Protegrity MapReduce protector, HBase coprocessor, and Spark protector only support bytes converted from the string data type. If any other data type is directly converted to bytes and passed as input to the MapReduce or Spark API that supports byte as input and provides byte as output, then data corruption might occur. If any other data type is directly converted to bytes and inserted in an HBase table, which is configured with the Protegrity HBase coprocessor, then data corruption might occur.

*3 – The Protegrity Application 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 Application Protectors APIs that support byte as input and provide byte as output, then data corruption might occur.

*4 – The newly created data elements using the SLT_2_6 tokenizer from v7.1 MR1 onwards are deployable to protectors with versions 7.1 MR1 and higher.

*5 – If you are using the Char tokenization UDFs in Hive, then ensure that the data elements have length preservation selected. In Char tokenization UDFs, using data elements without length preservation selected, is not supported.

The following table shows examples of the way in which a value will be tokenized with the Numeric token.

Table 3-19: Examples - Numeric tokenization values

Input Value	Tokenized Value	Comments
123	977	Numeric, SLT_1_3, Left=0, Right=0, Length Preservation=Yes The value has minimum length for SLT_1_3 tokenizer.
1	555241	Numeric, SLT_1_6, Left=0, Right=0, Length Preservation=No The value is padded up to 6 characters which is minimum length for SLT_1_6 tokenizer.
-7634.119	-4306.861	Numeric, SLT_1_3, Left=0, Right=0, Length Preservation=Yes Decimal point and sign are treated as delimiters and not tokenized.
12+38=50	98+24=62	Numeric, SLT_2_6, Left=0, Right=0, Length Preservation=Yes Arithmetic signs are treated as delimiters and not tokenized.
704-BBJ	134-BBJ	Numeric, SLT_1_3, Left=0, Right=0, Length Preservation=Yes Alpha characters are treated as delimiters and not tokenized.
704-BBJ	Error. Input too short.	Numeric, SLT_2_6, Left=0, Right=0, Length Preservation=Yes, Allow Short Data=No, generate error Input value has only three numeric characters to tokenize, which is short for SLT_2_6

Input Value	Tokenized Value	Comments
		tokenizer when Length Preservation=Yes and Allow Short Data=No, generate error.
704-BBJ	704-BBJ	Numeric, SLT_2_6, Left=0, Right=0, Length Preservation=Yes, Allow Short Data=No, return input as it is The alpha characters and dash are treated as delimiters. The remaining input is less than six characters and is not tokenized.
704356	134432	Numeric, SLT_2_6, Left=0, Right=0, Length Preservation=Yes, Allow Short Data=No, return input as it is The input value equals to six characters and is tokenized.
704-BBJ	134-BBJ	Numeric, SLT_2_6, Left=0, Right=0, Length Preservation=Yes, Allow Short Data=Yes Input value has three numeric characters to tokenize, which meets minimum length requirement for SLT_2_6 tokenizer when Length Preservation=Yes and Allow Short Data=Yes.
704	134	Numeric, SLT_1_3, Left=0, Right=0, Length Preservation=Yes, Allow Short Data=No, return input as it is If the input value has less than three characters to tokenize, then it is returned as is else it is tokenized.
704-BBJ	669-BBJ642	Numeric, SLT_1_6, Left=0, Right=0, Length Preservation=No Input value is padded up to 6 characters because Length Preservation=No. Alpha characters are treated as delimiters and not tokenized.
704-BBJ	764-6BBJ	Numeric, SLT_2_3, Left=1, Right=3, Length Preservation=No 1 character from left and 3 from right are left in clear. Two numeric characters left for tokenization ‘04’ were padded and tokenized as ‘646’.

3.4.2 Integer (0-9)

The Integer token type tokenizes 2, 4, or 8 byte size integers.

Table 3-20: Integer Tokenization Type properties for different Protectors

Tokenization Type Properties	Settings
Name	Integer

Tokenization Type Properties	Settings											
Token type and Format	2, 4, or 8 byte size integers											
Tokenizer	Length Preservation											
SLT_1_3	Yes											
Possibility to set Minimum/ maximum length	No											
Left/Right settings	No											
Internal IV	No											
External IV	Yes											
Supported input data types (by Application Protector)[*1]	**AP C**[*3]					**AP Java**[*3]	**AP Python**[*3]	**AP .Net**[*3]	**AP Go**[*3]	**AP NodeJS**[*3]		
	BYTE[]					SHORT: 2bytes INT: 4 bytes LONG: 8 bytes	INT: 4 bytes and 8 bytes	STRING BYTE[]	SHORT: 2bytes INT: 4 bytes LONG: 8 bytes	STRING BYTE[]		
Supported input data types (by DB Protectors)	**MSSQL Server**						**Oracle**			**DB/2**		
	INTEGER						INTEGER			INTEGER SMALLINT BIGINT		
Supported input data types (by MPP DB Protectors)	**Teradata**						**Greenplum**			**IBM Netezza**		
	INTEGER BIGINT						INTEGER			INTEGER		
Supported input Presto data types (for Big Data Protectors)[*1]	**MapReduce**[*2]	**Hive**	**Pig**	**HBase**[*2]	**Impala**	**Spark**[*2]	**Spark SQL**			**Presto**		
	INT: 4 bytes LONG: 8 bytes	INT: 4 bytes BIGINT: 8 bytes	INT: 4 bytes	BYTE[]	INT: 4 bytes	SHORT: 2 bytes INT: 4 bytes LONG: 8 bytes	SHORT: 2 bytes INT: 4 bytes LONG: 8 bytes			SMALL INT: 2 bytes INT: 4 bytes BIG INT: 8 bytes		
Return of Protected value	Yes											
Supported in Protegrity releases	6.6.x – 9.x.x.x											

Tokenization Type Properties	Settings
Token specific properties	Size 2, 4, or 8 bytes

Note:

*1 – If the input and output types of the API are BYTE[], then the customer application should convert the input to and output from the byte array, before calling the API.

*2 – The Protegrity MapReduce protector, HBase coprocessor, and Spark protector only support bytes converted from the string data type. If any other data type is directly converted to bytes and passed as input to the MapReduce or Spark API that supports byte as input and provides byte as output, then data corruption might occur. If any other data type is directly converted to bytes and inserted in an HBase table, which is configured with the Protegrity HBase coprocessor, then data corruption might occur.

*3 – The Protegrity Application 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 Application Protectors APIs that support byte as input and provide byte as output, then data corruption might occur.

*4 – The Integer token type with 2 bytes as input is not supported for AP Python. A protect, unprotect or a reprotect operation performed by using the Integer token type with 2 bytes as a data element for input data, will not be successful. For a Bulk call using the protect, unprotect, and reprotect APIs, the error code, `44`, appears. For a single call using the protect, unprotect, and reprotect APIs, an exception will be thrown and the error message, "`44, Content of input data is not valid`" appears.

*5 - If the user passes 4-byte integer (values ranging from -2,147,483,648 to +2,147,483,647) as data and uses the 8-byte Integer token type data element as input for the protect, unprotect, or reprotect APIs, then the data protection operation will not be successful. For a Bulk call using the protect, unprotect, and reprotect APIs, the error code, `44`, appears. For a single call using the protect, unprotect, and reprotect APIs, an exception will be thrown and the error message, "`44, Content of input data is not valid`" appears.

Note:

The z/OS Database protectors, FIELDPROC and UDFs, do not support BIGINT.

The following table shows examples of the way in which a value will be tokenized with the Integer token.

Table 3-21: Examples - Integer tokenization values

Input Value	Tokenized Value	Comments
12	31345	Integer, SLT_1_3, Left=0, Right=0, Length Preservation=Yes
3	1465	For 2 bytes, the values can range from -32768 to 32767.
3	782939681	For 4 bytes, the values can range from -2147483648 to 2147483647.
3	7268379031142372719	For 8 bytes, the value range can range from -9223372036854775808 to 9223372036854775807.

Note:

The *pty.ins_integer* UDF in the Oracle, Teradata, and Impala Protectors, supports input data length of 4 bytes only. For 2 bytes, the following error is returned: `Invalid input size`.

3.4.3 Credit Card

Our Credit Card token type helps maintain transparency, and also provides a way to clearly distinguish a token from the real value, which is a recommendation of the PCI DSS.

The Credit Card token type supports only numeric input (no separators are allowed as input).

Table 3-22: Credit Card Tokenization Type properties for different Protectors

Tokenization Type Properties	**Settings**					
Name	Credit Card					
Token type and Format	Digits 0 through 9 (no separators are allowed as input)					
Tokenizer	**Length Preservation**	**Minimum Length**	**Maximum Length**			
SLT_1_3 SLT_2_3	Yes	3	4096			
SLT_1_6 SLT_2_6*4	Yes	6	4096			
Possibility to set Minimum/ maximum length	No					
Left/Right settings	Yes					
Internal IV	Yes, if Left/Right settings are non-zero					
External IV	Yes					
Supported input data types (by Application Protectors) *1	**AP Python** *3	**AP Java** *3	**AP C**3	**AP NodeJS**3	**AP .Net**3	**AP Go*3**
	STRING BYTES	STRING CHAR[] BYTE[]	BYTE[]	STRING BYTE[]	STRING BYTE[]	STRING []BYTE
Supported input data types (by DB Protectors)	**MSSQL Server**	**Oracle**	**DB/2**			
	VARCHAR CHAR	VARCHAR2 CHAR	VARCHAR CHAR			
Supported input data types (by MPP DB Protectors)	**Teradata**	**GPDB**	**IBM Netezza**			
	VARCHAR LATIN CHAR LATIN	VARCHAR	VARCHAR			

Tokenization Type Properties	Settings							
Supported input data types (for Big Data Protectors) *1	**MapReduce** *2	**Hive**	**Pig**	**HBase** *2	**Impala**	**Spark** *2	**Spark SQL**	**Presto**
	BYTE[]	STRING	CHARARRAY	BYTE[]	STRING	BYTE[] STRING	STRING	VARCHAR
Return of Protected value	Yes							
Supported in Protegrity releases	6.6.x – 9.x.x.x							
Token specific properties	Invalid LUHN digit Invalid card type Alphabetic indicator							

Note:

*1 – If the input and output types of the API are BYTE[], then the customer application should convert the input to and output from the byte array, before calling the API.

*2 – The Protegrity MapReduce protector, HBase coprocessor, and Spark protector only support bytes converted from the string data type. If any other data type is directly converted to bytes and passed as input to the MapReduce or Spark API that supports byte as input and provides byte as output, then data corruption might occur. If any other data type is directly converted to bytes and inserted in an HBase table, which is configured with the Protegrity HBase coprocessor, then data corruption might occur.

*3 – The Protegrity Application 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 Application Protectors APIs that support byte as input and provide byte as output, then data corruption might occur.

*4 – The newly created data elements using the SLT_2_6 tokenizer from v7.1 Maintenance Release 1 (MR1) onwards are deployable to protectors with versions 7.1 MR1 and higher.

The credit card number real value is distinguished from the tokenized value based on the token value validation properties.

Table 3-23: Specific Properties of the Credit Card Token Type

Credit Card token value validation properties	Left	Right	Comments	Validation properties compatibility
Invalid Luhn Checksum (On/Off)	yes	yes	Right characters to be left in the clear can be specified (usually, 4 characters). For more information about the Invalid Luhn Checksum property, refer to section *Invalid Luhn Checksum*.	Can be used together

Credit Card token value validation properties	Left	Right	Comments	Validation properties compatibility
Invalid Card Type (On/Off)	0	yes	Left cannot be specified, it is zero by default. For more information about the Invalid Card Type property, refer to section *Invalid Card Type*.	
Alphabetic Indicator (On/Off)	yes	yes	The indicator will be in the token, which means that left and right can be specified. For more information about the Alphabetic Indicator property, refer to section *Alphabetic Indicator*.	Can be used only separately from the other token validation properties

Note: You can create a Credit Card token element and select no validation property for it. In that case, the Credit Card token element will be treated the same as a Numeric token with the exception that additional checks will be made on the input explained in the Credit Card token general properties column in the table above.

Note:

If you are enabling the Credit Card token properties, such as, Invalid LUHN checksum and Invalid Card Type, with the SLT Tokenizers, then refer to section *Credit Card Properties with SLT Tokenizers*.

3.4.3.1 Invalid Luhn Checksum

If you enable Invalid Luhn Checksum token validation, then you must use valid credit cards otherwise tokenization will be denied for an invalid credit card number.

A valid credit card has a valid Luhn checksum. Upon tokenization the tokenized value will have an invalid Luhn checksum. Here is an example of the tokenized credit card with the invalid Luhn digit.

Table 3-24: Credit Card Number with Luhn Checksum Examples

Credit Card Number	Tokenized Value	Comments
4067604564321453	Token is not generated due to invalid input value. Error is returned.	The input value contains invalid Luhn checksum. The value cannot be tokenized with Luhn enabled.
4067604564321454	2009071778438613	The Luhn in the input value is correct, the value is tokenized. Tokenized value has invalid Luhn checksum.

3.4.3.2 Invalid Card Type

When Invalid Card Type is enabled, token values will not begin with the digits that real credit card numbers begin with.

The first digit in a real credit card number is the Major Industry Identifier. Thus, digits **3,4,5,6** and **0** can be the first digits of the real credit card number which will be substituted during tokenization (refer to the following table).

Table 3-25: Real Credit Card Values with Tokenized Values

Real credit card value	3	4	5	6	0
Tokenized value	2	7	8	9	1

Here is an example of the tokenized credit card with the invalid credit card type:

Table 3-26: Credit Card Number with Tokenized Values - Examples

Credit Card Number	Tokenized Value	Comments
4067604564321454	7335610268467066	The credit card type is valid, the tokenization is successful.
2067604564321454	Token is not generated due to invalid input value. Error is returned.	The credit card type is invalid since the first digit of the value (‘2’) does not belong to a real credit card. The value cannot be tokenized.

3.4.3.3 Alphabetic Indicator

If you enable Alphabetic Indicator validation, then the resulting token value will have one alphabetic character.

You will need to choose the position of the alphabetic character before tokenizing a credit card number otherwise the resulting token will have no alphabetic indicator.

The alphabetic indicator will substitute the tokenized value according to the following rule:

Table 3-27: Alphabet Indicator with Tokenized Digits

Tokenized digit	0	1	2	3	4	5	6	7	8	9
Alphabetic indicator	A	B	C	D	E	F	G	H	I	J

In the following table the Visa Card Number “4067604564321454” has been tokenized with a value of “7594107411315001” and the tokenized value in a selected position is substituted with an alphabetic character.

Table 3-28: Credit Card Tokenization Examples

Credit Card Number (Input Value)	Position	Tokenized Value	Comments
4067604564321454	-	7594107411315001	No substitution since the position is undefined.
4067604564321454	14	7594107411315**A**01	Digit “0” is substituted with character “A” at position 14.

3.4.3.4 Credit Card Properties with SLT Tokenizers

This section helps you to understand the minimum data length required for the tokenization, when using the Credit Card token properties in combination with the SLT Tokenizers.

If you are enabling the Credit Card token properties, such as, Invalid LUHN checksum and Invalid Card Type, for tokenization of the input data, then you must also select the appropriate SLT Tokenizer to ensure the minimum data length is available to successfully perform the tokenization.

The following table represents the minimum data length required for tokenization as per the usage of Credit Card token properties with the SLT Tokenizers.

Table 3-29: Minimum Data Length - Credit Card Token Properties with SLT Tokenizers

Enabled Credit Card Token Property	Minimum Data Length (in digits) Required for Tokenization	
	SLT_1_3/SLT_2_3	SLT_1_6/SLT_2_6
Invalid LUHN Checksum	4	7
Invalid Card Type	4	7
Invalid LUHN Checksum and Invalid Card Type	5	8

3.4.4 Alpha (A-Z)

The Alpha token type tokenizes both uppercase and lowercase letters.

Table 3-30: Alpha Tokenization Type properties for different Protectors

Tokenization Type Properties	Settings			
Name	Alpha			
Token type and Format	Lowercase letters a through z Uppercase letters A through Z			
Tokenizer	**Length Preservation**	**Allow Short Data**	**Minimum Length**	**Maximum Length**
SLT_1_3 SLT_2_3	Yes	Yes	1	4096
		No, return input as it is	3	
		No, generate error		
	No	NA	1	4076 *3
Possibility to set Minimum/ maximum length	No			
Left/Right settings	Yes			
Internal IV	Yes, if Left/Right settings are non-zero			
External IV	Yes			

Tokenization Type Properties	Settings					
Supported input data types (by Application Protectors) *1	**AP Python** *4	**AP Java***4	**AP .Net***4	**AP NodeJS***4	**AP C***4	**AP Go*4**
	BYTES STRING	BYTE[] CHAR[] STRING	STRING BYTE[]	STRING BYTE[]	BYTE[]	[]BYTE STRING
Supported input data types (by DB Protectors)	**MSSQL Server**	**Oracle**			**DB/2**	
	VARCHAR CHAR	VARCHAR2 CHAR			VARCHAR CHAR	
Supported input data types (by MPP DB Protectors)	**Teradata**	**GPDB**			**IBM Netezza**	
	VARCHAR LATIN CHAR LATIN	VARCHAR			VARCHAR	

Supported input data types (for Big Data Protectors) *1	**MapReduce** *2	**Hive**	**Pig**	**HBase** *2	**Impala**	**Spark** *2	**Spark SQL**	**Presto**
	BYTE[]	CHAR *5 STRING	CHARARRAY	BYTE[]	STRING	BYTE[] STRING	STRING	VARCHAR

Return of Protected value	Yes
Supported in Protegrity releases	6.6.x – 9.x.x.x
Token specific properties	None

Note:

*1– If the input and output types of the API are BYTE[], then the customer application should convert the input to and output from the byte array, before calling the API.

*2– The Protegrity MapReduce protector, HBase coprocessor, and Spark protector only support bytes converted from the string data type. If any other data type is directly converted to bytes and passed as input to the MapReduce or Spark API that supports byte as input and provides byte as output, then data corruption might occur. If any other data type is directly converted to bytes and inserted in an HBase table, which is configured with the Protegrity HBase coprocessor, then data corruption might occur.

*3 – The Alpha token element for Application Protector with tokenizer SLT_1_3 and SLT_2_3 that has no length preservation has the maximum length of the protected data as 4080 bytes and 4082 bytes respectively.

*4 – The Protegrity Application 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 Application Protector APIs that support byte as input and provide byte as output, then data corruption might occur.

*5 – If you are using the Char tokenization UDFs in Hive, then ensure that the data elements have length preservation selected. In Char tokenization UDFs, using data elements without length preservation selected, is not supported.

The following table shows examples of the way in which a value will be tokenized with the Alpha token.

Table 3-31: Examples - Numeric tokenization values

Input Value	Tokenized Value	Comments
abc	nvr	Alpha, SLT_1_3, Left=0, Right=0, Length Preservation=Yes The value has minimum length for SLT_1_3 tokenizer.
MA	TGi	Alpha, SLT_2_3, Left=0, Right=0, Length Preservation=No The value is padded up to 3 characters which is minimum length for SLT_2_3 tokenizer.
MA	Error. Input too short.	Alpha, SLT_1_3, Left=0, Right=0, Length Preservation=Yes, Allow Short Data=No, generate error Input value has only two alpha characters to tokenize, which is short for SLT_1_3 tokenizer when Length Preservation=Yes and Allow Short Data=No, generate error.
MA MAC	MA TGH	Alpha, SLT_1_3, Left=0, Right=0, Length Preservation=Yes, Allow Short Data=No, return input as it is If the input value has less than three characters to tokenize, then it is returned as is else it is tokenized.
MA	TG	Alpha, SLT_1_3, Left=0, Right=0, Length Preservation=Yes, Allow Short Data=Yes Input value has only two alpha characters, which meets minimum length requirement for SLT_1_3 tokenizer when Length Preservation=Yes and Allow Short Data=Yes.
131 Summer Street, Bridgewater	131 VDYgAK q vMDUn, zAEXmwqWYNQG	Alpha, SLT_2_3, Left=0, Right=0, Length Preservation=No Numeric characters, spaces and comma are treated as delimiters and not tokenized. Output value is longer than initial value.
Albert Einstein	SldGzm OOCTzSFo	Alpha, SLT_1_3, Left=0, Right=0, Length Preservation=Yes

Input Value	Tokenized Value	Comments
		Space is treated as delimiters and not tokenized. Output value is the same length as initial value.
Albert Einstein	AjAkqD vvBFYLdo	Alpha, SLT_1_3, Left=1, Right=0, Length Preservation=Yes 1 character from left remains in the clear.

3.4.5 Upper-case Alpha (A-Z)

The Upper-case Alpha token type tokenizes all alphabetic symbols as uppercase. After de-tokenization, all alphabetic symbols are returned as uppercase. This means that initial and detokenized values would not match if the input contains lowercase letters.

> **Note:** In z/OS, the Upper-case Alpha token type considers lowercase characters as delimiter. It is recommended not to use Upper-case Alpha token type for tokenizing and de-tokenizing operations across different platforms.

Table 3-32: Upper-case Alpha Tokenization Type properties for different Protectors

<table>
<tr><th>Tokenization Type Properties</th><th colspan="4">Settings</th></tr>
<tr><td>Name</td><td colspan="4">Upper-case Alpha</td></tr>
<tr><td>Token type and Format</td><td colspan="4">Uppercase letters A through Z</td></tr>
<tr><td>Tokenizer</td><td>Length Preservation</td><td>Allow Short Data</td><td>Minimum Length</td><td>Maximum Length</td></tr>
<tr><td rowspan="3">SLT_1_3

SLT_2_3</td><td rowspan="3">Yes</td><td>Yes</td><td>1</td><td rowspan="3">4096</td></tr>
<tr><td>No, return input as it is</td><td rowspan="2">3</td></tr>
<tr><td>No, generate error</td></tr>
<tr><td>No</td><td>NA</td><td>1</td><td>4049</td></tr>
<tr><td>Possibility to set Minimum/ maximum length</td><td colspan="4">No</td></tr>
<tr><td>Left/Right settings</td><td colspan="4">Yes</td></tr>
<tr><td>Internal IV</td><td colspan="4">Yes, if Left/Right settings are non-zero</td></tr>
</table>

Tokenization Type Properties	Settings									
External IV	Yes									
Supported input data types (by Application Protectors) *1	**AP Python** *3	**AP Java** *3	**AP .Net** *3	**AP NodeJS** *3	**AP C** *3	**AP Go** *3				
	BYTES STRING	BYTE[] CHAR[] STRING	STRING BYTE[]	STRING BYTE[]	BYTE[]	[]BYTE STRING				
Supported input data types (by DB Protectors)	**MSSQL Server**	**Oracle**			**DB/2**					
	VARCHAR CHAR	VARCHAR2 CHAR			VARCHAR CHAR					
Supported input data types (by MPP DB Protectors)	**Teradata**	**GPDB**			**IBM Netezza**					
	VARCHAR LATIN CHAR LATIN	VARCHAR			VARCHAR					
Supported input data types (for Big Data Protectors) *1	**MapReduce** *2	**Hive**	**Pig**	**HBase** *2	**Impala**	**Spark** *2	**Spark SQL**	**Presto**		
	BYTE[]	CHAR* 4 STRING	CHARARRAY	BYTE[]	STRING	BYTE[] STRING	STRING	VARCHAR		
Return of Protected value	Yes									
Supported in Protegrity releases	6.6.x – 9.x.x.x									
Token specific properties	Lower case characters are accepted in the input but they will be converted to upper-case in output value									

Note:

*1 – If the input and output types of the API are BYTE[], then the customer application should convert the input to and output from the byte array, before calling the API.

*2 – The Protegrity MapReduce protector, HBase coprocessor, and Spark protector only support *bytes* converted from the *string* data type. If any other data type is directly converted to *bytes* and passed as input to the MapReduce or Spark API that supports *byte* as input and provides *byte* as output, then data corruption might occur. If any other data type is directly converted to *bytes* and inserted in an HBase table, which is configured with the Protegrity HBase coprocessor, then data corruption might occur.

*3 – The Protegrity Application Protectors only support *bytes* converted from the *string* data type. If *int*, *short*, or *long* format data is directly converted to *bytes* and passed as input to the Application Protector APIs that support *byte* as input and provide *byte* as output, then data corruption might occur.

*4 – If you are using the *Char* tokenization UDFs in Hive, then ensure that the data elements have length preservation selected. In *Char* tokenization UDFs, using data elements without length preservation selected, is not supported.

The following table shows examples of the way in which a value will be tokenized with the Upper-case Alpha token.

Table 3-33: Examples - Upper Case Alpha tokenization values

Input Value	Tokenized Value	Comments
abc	OIM	Upper-case Alpha, SLT_2_3, Left=0, Right=0, Length Preservation=Yes The value has minimum length for SLT_2_3 tokenizer. Lowercase characters in the input are converted to uppercase in output. De-tokenization will return ‘ABC’.
NY	ZIZ	Upper-case Alpha, SLT_1_3, Left=0, Right=0, Length Preservation=No The value is padded up to 3 characters which is minimum length for SLT_1_3 tokenizer.
NY	Error. Input too short.	Upper-case Alpha, SLT_2_3, Left=0, Right=0, Length Preservation=Yes, Allow Short Data=No, generate error Input value has only two alpha characters to tokenize, which is short for SLT_2_3 tokenizer when Length Preservation=Yes and Allow Short Data=No, generate error.
NY NYA	NY ZIO	Upper-case Alpha, SLT_2_3, Left=0, Right=0, Length Preservation=Yes, Allow Short Data=No, return input as it is If the input value has less than three characters to tokenize, then it is returned as is else it is tokenized.
NY	ZI	Upper-case Alpha, SLT_2_3, Left=0, Right=0, Length Preservation=Yes, Allow Short Data=Yes Input value has only two alpha characters to tokenize, which meets minimum length requirement for SLT_2_3 tokenizer when Length Preservation=Yes and Allow Short Data=Yes.
131 Summer Street, Bridgewater	131 ZBXDPW G FYTZP, CRTTPXPLYGCU	Upper-case Alpha, SLT_1_3, Left=0, Right=0, Length Preservation=No Numeric characters, spaces and comma are treated as delimiters and not tokenized. Output value is longer than initial value.

Input Value	Tokenized Value	Comments
Albert Einstein	AOALXO POHLFHMU	Upper-case Alpha, SLT_2_3, Left=0, Right=0, Length Preservation=Yes Space is treated as delimiters and not tokenized. Output value is the same length as initial value.
704-BBJ	704-GTU	Upper-case Alpha, SLT_1_3, Left=3, Right=0, Length Preservation=Yes 3 characters from left are left in clear. Dash is treated as delimiter. The rest of the value is tokenized.

3.4.6 Alpha-Numeric (0-9, a-z, A-Z)

The Alpha-numeric token type tokenizes all alphabetic symbols (both lowercase and uppercase letters), as well as digits from 0 to 9.

Table 3-34: Alpha-Numeric Tokenization Type properties for different Protectors

<table>
<tr><th>Tokenization Type Properties</th><th colspan="4">Settings</th></tr>
<tr><td>Name</td><td colspan="4">Alpha-Numeric</td></tr>
<tr><td>Token type and Format</td><td colspan="4">Digits 0 through 9

Lowercase letters a through z

Uppercase letters A through Z</td></tr>
<tr><td>Tokenizer</td><td>Length Preservation</td><td>Allow Short Data</td><td>Minimum Length</td><td>Maximum Length</td></tr>
<tr><td rowspan="3">SLT_1_3

SLT_2_3</td><td rowspan="3">Yes</td><td>Yes</td><td>1</td><td rowspan="3">4096</td></tr>
<tr><td>No, return input as it is</td><td rowspan="2">3</td></tr>
<tr><td>No, generate error</td></tr>
<tr><td>No</td><td>NA</td><td>1</td><td>4080</td></tr>
<tr><td>Preserve Case</td><td colspan="4" rowspan="2">Yes, if SLT_2_3 tokenizer is selected

Note:
If you are selecting the Preserve Case or Preserve Position property on the ESA Web UI, then the Preserve Length property is enabled and Allow Short Data property is set to Yes, by default. In addition, these two properties are not modifiable.</td></tr>
<tr><td>Preserve Position</td></tr>
</table>

<table>
<tr><th>Tokenization Type Properties</th><th>Settings</th></tr>
<tr><td>Possibility to set Minimum/ maximum length</td><td>No</td></tr>
<tr><td>Left/Right settings</td><td>Yes

Note:
If you are selecting the Preserve Case or Preserve Position property on the ESA Web UI, then the retention of characters or digits from the left and the right are disabled, by default. In addition, the From Left and From Right properties are both set to zero.</td></tr>
<tr><td>Internal IV</td><td>Yes, if Left/Right settings are non-zero

Note:
If you are selecting the Preserve Case or Preserve Position property on the ESA Web UI, then the alphabetic part of the input value is applied as an internal IV to the numeric part of the input value prior to tokenization.</td></tr>
<tr><td>External IV</td><td>Yes

Note:
If you are selecting the Preserve Case or Preserve Position property on the ESA Web UI, then the external IV property is not supported.</td></tr>
<tr><td>Supported input data types (by Application Protectors) *1</td><td>
<table>
<tr><th>AP Python *3</th><th>AP Java *3</th><th>AP .Net*3</th><th>AP NodeJS*3</th><th>AP C*3</th><th>AP Go*3</th></tr>
<tr><td>STRING

BYTES</td><td>STRING

CHAR[]

BYTE[]</td><td>STRING

BYTE[]</td><td>STRING

BYTE[]</td><td>BYTE[]</td><td>STRING

[]BYTE</td></tr>
</table>
</td></tr>
<tr><td>Supported input data types (by DB Protectors)</td><td>
<table>
<tr><th>MSSQL Server</th><th>Oracle</th><th>DB/2</th></tr>
<tr><td>VARCHAR

CHAR</td><td>VARCHAR2

CHAR</td><td>VARCHAR
CHAR</td></tr>
</table>
</td></tr>
<tr><td>Supported input data types (by MPP DB Protectors)</td><td>
<table>
<tr><th>Teradata</th><th>GPDB</th><th>IBM Netezza</th></tr>
<tr><td>VARCHAR LATIN

CHAR LATIN</td><td>VARCHAR

CHAR</td><td>VARCHAR</td></tr>
</table>
</td></tr>
</table>

Tokenization Type Properties	Settings							
Supported input data types (for Big Data Protectors) *1	**MapReduce** *2	**Hive**	**Pig**	**HBase** *2	**Impala**	**Spark** *2	**Spark SQL**	**Presto**
	BYTE[]	CHAR*4 STRING	CHARARRAY	BYTE[]	STRING	BYTE[] STRING	STRING	VARCHAR
Return of Protected value	Yes							
Supported in Protegrity releases	6.6.x – 9.x.x.x							
Token specific properties	None							

Note:

*1 – If the input and output types of the API are BYTE[], then the customer application should convert the input to and output from the byte array, before calling the API.

*2 – The Protegrity MapReduce protector, HBase coprocessor, and Spark protector only support *bytes* converted from the *string* data type. If any other data type is directly converted to *bytes* and passed as input to the MapReduce or Spark API that supports *byte* as input and provides *byte* as output, then data corruption might occur. If any other data type is directly converted to *bytes* and inserted in an HBase table, which is configured with the Protegrity HBase coprocessor, then data corruption might occur.

*3 – The Protegrity Application Protectors only support *bytes* converted from the *string* data type. If *int*, *short*, or *long* format data is directly converted to *bytes* and passed as input to the Application Protector APIs that support *byte* as input and provide *byte* as output, then data corruption might occur.

*4 – If you are using the *Char* tokenization UDFs in Hive, then ensure that the data elements have length preservation selected. In *Char* tokenization UDFs, using data elements without length preservation selected, is not supported.

The following table shows examples of the way in which a value will be tokenized with the Alpha-Numeric token.

Table 3-35: Tokenization for Alpha-Numeric Values

Input Value	Tokenized Value	Comments
123	sQO	Alpha-Numeric, SLT_1_3, Left=0, Right=0, Length Preservation=Yes Input is numeric but tokenized value contains uppercase and lowercase alpha characters.
NY	1DT	Alpha-Numeric, SLT_2_3, Left=0, Right=0, Length Preservation=No The value is padded up to 3 characters which is minimum length for SLT_2_3 tokenizer.
j1	4t	Alpha-Numeric, SLT_1_3, Left=0, Right=0, Length Preservation=Yes, Allow Short Data=Yes The minimum length meets the requirement for SLT_1_3 tokenizer when Length Preservation=Yes and Allow Short Data=Yes.

Input Value	Tokenized Value	Comments
j1	Error. Input too short.	Alpha-Numeric, SLT_1_3, Left=0, Right=0, Length Preservation=Yes, Allow Short Data=No, generate error The input has two characters to tokenize, which is short for SLT_1_3 tokenizer when Length Preservation=Yes and Allow Short Data=No, generate error.
j1 j1Y	j1 4tD	Alpha-Numeric, SLT_1_3, Left=0, Right=0, Length Preservation=Yes, Allow Short Data=No, return input as it is If the input value has less than three characters to tokenize, then it is returned as is else it is tokenized.
131 Summer Street, Bridgewater	ikC ejCxxp kLa 2ZZ, 5x8K2IMubcn	Alpha-Numeric, SLT_2_3, Left=0, Right=0, Length Preservation=No Spaces and comma are treated as delimiters and not tokenized.
704-BBJ	jf7-oVY	Alpha-Numeric, SLT_1_3, Left=0, Right=0, Length Preservation=Yes Dash is treated as delimiter. The rest of value is tokenized.
704-BBJ	uHq-fTr	Alpha-Numeric, SLT_2_3, Left=0, Right=0, Length Preservation=Yes Dash is treated as delimiter. The rest of value is tokenized.
Protegrity2012	Pr3CYMPilr9n12	Alpha-Numeric, SLT_1_3, Left=2, Right=2, Length Preservation=Yes 2 characters from left and 2 characters from right are left in clear. The rest of value is tokenized.

3.4.7 Upper Alpha-Numeric (0-9, A-Z)

The Upper Alpha-Numeric token type tokenizes uppercase letters A through Z and digits 0 to 9.

> **Note:** In z/OS platform, the Upper Alpha-Numeric token type considers lowercase characters as delimiter. It is recommended not to use Upper Alpha-Numeric token type for tokenizing and de-tokenizing operations across different platforms.

Table 3-36: Upper Alpha-Numeric Tokenization Type properties for different Protectors

Tokenization Type Properties	Settings			
Name	Upper Alpha-Numeric			
Token type and Format	Digits 0 through 9 Uppercase letters A through Z			
Tokenizer	**Length Preservation**	**Allow Short Data**	**Minimum Length**	**Maximum Length**

<table>
<tr><th>Tokenization Type Properties</th><th colspan="12">Settings</th></tr>
<tr><td rowspan="3">SLT_1_3

SLT_2_3</td><td rowspan="3">Yes</td><td>Yes</td><td>1</td><td colspan="8">4096</td></tr>
<tr><td>No, return input as it is</td><td rowspan="2">3</td><td colspan="8" rowspan="2"></td></tr>
<tr><td>No, generate error</td></tr>
<tr><td>No</td><td>NA</td><td>1</td><td colspan="8">4064</td></tr>
<tr><td>Possibility to set Minimum/ maximum length</td><td colspan="12">No</td></tr>
<tr><td>Left/Right settings</td><td colspan="12">Yes</td></tr>
<tr><td>Internal IV</td><td colspan="12">Yes, if Left/Right settings are non-zero</td></tr>
<tr><td>External IV</td><td colspan="12">Yes</td></tr>
<tr><td rowspan="2">Supported input data types (by Application Protectors) *1</td><td colspan="2">AP Python *3</td><td>AP Java*3</td><td>AP .Net* 3</td><td>AP NodeJS* 3</td><td colspan="2">AP C*3</td><td colspan="4">AP Go*3</td></tr>
<tr><td colspan="2">STRING

BYTES</td><td>STRING

CHAR[]

BYTE[]</td><td>STRING

BYTE[]</td><td>STRING

BYTE[]</td><td colspan="2">BYTE[]</td><td colspan="4">STRING

[]BYTE</td></tr>
<tr><td rowspan="2">Supported input data types (by DB Protectors)</td><td colspan="2">MSSQL Server</td><td colspan="3">Oracle</td><td colspan="6">DB/2</td></tr>
<tr><td colspan="2">VARCHAR

CHAR</td><td colspan="3">VARCHAR2

CHAR</td><td colspan="6">VARCHAR CHAR</td></tr>
<tr><td rowspan="2">Supported input data types (by MPP DB Protectors)</td><td colspan="2">Teradata</td><td colspan="3">GPDB</td><td colspan="6">IBM Netezza</td></tr>
<tr><td colspan="2">VARCHAR LATIN

CHAR LATIN</td><td colspan="3">VARCHAR</td><td colspan="6">VARCHAR</td></tr>
<tr><td>Supported input data types (for Big Data Protectors) *1</td><td>MapReduce *2</td><td>Hive</td><td>Pig</td><td>HBase *2</td><td>Impala</td><td colspan="3">Spark *2</td><td>Spark SQL</td><td colspan="2">Presto</td></tr>
</table>

Tokenization Type Properties	Settings							
	BYTE[]	CHAR*4 STRING	CHARARRAY	BYTE[]	STRING	BYTE[] STRING	STRING	VARCHAR
Return of Protected value	Yes							
Supported in Protegrity releases	6.6.x – 9.x.x.x							
Token specific properties	Lower case characters are accepted in the input but they will be converted to upper-case in output value.							

Note:

*1– If the input and output types of the API are BYTE[], then the customer application should convert the input to and output from the byte array, before calling the API.

*2– The Protegrity MapReduce protector, HBase coprocessor, and Spark protector only support *bytes* converted from the *string* data type. If any other data type is directly converted to *bytes* and passed as input to the MapReduce or Spark API that supports *byte* as input and provides *byte* as output, then data corruption might occur. If any other data type is directly converted to *bytes* and inserted in an HBase table, which is configured with the Protegrity HBase coprocessor, then data corruption might occur.

*3 – The Protegrity Application Protectors only support *bytes* converted from the *string* data type. If *int*, *short*, or *long* format data is directly converted to *bytes* and passed as input to the Application Protector APIs that support *byte* as input and provide *byte* as output, then data corruption might occur.

*4 – If you are using the *Char* tokenization UDFs in Hive, then ensure that the data elements have length preservation selected. In *Char* tokenization UDFs, using data elements without length preservation selected, is not supported.

The following table shows examples of the way in which a value will be tokenized with the Upper Alpha-Numeric token.

Table 3-37: Tokenization for Upper-case Alpha-Numeric Values

Input Value	Tokenized Value	Comments
123	STD	Upper Alpha-Numeric, SLT_1_3, Left=0, Right=0, Length Preservation=Yes Input is numeric but tokenized value contains uppercase alpha characters.
J1	4T	Upper Alpha-Numeric, SLT_1_3, Left=0, Right=0, Length Preservation=Yes, Allow Short Data=Yes The minimum length meets the requirement for SLT_1_3 tokenizer when Length Preservation=Yes and Allow Short Data=Yes.

Input Value	Tokenized Value	Comments
J1	Error. Input too short.	Upper Alpha-Numeric, SLT_1_3, Left=0, Right=0, Length Preservation=Yes, Allow Short Data=No, generate error The input has two characters to tokenize, which is short for SLT_1_3 tokenizer when Length Preservation=Yes and Allow Short Data=No, generate error.
J1 J1Y	J1 4TD	Upper Alpha-Numeric, SLT_1_3, Left=0, Right=0, Length Preservation=Yes, Allow Short Data=No, return input as it is If the input value has less than three characters to tokenize, then it is returned as is else it is tokenized.
NY	AOZ	Upper Alpha-Numeric, SLT_2_3, Left=0, Right=0, Length Preservation=No The value is padded up to 3 characters which is minimum length for SLT_2_3 tokenizer.
131 Summer Street, Bridgewater	8C9 CSD5PS 1X5 ZJH, 231JHXW8CVF	Upper Alpha-Numeric, SLT_2_3, Left=0, Right=0, Length Preservation=No Spaces and comma are treated as delimiters and not tokenized. Lowercase characters in the input are converted to uppercase in output. De-tokenization will return all alpha characters in upper case.
704-BBJ	704-EC0	Upper Alpha-Numeric, SLT_1_3, Left=3, Right=0, Length Preservation=Yes 3 characters from the left are left in clear. Dash is treated as delimiter. The rest of value is tokenized.
704-BBJ	704-HHT	Upper Alpha-Numeric, SLT_2_3, Left=3, Right=0, Length Preservation=Yes 3 characters from the left are left in clear. Dash is treated as delimiter. The rest of value is tokenized.
support@protegrity.com	FKNKHHQ@72CN84 UKEI.com	Upper Alpha-Numeric, SLT_2_3, Left=0, Right=3, Length Preservation=Yes 3 characters from right are left in clear. ‘@’ and ‘.’ are treated as delimiters. The rest of value is tokenized. De-tokenization will return all alpha characters in upper case.

3.4.8 Lower ASCII

The Lower ASCII token type is provided to address the handling of spaces in such data types as CHAR and VARCHAR.

Table 3-38: Lower ASCII Tokenization Type properties for different Protectors

Tokenization Type Properties	Settings
Name	Lower ASCII
Token type and Format	The lower part of ASCII table. Hex character codes from 0x21 to 0x7E. For the list of ASCII characters supported by Lower ASCII token, refer to

<table>
<tr><th>Tokenization Type Properties</th><th colspan="11">Settings</th></tr>
<tr><td></td><td colspan="11">Appendix A: ASCII Character Codes</td></tr>
<tr><td>Tokenizer</td><td>Length Preservation</td><td colspan="3">Allow Short Data</td><td colspan="3">Minimum Length</td><td colspan="3">Maximum Length</td></tr>
<tr><td rowspan="3">SLT_1_3</td><td rowspan="3">Yes</td><td colspan="3">Yes</td><td colspan="3">1</td><td colspan="3">4096</td></tr>
<tr><td colspan="3">No, return input as it is</td><td colspan="3" rowspan="2">3</td><td colspan="3" rowspan="2">4096</td></tr>
<tr><td colspan="3">No, generate error</td></tr>
<tr><td>No</td><td colspan="3">NA</td><td colspan="3">1</td><td colspan="3">4086</td></tr>
<tr><td>Possibility to set Minimum/ maximum length</td><td colspan="11">No</td></tr>
<tr><td>Left/Right settings</td><td colspan="11">Yes</td></tr>
<tr><td>Internal IV</td><td colspan="11">Yes, if Left/Right settings are non-zero</td></tr>
<tr><td>External IV</td><td colspan="11">Yes</td></tr>
<tr><td rowspan="2">Supported input data types (by Application Protectors) *1</td><td colspan="2">AP Python *4</td><td colspan="2">AP Java*4</td><td colspan="2">AP .Net*4</td><td colspan="2">AP NodeJS*4</td><td colspan="2">AP C*4</td><td>AP Go*4</td></tr>
<tr><td colspan="2">STRING
BYTES</td><td colspan="2">STRING
CHAR[]
BYTE[]</td><td colspan="2">STRING
BYTE[]</td><td colspan="2">STRING
BYTE[]</td><td colspan="2">BYTE[]</td><td>STRING
[]BYTE</td></tr>
<tr><td rowspan="2">Supported input data types (by DB Protectors)</td><td colspan="3">MSSQL Server</td><td colspan="5">Oracle</td><td colspan="3">DB/2</td></tr>
<tr><td colspan="3">VARCHAR*5
CHAR</td><td colspan="5">VARCHAR2
CHAR</td><td colspan="3">VARCHAR CHAR</td></tr>
<tr><td rowspan="2">Supported input data types (by MPP DB Protectors)</td><td colspan="3">Teradata</td><td colspan="5">GPDB</td><td colspan="3">IBM Netezza</td></tr>
<tr><td colspan="3">VARCHAR LATIN
CHAR LATIN</td><td colspan="5">VARCHAR</td><td colspan="3">VARCHAR</td></tr>
<tr><td>Supported input data types (for Big Data Protectors) *1</td><td>Map Reduce *3</td><td>Hive*2</td><td>Pig*2</td><td>HBase *3</td><td colspan="4">Impala*2</td><td>Spark *3</td><td>Spark SQL</td><td>Presto*3</td></tr>
</table>

Tokenization Type Properties	Settings							
	BYTE[]	STRING	CHARARRAY	BYTE[]	STRING	BYTE[] STRING	STRING	VARCHAR
Return of Protected value	Yes							
Supported in Protegrity releases	6.6.x – 9.x.x.x							
Token specific properties	Space character is treated as delimiter							

Note:

*1– If the input and output types of the API are BYTE[], then the customer application should convert the input to and output from the byte array, before calling the API.

2– Ensure that you use the Horizontal tab (\t*) as the field or column delimiter when loading data that is tokenized using Lower ASCII tokens for Hive, Pig, Impala, and Presto.

*3– The Protegrity MapReduce protector, HBase coprocessor, and Spark protector only support *bytes* converted from the *string* data type. If any other data type is directly converted to *bytes* and passed as input to the MapReduce or Spark API that supports *byte* as input and provides *byte* as output, then data corruption might occur. If any other data type is directly converted to *bytes* and inserted in an HBase table, which is configured with the Protegrity HBase coprocessor, then data corruption might occur.

*4 – The Protegrity Application Protectors only support *bytes* converted from the *string* data type. If *int*, *short*, or *long* format data is directly converted to *bytes* and passed as input to the Application Protector APIs that support *byte* as input and provide *byte* as output, then data corruption might occur.

*5 – Lower ASCII tokenization is not used with JSON or XML UDFs.

The following table shows examples of the way in which a value will be tokenized with the Lower ASCII token.

Table 3-39: Tokenization for Lower ASCII Values

Input Value	Tokenized Value	Comments
La Scala 05698	:H HnwqP v/Q`>	All characters in the input value (except of spaces) are tokenized.
Ford Mondeo CA-0256TY M34 567 K-45	j`1$ nRSD<X T]!(~4MWF l:f cF+ R?V{	All characters in the input value (except of spaces) are tokenized.
ac	;H	Lower ASCII, SLT_1_3, Left=0, Right=0, Length Preservation=Yes, Allow Short Data=Yes The minimum length meets the requirement for the SLT_1_3 tokenizer when Length Preservation=Yes and Allow Short Data=Yes.

Input Value	Tokenized Value	Comments
ac	Error. Input too short.	Lower ASCII, SLT_1_3, Left=0, Right=0, Length Preservation=Yes, Allow Short Data=No, generate an error The input has two characters to tokenize, which is short for SLT_1_3 tokenizer when Length Preservation=Yes and Allow Short Data=No, generate an error.
ac aca	ac ;HH	Lower ASCII, SLT_1_3, Left=0, Right=0, Length Preservation=Yes, Allow Short Data=No, return input as it is If the input value has less than three characters to tokenize, then it is returned as is else it is tokenized.

3.4.9 Printable

The Printable token type tokenizes ASCII printable characters from the ISO 8859-15 alphabet, which include letters, digits, punctuation marks, and miscellaneous symbols.

Table 3-40: Printable Tokenization Type properties for different Protectors

Tokenization Type Properties	Settings			
Name	Printable			
Token type and Format	ASCII printable characters, which include letters, digits, punctuation marks, and miscellaneous symbols Hex character codes from 0x20 to 0x7E and from 0xA0 to 0xFF. Refer to Appendix A: ASCII Character Codes for the list of ASCII characters supported by Printable token.			
Tokenizer*6 *7	**Length Preservation**	**Allow Short Data**	**Minimum Length**	**Maximum Length**
SLT_1_3	Yes	Yes	1	4096
		No, return input as it is	3	
		No, generate error		
	No	NA	1	4091
Possibility to set Minimum/ maximum length	No			
Left/Right settings	Yes			

Tokenization Type Properties	Settings							
Internal IV	Yes, if Left/Right settings are non-zero							
External IV	Yes							
Supported input data types (by Application Protectors) *1*10	**AP Python** *8	**AP Java** *8	**AP C**[*8]	**AP Go**[*8]				
	STRING BYTES	STRING CHAR[] BYTE[]	BYTE[]	STRING []BYTE				
Supported input data types (by DB Protectors) *10	**MSSQL Server**	**Oracle**	**DB/2**					
	VARCHAR CHAR	VARCHAR2 CHAR	VARCHAR CHAR					
Supported input data types (by MPP DB Protectors) *10	**Teradata**[*11]	**GPDB**	**IBM Netezza**					
	VARCHAR LATIN *9 CHAR LATIN	VARCHAR	VARCHAR					
Supported input data types (for Big Data Protectors) *1*10	**MapReduce** *4*5	**Hive**	**Pig**	**HBase** *4*5	**Impala**[*2*3]	**Spark** *4*5	**Spark SQL**	**Presto**
	BYTE[]	Not supported	Not supported	BYTE[]	STRING	BYTE[] *5	Not supported	VARCHAR
Return of Protected value	Yes							
Supported in Protegrity releases	6.6.x – 9.x.x.x							
Token specific properties	Token tables are large (~ 27MB, refer to the exact numbers in the table *SLT Tokenizer Characteristics*).							

Note:

*1 – If the input and output types of the API are BYTE[], then the customer application should convert the input to and output from the byte array, before calling the API.

2 – Ensure that you use the Horizontal tab (\t*) as the field or column delimiter when loading data that is tokenized using Printable tokens for Impala.

*3 – Though the tokenization results for Impala may not be formatted and displayed accurately, they will be unprotected to the original values, using the respective protector.

*4 – The Protegrity MapReduce protector, HBase coprocessor, and Spark protector only support *bytes* converted from the *string* data type. If any other data type is directly converted to *bytes* and passed as input to the MapReduce or Spark API that supports *byte* as input and provides *byte* as output, then data corruption might occur. If any other data type is directly converted to *bytes* and inserted in an HBase table, which is configured with the Protegrity HBase coprocessor, then data corruption might occur.

*5 – It is recommended to use Printable tokenization only with APIs that accept BYTE[] as input and provide BYTE[] as output. If Printable tokens are generated using APIs that accept BYTE[] as input and provide BYTE[] as output, and uniform encoding is maintained across protectors, then the tokens can be used across various protectors. In addition, for the tokenization results to be formatted and displayed accurately, ensure that the clients use the *ISO 8859-15* character encoding, while converting the input from the *String* data type to *Byte* and passing it as input to the *Byte APIs*.

*6 – The character column (CHAR) to protect is configured to remove trailing spaces before the tokenization. This means that the space character can be lost in translation for Printable tokens. To avoid this consider using Lower ASCII token instead of Printable for CHAR columns and input data having spaces.

*7 – *Printable* tokenization is not supported on databases where the character set is UTF.

*8 – The Protegrity AP C, AP Java, AP Python, and AP Golang protectors only support *bytes* converted from the *string* data type. If any other data type is directly converted to *bytes* and passed as input to the AP C, AP Java, AP Python, and AP Golang APIs that support *byte* as input and provide *byte* as output, then data corruption might occur.

*9 – JSON and XML UDFs are only used for Teradata.

*10 – If Printable tokens are generated using APIs or UDFs that accept STRING or VARCHAR as input, then the protected values can only be unprotected using the protector with which it was protected. If you are unprotecting the protected data using any other protector, then you could get inconsistent results.

*11 – Tokenizing XML or JSON data with Printable tokenization will not return valid XML or JSON format output.

Note:
Printable token data element is unsupported by the AP .Net and AP NodeJS.

Note:
For Non-US, when using *Printable* token, the From Codepage (FRCODEPG) parameter in PTYPARM file has to be set to the current system codepage by the system administrator. This is valid for z/OS customers only.

The following table shows examples of the way in which a value will be tokenized with the Printable token.

Table 3-41: Tokenization for Printable Values

Input Value	Tokenized Value	Comments
La Scala 05698	F|ZpÙç|Ôä%s^¦4	All characters in the input value (including spaces) are tokenized.
Ford Mondeo CA-0256TY M34 567 K-45	§)%ß#)ðYjt{Â¬ÓÊEµV²ù²	All characters in the input value (including spaces) are tokenized.
qw	rD	Printable, SLT_1_3, Left=0, Right=0, Length Preservation=Yes, Allow Short Data=Yes

Input Value	Tokenized Value	Comments
		The minimum length meets the requirement for the SLT_1_3 tokenizer when Length Preservation=Yes and Allow Short Data=Yes.
qw	Error. Input too short.	Printable, SLT_1_3, Left=0, Right=0, Length Preservation=Yes, Allow Short Data=No, generate an error The input has two characters to tokenize, which is short for SLT_1_3 tokenizer when Length Preservation=Yes and Allow Short Data=No, generate an error.
qw qwa	qw rDZ	Printable, SLT_1_3, Left=0, Right=0, Length Preservation=Yes, Allow Short Data=No, return input as it is. If the input value has less than three characters to tokenize, then it is returned as is else it is tokenized.

3.4.10 Date (YYYY-MM-DD, DD/MM/YYYY, MM.DD.YYYY)

The Date token type protects dates in one of the following formats:

- YYYY<delim>MM<delim>DD
- DD<delim>MM<delim>YYYY
- MM<delim>DD<delim>YYYY

where <delim> is one of the allowed separators: . (dot), / (slash), or – (dash).

Supported date formats correspond to the big endian, little endian, and middle endian forms.

Table 3-42: Date Tokenization Type properties for different Databases

Tokenization Type Properties	Settings		
Name	Date		
Token type and Format	Date in big endian form, starting with the year (YYYY-MM-DD). Date in little endian form, starting with the day (DD/MM/YYYY). Date in middle endian form, starting with the month (MM.DD.YYYY). The following separators are supported: . (dot), / (slash), - (dash) supported.		
Tokenizer	**Length Preservation**	**Minimum Length**	**Maximum Length**
SLT_1_3 SLT_2_3 SLT_1_6 SLT_2_6[*5]	Yes	10	10

Tokenization Type Properties	Settings								
Possibility to set Minimum/ maximum length	No								
Left/Right settings	No								
Internal IV	No								
External IV	No								
Supported input data types (by Application Protectors) *1	**AP Python** *4	**AP Java**[*4]	**AP .Net**[*4]	**AP NodeJS**[*4]	**AP C**[*4]	**AP Go**[*4]			
	DATE BYTES STRING	DATE STRING CHAR[] BYTE[]	STRING BYTE[]	STRING BYTE[]	BYTE[]	STRING []BYTE			
Supported input data types (by DB Protectors)	**MSSQL Server**	**Oracle**			**DB/2**				
	VARCHAR CHAR	DATE VARCHAR2 CHAR			DATE CHAR VARCHAR				
Supported input data types (by MPP DB Protectors)	**Teradata**	**GPDB**			**IBM Netezza**				
	VARCHAR LATIN CHAR LATIN	DATE			DATE				
Supported input data types (for Big Data Protectors) *1	**MapReduce** *2	**Hive**	**Pig**	**HBase** *2	**Impala**	**Spark** *2	**Spark SQL**	**Presto**	
	BYTE[]	STRING DATE *3	CHAR ARRAY	BYTE[]	STRING	BYTE[] STRING	STRING DATE[*3]	DATE[*6]	
Return of Protected value	Yes								
Supported in Protegrity releases	6.6.x – 9.x.x.x								
Token specific properties	In Release 5.5 only specific separators were allowed. Starting from 6.0 all separators (. (dot), / (slash), - (dash)) are allowed.								

Tokenization Type Properties	Settings
Supported range of input dates	From “0600-01-01” to “3337-11-27”.

Note:

*1 – If the input and output types of the API are BYTE[], then the customer application should convert the input to and output from the byte array, before calling the API.

*2 – The Protegrity MapReduce protector, HBase coprocessor, and Spark protector only support *bytes* converted from the *string* data type. If any other data type is directly converted to *bytes* and passed as input to the MapReduce or Spark API that supports *byte* as input and provides *byte* as output, then data corruption might occur. If any other data type is directly converted to *bytes* and inserted in an HBase table, which is configured with the Protegrity HBase coprocessor, then data corruption might occur.

*3 – In the Big Data Protector, the date format supported for Hive and Spark SQL is YYYY-MM-DD only.

*4 – The Protegrity Application Protectors only support *bytes* converted from the *string* data type. If *int*, *short*, or *long* format data is directly converted to *bytes* and passed as input to the Application Protector APIs that support *byte* as input and provide *byte* as output, then data corruption might occur.

*5 – The newly created data elements using the SLT_2_6 tokenizer from v7.1 Maintenance Release 1 (MR1) onwards are deployable to protectors with versions 7.1 MR1 and higher.

*6 - In the Presto Protector, the date format supported is YYYY-MM-DD only.

Date token types are not fully validated for input value as valid date. For example, a day value greater than 31 and month value greater than 12 results in an error, but the date input 2011-02-30 does not. However, the detokenized value would be 2011-03-02, which is not the initial value.

The following table shows examples of the way in which a value will be tokenized with the Date token.

Table 3-43: Tokenization Examples for Date

Input Value	Tokenized Value	Comments
2012-02-29 2012/02/29 2012.02.29	2150-02-20 2150/02/20 2150.02.20	Date (YYYY-MM-DD) Token. All three separators are successfully accepted. They are treated as delimiters not impacting tokenized value.
31/01/0600	08/05/2215	Date (DD/MM/YYYY) Token. Date in the past is tokenized.
10.30.3337	09.05.2042	Date (MM.DD.YYYY) Date in the future is tokenized.
2012:08:24 1975-01-32	Token is not generated due to invalid input value. Error is returned.	Date (YYYY-MM-DD) Token.

Input Value	Tokenized Value	Comments
		Input values with non-supported separators or with unsupported dates produce error.

Date Tokenization for Cutover Dates of the Proleptic Gregorian Calendar

The data systems, such as, Oracle or Java-based systems, do not accept the cutover dates of the Proleptic Gregorian Calendar. The cutover dates of the Proleptic Gregorian Calendar fall in the interval *1582-10-05* to *1582-10-14*. These dates are converted to *1582-10-15*. When using Oracle, conversion occurs by adding ten days to the source date. Due to this conversion, data loss occurs as the system is not capable to return the actual date value after the de-tokenization.

The following points are applicable for the tokenization and de-tokenization of the cutover dates of the Proleptic Gregorian Calendar:

- The tokenization of the date values in the cutover date range of the Proleptic Gregorian Calendar results in an 'Invalid Input' error.
- During tokenization, an internal validation is performed to check whether the value is tokenized to the cutover date. If it is a cutover date, then the *Year* part (*1582*) of the tokenized value is converted to *3338* and then returned. During de-tokenization, an internal check is performed to validate whether the *Year* is *3338*. If the *Year* is *3338*, then it is internally converted to *1582*.

> **Note:**
> The tokenization accepts the date range *0600-01-01* to *3337-11-27* excluding the cutover date range.
>
> The de-tokenization accepts the date ranges *0600-01-01* to *3337-11-27* and *3338-10-05* to *3338-10-14*.

Consider a scenario where you are migrating the protected data from Protector 1 to Protector 2. The Protector 1 includes the Date tokenizer update to process the cutover dates of the Proleptic Gregorian Calendar as input. The Protector 2 does not include this update. In such a scenario, an 'Invalid Date Format' error occurs in Protector 2, when you try to unprotect the protected data as it fails to accept the input year *3338*. The following steps must be performed to mitigate this issue:

1. Unprotect the protected data from Protector 1
2. Migrate the unprotected data to Protector 2
3. Protect the data from Protector 2

3.4.11 Datetime (YYYY-MM-DD HH:MM:SS)

The Datetime token type was introduced in response to requirements to allow specific date parts to remain in the clear and for date tokens to be distinguishable from real dates. This token type will also allow for time to be tokenized (HH:MM:SS) with the exception of fractions of a second, including milliseconds (MMM), microseconds (mmmmmm), and nanoseconds (nnnnnnnnn).

Table 3-44: Datetime Tokenization Type properties for different Protectors

Tokenization Type Properties	Settings
Name	Datetime
Token type and Format	Datetime in the following formats: YYYY-MM-DD HH:MM:SS.MMM[*5] YYYY-MM-DDTHH:MM:SS.MMM[*5]

<table>
<tr><th>Tokenization Type Properties</th><th colspan="6">Settings</th></tr>
<tr><td></td><td colspan="6">YYYY-MM-DD HH:MM:SS.mmmmmm*5
YYYY-MM-DDTHH:MM:SS.mmmmmm*5
YYYY-MM-DD HH:MM:SS.nnnnnnnnn*5
YYYY-MM-DDTHH:MM:SS.nnnnnnnnn*5
YYYY-MM-DD HH:MM:SS
YYYY-MM-DDTHH:MM:SS
YYYY-MM-DD
From “0600-01-01” to “3337-11-27”, alphabetic digits 0-9, only specific delimiters*4</td></tr>
<tr><td>Input separators (delimiter) between date, month and year</td><td colspan="6">. (dot), / (slash), or - (dash)</td></tr>
<tr><td>Input separators (delimiter) between hours, minutes and seconds</td><td colspan="6">colon only (“:”)</td></tr>
<tr><td>Input separator (delimiter) between date and hour</td><td colspan="6">space (“ ”) or T letter (“T”)</td></tr>
<tr><td rowspan="2">Input separator (delimiter) between seconds and milliseconds</td><td colspan="6">For DATE datatype “.” (dot)</td></tr>
<tr><td colspan="6">For CHAR, VARCHAR, and STRING datatypes “.” (dot), “,” (comma)</td></tr>
<tr><td>Tokenizer</td><td>Length Preservation</td><td colspan="2">Minimum Length</td><td colspan="2">Maximum Length</td></tr>
<tr><td>SLT_DATETIME</td><td>Yes</td><td colspan="2">10</td><td colspan="2">29</td></tr>
<tr><td>Possibility to set Minimum/ maximum length</td><td colspan="6">No</td></tr>
<tr><td>Left/Right settings</td><td colspan="6">No</td></tr>
<tr><td>Internal IV</td><td colspan="6">No</td></tr>
<tr><td>External IV</td><td colspan="6">No</td></tr>
<tr><td rowspan="2">Supported input data types (by Application Protectors) *1</td><td>AP Python *3</td><td>AP Java*3</td><td>AP NodeJS*3</td><td>AP C*3</td><td>AP .Net*3</td><td>AP Go*3</td></tr>
<tr><td>DATE
BYTES
STRING</td><td>DATE
STRING
CHAR[]</td><td>STRING
BYTE[]</td><td>BYTE[]</td><td>STRING
BYTE[]</td><td>STRING
[]BYTE</td></tr>
</table>

Tokenization Type Properties	Settings							
			BYTE[]					
Supported input data types (by DB Protectors)	**MSSQL Server**		**Oracle**		**DB/2**			
	VARCHAR CHAR		DATE VARCHAR2 CHAR		DATE CHAR VARCHAR			
Supported input data types (by MPP DB Protectors)	**Teradata**		**GPDB**		**IBM Netezza**			
	VARCHAR LATIN CHAR LATIN		VARCHAR		VARCHAR			
Supported input data types (for Big Data Protectors) *1	**MapReduce** *2	**Hive**	**Pig**	**HBase** *2	**Impala**	**Spark** *2	**Spark SQL**	**Presto**
	BYTE[]	STRING DATETIME	CHARARRAY	BYTE[]	STRING	BYTE[] STRING	STRING DATETIME	TIMESTAMP
Return of Protected value	Yes							
Supported in Protegrity releases	6.6.x – 9.x.x.x							
Token specific properties								
Tokenize time	Yes/No							
Distinguishable date	Yes/No							
Date in clear	Month/Year/None							

Note:

*1 – If the input and output types of the API are BYTE[], then the customer application should convert the input to and output from the byte array, before calling the API.

*2 – The Protegrity MapReduce protector, HBase coprocessor, and Spark protector only support *bytes* converted from the *string* data type. If any other data type is directly converted to *bytes* and passed as input to the MapReduce or Spark API that supports *byte* as input and provides *byte* as output, then data corruption might occur. If any other data type is directly converted to *bytes* and inserted in an HBase table, which is configured with the Protegrity HBase coprocessor, then data corruption might occur.

*3 – The Protegrity Application Protectors only support *bytes* converted from the *string* data type. If *int*, *short*, or *long* format data is directly converted to *bytes* and passed as input to the Application Protector APIs that support *byte* as input and provide *byte* as output, then data corruption might occur.

[*4] – The Microsoft SQL Server accepts input value ranging between “1753-01-01” through “9999-12-3” for *DATETIME* data type. As the tokenized value ranges between “0600-01-01” through “3337-11-27”, the incompatible values are not supported by the column defined with *DATETIME* data type. The following solutions are available to address this issue:

- Retain the *Year* using the **Date in Clear** property. The tokenized value retains the *Year* in clear.
- Change the column data type to *DATETIME2* date type. The Microsoft SQL Server supports input value ranging between “0001-01-01 00:00:00.0000000” through “9999-12-31 23:59:59.9999999” for *DATETIME2* data type.

[*5] - The Protegrity Datetime token type supports fractions of a second as an input based on the version of protector you are using, as specified in the following list:

- If you are using an older release of the protector version 9.1.0.0 or an older version of the protector, then the Protegrity Datetime token supports fractions of a second as an input, only up to 3 digits.
- If you are using a newer release of the protector version 9.1.0.0 or a later version of the protector, then the Protegrity Datetime token supports fractions of a second as an input, up to 9 digits.

  For more information on whether the protector that you are using supports this new functionality, refer to the respective protector Readme.

Note:

The *Datetime* token type and the DB2 Timestamp data type have incompatible date formats, hence the *Datetime* token type is not supported for z/OS protectors except for z/OS UDFs.

The z/OS Protectors from build number 7.0.1.17, support the *Datetime* tokens. However, the *Datetime* tokens with tokenized time set to *Yes* are not supported.

Tokenize Time property defines whether the time part (HH:MM:SS) will be tokenized. If Tokenize Time is set to “No”, then the time part will be treated as a delimiter and will be added to the date after tokenization.

Distinguishable Date property defines whether the tokenized values will be outside of the normal date range.

If the Distinguishable Date option is enabled, then all tokenized dates will be in the range from *year 5596-09-06* to *8334-08-03*. The tokenized value will become recognizable.

If the Distinguishable Date option is disabled, then the tokenized dates will be in the range from *year 0600-01-01* to *3337-11-27*. As an example, tokenizing "2012-04-25" will result in “1856-12-03” (non-distinguishable) and “6457-07-12” (distinguishable).

The **Date in Clear** property defines whether Month or Year will be left in the clear in the tokenized value.

Note: You cannot use enabled Distinguishable Date and select month or year to be left in the clear at the same time

The following points are applicable when you tokenize the Dates with *Year* as *3337* by setting the *Year* part to be in clear:

- The tokenized Date value can be outside of the accepted Date range.
- The tokenized Date value can be de-tokenized to obtain the original Date value.

For example, if the Date *3337-11-27* is tokenized by setting the Year part *3337* in clear, then the resultant tokenized value *3337-12-15* is outside of the accepted Date range. The de-tokenization of this tokenized value returns the original Date *3337-11-27*.

The following table shows examples of the way in which a value will be tokenized with the Datetime token.

Table 3-45: Tokenization for DateTime Values

Input Value	Tokenized Value	Comments
2009.04.12 12:23:34.333	1595.06.19 14:31:51.333	YYYY-MM-DD HH:MM:SS.MMM. Milliseconds value is left in the clear.
2009.04.12 12:23:34.333666	1595.06.19 14:31:51.333666	YYYY-MM-DD HH:MM:SS.mmmmmm. The microseconds value is left in the clear.
2009.04.12 12:23:34.333666999	1595.06.19 14:31:51.333666999	YYYY-MM-DD HH:MM:SS.nnnnnnnnn. The nanoseconds value is left in the clear.
2009.04.12 12:23:34	1595.06.19 14:31:51	YYYY-MM-DD HH:MM:SS with space separator between day and hour.
2234.10.12T12:23:23	2755.08.04T22:33:43	YYYY-MM-DDTHH:MM:SS with T separator between day and hour values.
2009.04.12 12:23:34.333	5150.05.14T17:49:34.333	Datetime with distinguishable date on (the year value is unreal).
2234.12.22 22:53:34	2755.03.15 19:03:21	Datetime token in any format with distinguishable date off (the year value is real).
2009.04.12 12:23:34.333	1595.04.19 14:31:51.333	Datetime token with month in the clear.
2009.04.12 12:23:34.333	2009.06.19 14:31:51.333	Datetime token with year in the clear.

Datetime Tokenization for Cutover Dates of the Proleptic Gregorian Calendar

The data systems, such as, Oracle or Java-based systems, do not accept the cutover dates of the Proleptic Gregorian Calendar. The cutover dates of the Proleptic Gregorian Calendar fall in the interval *1582-10-05* to *1582-10-14*. These dates are converted to *1582-10-15*. When using Oracle, conversion occurs by adding ten days to the source date. Due to this conversion, data loss occurs as the system is not capable to return the actual date value after the de-tokenization.

> **Note:** The tokenization of the Date values in the cutover Date range of the Proleptic Gregorian Calendar results in an 'Invalid Input' error.

The following points are applicable when the *Distinguishable Date* option is disabled:

- If the *Distinguishable Date* option is disabled, then the tokenized dates are in the range *0600-01-01* to *3337-11-27*, which also includes the cutover date range. During tokenization, an internal validation is performed to check whether the value is tokenized to the cutover date. If it is a cutover date, then the *Year* part (*1582*) of the tokenized value is converted to *3338* and then returned.
- During de-tokenization, an internal check is performed to validate whether the Year is *3338*. If the Year is *3338*, then it is internally converted to *1582*.

The following points are applicable when you tokenize the dates from the Year *1582* by setting the *Year* part to be in clear:

- The tokenized value can result in the cutover Date range. In such a scenario, the *Year* part of the tokenized Date value is converted to *3338*.
- During de-tokenization, the *Year* part of the Date value is converted to *1582* to obtain the original date value.

For example, if the date *1582.04.30 12:12:12* is tokenized by setting the *Year* part in clear and the resultant tokenized value falls in the cutover Date range, then the *Year* part is converted to *3338* resulting in a tokenized value as *3338.10.10 12:12:12*. The de-tokenization of this tokenized value returns the original Date *1582.04.30 12:12:12*.

> **Note:**
> The tokenization accepts the date range *0600-01-01* to *3337-11-27* excluding the cutover date range.
>
> The de-tokenization accepts the date range *0600-01-01* to *3337-11-27* and date values from the Year *3338*.

Consider a scenario where you are migrating the protected data from Protector 1 to Protector 2. The Protector 1 includes the Datetime tokenizer update to process the cutover dates of the Proleptic Gregorian Calendar as input. The Protector 2 does not include this update. In such a scenario, an 'Invalid Date Format' error occurs in Protector 2, when you try to unprotect the protected data as it fails to accept the input year *3338*. The following steps must be performed to mitigate this issue:

1. Unprotect the protected data from Protector 1
2. Migrate the unprotected data to Protector 2
3. Protect the data from Protector 2

3.4.12 Decimal

The Decimal token type tokenizes numbers which may have a precision and scale. The resulting token does not contain any zeros which makes it suitable to store in a decimal data type in a database. Any sign or decimal point delimiter are stripped from the input value before tokenization and put back after tokenization.

> **Note:** When data with decimal point delimiter is protected, the number of digits counted after the decimal point are length preserving. For example, consider decimal data '345645.345' is protected to return the protected value as '3456736.768'. The number of digits that exist after the decimal point remain the same in both the values.

Table 3-46: Decimal Tokenization Type properties for different Protectors

Tokenization Type Properties	**Settings**		
Name	Decimal		
Token type and Format	Digits 0 through 9 in input value, 1 thorough 9 in output value The sign (+ or -) and decimal point (. or ,) separator		
Tokenizer	**Length Preservation**	**Minimum Length**	**Maximum Length**
SLT_6_DECIMAL	No	1	36 *2
Possibility to set Minimum/ maximum length	Yes		
Left/Right settings	No		

Tokenization Type Properties	Settings							
Internal IV	No							
External IV	No							
Supported input data types (by Application Protectors) *1	**AP Python** *4	**AP Java** *4	**AP NodeJS**[*3]	**AP C**[*4]	**AP .Net**[*3]	**AP Go**[*4]		
	STRING BYTES	STRING CHAR[] BYTE[]	STRING BYTE[]	BYTE[]	STRING BYTE[]	STRING []BYTE		
Supported input data types (by DB Protectors)	**MSSQL Server**	**Oracle**	**DB/2**					
	VARCHAR CHAR	NUMBER (p,s) VARCHAR2 CHAR	CHAR VARCHAR2					
Supported input data types (by MPP DB Protectors)	**Teradata**	**GPDB**	**IBM Netezza**					
	VARCHAR LATIN CHAR LATIN	VARCHAR	VARCHAR					
Supported input data types (for Big Data Protectors) *1	**MapReduce** *3	**Hive**	**Pig**	**HBase** *3	**Impala**	**Spark** *3	**Spark SQL**	**Presto**
	BYTE[]	STRING	CHAR ARRAY	BYTE[] *3	STRING	BYTE[] STRING	STRING	VARCHAR
Return of Protected value	Yes							
Supported in Protegrity releases	6.6.x – 9.x.x.x							
Token specific properties	Supports Numeric data with precision and scale. The token will not contain any zeros.							

Note:

*1 – If the input and output types of the API are BYTE[], then the customer application should convert the input to and output from the byte array, before calling the API.

*2 – The configurable input length for decimal values is between 1 and 36 digits. The upper range is 38 digits. However, since decimal token is not length preserving, only up to 36 digits are supported.

*3 – The Protegrity MapReduce protector, HBase coprocessor, and Spark protector only support *bytes* converted from the *string* data type. If any other data type is directly converted to *bytes* and passed as input to the MapReduce or Spark API that supports *byte* as input and provides *byte* as output, then data corruption might occur. If any other data type is directly converted to *bytes* and inserted in an HBase table, which is configured with the Protegrity HBase coprocessor, then data corruption might occur.

*4 – The Protegrity Application Protectors only support *bytes* converted from the *string* data type. If *int*, *short*, or *long* format data is directly converted to *bytes* and passed as input to the Application Protector APIs that support *byte* as input and provide *byte* as output, then data corruption might occur.

Note: If you set custom maximum length for decimal token, then take into account that the actual maximum length of the input value should be 1-2 characters less than custom maximum. This type of token is non-length preserving, and the tokenized value can be 1-2 characters longer than the input value.

The following table shows examples of the way in which a value will be tokenized with the Decimal token.

Table 3-47: Tokenization for Decimal Values

Input Value	Tokenized Value	Comments
519.02	268.68	Input value has (.) dot separator.
-0.333807	-9.893967	Input value has sign and (.) dot separator.
+,461	+,918	Input value has sign and (,) comma separator.
0	1	Minimum length, no sign or separator.

3.4.13 Unicode

The Unicode token type can be used to tokenize multi-byte character strings. The input is treated as a byte stream, hence there are no delimiters. There are also no character conversions or code point validation done on the input. The token value will be alpha-numeric.

Unicode tokenization is supported only by Application Protectors, Big Data Protector and Teradata Database Protector.

Note: z/OS supports Unicode tokenization, from version 6.5.2.

Table 3-48: Unicode Tokenization Type properties for different Protectors

Tokenization Type Properties	Settings
Name	Unicode
Token type and Format	Application protectors support UTF-8, UTF-16LE, and UTF-16BE encoding. Hex character codes from 0x00 to 0xFF. **Note:** For the list of supported characters, refer to

<table>
<tr><th>Tokenization Type Properties</th><th colspan="8">Settings</th></tr>
<tr><td></td><td colspan="8">AppendixA: ASCII Character Codes.</td></tr>
<tr><td>Tokenizer</td><td>Length Preservation</td><td>Allow Short Data*11</td><td colspan="2">Minimum Length*10</td><td colspan="4">Maximum Length12</td></tr>
<tr><td rowspan="3">SLT_1_3 *1

SLT_2_3 *1</td><td rowspan="3">No</td><td>Yes</td><td colspan="2">1 byte</td><td colspan="4" rowspan="3">4096</td></tr>
<tr><td>No, return input as it is</td><td colspan="2" rowspan="2">3 bytes</td></tr>
<tr><td>No, generate error</td></tr>
<tr><td>Possibility to set Minimum/ maximum length</td><td colspan="8">No</td></tr>
<tr><td>Left/Right settings</td><td colspan="8">No</td></tr>
<tr><td>Internal IV</td><td colspan="8">No</td></tr>
<tr><td>External IV</td><td colspan="8">Yes

Note: The Database Protectors do not support External IV.</td></tr>
<tr><td rowspan="2">Supported input data types (by Application Protectors)*3</td><td>AP Python *9</td><td>AP Java*9</td><td>AP NodeJS*9</td><td>AP C*9</td><td colspan="2">AP .Net*9</td><td colspan="2">AP Go*9</td></tr>
<tr><td>BYTES

STRING</td><td>BYTE[]

CHAR[]

STRING</td><td>STRING

BYTE[]</td><td>BYTE[]</td><td colspan="2">STRING

BYTE[]</td><td colspan="2">[]BYTE

STRING</td></tr>
<tr><td rowspan="2">Supported input data types (by DB Protectors)</td><td colspan="3">MSSQL Server</td><td colspan="3">Oracle</td><td colspan="3">DB/2</td></tr>
<tr><td colspan="3">NVARCHAR</td><td colspan="3">VARCHAR2</td><td colspan="3">VARCHAR*4</td></tr>
<tr><td rowspan="2">Supported input data types (by MPP DB Protectors)</td><td colspan="3">Teradata *7</td><td colspan="3">Greenplum</td><td colspan="3">IBM Netezza</td></tr>
<tr><td colspan="3">VARCHAR UNICODE</td><td colspan="3">VARCHAR</td><td colspan="3">Not supported</td></tr>
<tr><td>Supported input data types (for Big Data Protectors)*2</td><td>MapReduce *5</td><td>Hive</td><td>Pig</td><td>HBase *5</td><td>Impala*6</td><td>Spark *5</td><td>Spark SQL</td><td>Presto</td></tr>
</table>

Tokenization Type Properties	Settings							
	BYTE[]	STRING	Not supported	BYTE[]	STRING	BYTE[] STRING	STRING	VARCHAR
Supported input data types (z/OS)	GRAPHIC VARGRAPHIC							
Return of Protected value	Yes							
Supported in Protegrity releases	From release 6.6.0 onwards							
Token specific properties	Tokenization result is Alpha-Numeric.							

Note:

*1 – In z/OS, the maximum input length supported for all protectors is 128.

*2 – The input should be Base64 encoded.

*3 – If the input and output types of the API are BYTE[], then the customer application should convert the input to and output from the byte array, before calling the API.

*4 - The z/OS Database Protector supports Unicode tokens with GRAPHIC and VARGRAPHIC data types and the maximum input length supported is 94.

The unprotect only FIELDPROC, *PTYPFPU*, does not support unprotecting a data that is protected with Unicode tokens.

*5 – The Protegrity MapReduce protector, HBase coprocessor, and Spark protector only support *bytes* converted from the *string* data type. If any other data type is directly converted to *bytes* and passed as input to the MapReduce or Spark API that supports *byte* as input and provides *byte* as output, then data corruption might occur. If any other data type is directly converted to *bytes* and inserted in an HBase table, which is configured with the Protegrity HBase coprocessor, then data corruption might occur.

*6 – Impala is supported from release 6.6.2 onwards.

*7 – If you are configuring Unicode tokens with the Teradata database protector, then ensure that the prerequisites for the same are met.

For more information about the prerequisites, refer to section *Using Unicode Tokens with Teradata* in the *Database Protector Guide 9.1.0.0*.

*8 – For XCClient, Unicode data element supports 1560 bytes.

*9 – The Protegrity Application Protectors only support *bytes* converted from the *string* data type. If *int*, *short*, or *long* format data is directly converted to *bytes* and passed as input to the Application Protector APIs that support *byte* as input and provide *byte* as output, then data corruption might occur.

*10 - If short data tokenization is not enabled, the minimum length for Unicode tokenization type is 3 bytes. The input value in Teradata Unicode UDF is encoded using UTF16 due to which internally the data length is multiplied by 2 bytes. Hence, the Teradata Unicode UDF is able to tokenize a data length that is less than the minimum supported length of 3 bytes.

*11 - The options to enable short data tokenization and to return input as is for Unicode data elements is supported by protectors with version 7.2 or higher.

*12 - The maximum input length to safely tokenize and detokenize the data is 1000 Unicode characters, which is irrespective of the byte representation.

Note:
For z/OS Application Protector and z/OS File Protector, the maximum input length supported is 190 bytes.

Note:
The DB2 Database Protector supports a maximum input length of 4096 bytes for *pty.ins_unicode_varchar*, *pty.upd_unicode_varchar*, and *pty.sel_unicode_varchar* UDFs.

The minimum and maximum lengths supported for the Big Data Protector are as described by the following points:

- **MapReduce**: The maximum limit that can be safely tokenized and detokenized back is 4096 bytes. The user controls the encoding, as required.
- **Spark**: The maximum limit that can be safely tokenized and detokenized back is 4096 bytes. The user controls the encoding, as required.
- **Hive**: The *ptyProtectUnicode* and *ptyUnprotectUnicode* UDFs convert the data to UTF-16LE encoding internally, which has a minimum requirement of four bytes of data (in UTF-16LE encoding), and a maximum limit of 4096 bytes (in UTF-16LE encoding) to safely tokenize and detokenize the data.

  The *pty_ProtectStr* and *pty_UnprotectStr* UDFs convert the data to UTF-8 encoding internally, which has a minimum requirement of three bytes of data (in UTF-8 encoding), and a maximum limit of 4096 bytes (in UTF-8 encoding) to safely tokenize and detokenize the data.

- **Impala**: The *pty_UnicodeStringIns* and *pty_UnicodeStringSel* UDFs convert the data to UTF-16LE encoding internally, which has a minimum requirement of four bytes of data (in UTF-16LE encoding), and a maximum limit of 4096 bytes (in UTF-16LE encoding) to safely tokenize and detokenize the data.

  The *pty_StringIns* and *pty_StringSel* UDFs convert the data to UTF-8 encoding internally, which has a minimum requirement of three bytes of data (in UTF-8 encoding), and a maximum limit of 4096 bytes (in UTF-8 encoding) to safely tokenize and detokenize the data.

For instance, the respective lengths for UTF-8 and UTF-16LE, in bytes, is described in the following table.

Table 3-49: Lengths for UTF-8 and UTF-16LE

Input Value	UTF-8	UTF-16LE
導字社導字會	18 bytes	12 bytes
Protegrity	10 bytes	20 bytes

The following table shows examples of the way in which a value will be tokenized with the Unicode token.

Table 3-50: Tokenization for Unicode Values

Input Value	Tokenized Value	Comments
Протегріті	WurIeXLFZPApXQorkFCKl3hpRaGR28K	Input value contains Cyrillic characters. Tokenization result is Alpha-Numeric.

Input Value	Tokenized Value	Comments
安全	xM2EcAQ0LVtQJ	Input value contains characters in Simplified Chinese. Tokenization result is Alpha-Numeric.
Protegrity	RsbQU8KdcQzHJ1	Algorithm is non-length preserving. Tokenized value is longer than initial one.
a	V2wU	Unicode, Allow Short Data=Yes Algorithm is non-length preserving. Tokenized value is longer than initial one.
a9c	A0767Vo	

3.4.14 Unicode Base64

The Unicode Base64 token type can be used to tokenize multi-byte character strings. The input is treated as a byte stream, hence there are no delimiters. Any character conversions or code point validation are not performed on the input. This token element uses Base64 encoding resulting in better performance compared to Unicode token element as it supports three additional characters, namely *+*, */*, and *=* along with alpha numeric characters. The token value generated includes alpha numeric, *+*, */*, and *=*.

The Unicode Base64 tokenization is supported only by Application Protectors, Big Data Protector, Oracle Database Protector, and Data Security Gateway.

Table 3-51: Unicode Base64 Tokenization Type properties for different Protectors

Tokenization Type Properties	Settings			
Name	Unicode Base64			
Token type and Format	Application protectors support UTF-8 encoding. Hex character codes from 0x00 to 0xFF. For the list of supported characters, refer to *AppendixA: ASCII Character Codes.*			
Tokenizer	**Length Preservation**	**Allow Short Data**	**Minimum Length**	**Maximum Length**[*6]
SLT_1_3 SLT_2_3	No	Yes	1 byte	4096
		No, return input as it is	3 bytes	
		No, generate error		
Possibility to set Minimum/Maximum length	No			
Left/Right settings	No			

<table>
<tr><th>Tokenization Type Properties</th><th colspan="8">Settings</th></tr>
<tr><td>Internal IV</td><td colspan="8">No</td></tr>
<tr><td>External IV</td><td colspan="8">Yes

Note: External IV is not supported in Database Protector.</td></tr>
<tr><td rowspan="2">Supported input data types (by Application Protectors)*1</td><td>AP Python *5</td><td>AP Java*5</td><td>AP NodeJS*5</td><td>AP C*5</td><td>AP .Net*5</td><td>AP Go*5</td><td colspan="2"></td></tr>
<tr><td>BYTES

STRING</td><td>BYTE[]

CHAR[]

STRING</td><td>STRING

BYTE[]</td><td>BYTE[]</td><td>STRING

BYTE[]</td><td>[]BYTE

STRING</td><td colspan="2"></td></tr>
<tr><td rowspan="2">Supported input data types (by DB Protectors)</td><td colspan="2">MSSQL Server</td><td colspan="3">Oracle*7</td><td colspan="3">DB/2*2</td></tr>
<tr><td colspan="2">NVARCHAR</td><td colspan="3">VARCHAR2

NVARCHAR2</td><td colspan="3">VARCHAR*2</td></tr>
<tr><td rowspan="2">Supported input data types (by MPP DB Protectors)</td><td colspan="2">Teradata</td><td colspan="3">GPDB</td><td colspan="3">IBM Netezza</td></tr>
<tr><td colspan="2">VARCHAR UNICODE</td><td colspan="3">Not supported</td><td colspan="3">Not supported</td></tr>
<tr><td rowspan="2">Supported input data types (for Big Data Protectors)</td><td>MapReduce *3</td><td>Hive</td><td>Pig</td><td>HBase *3</td><td>Impala</td><td>Spark *3</td><td>Spark SQL</td><td>Presto</td></tr>
<tr><td>BYTE[]</td><td>STRING</td><td>Not supported</td><td>BYTE[]</td><td>STRING</td><td>BYTE[]

STRING</td><td>STRING</td><td>VARCHAR</td></tr>
<tr><td>Return of Protected value</td><td colspan="8">Yes</td></tr>
<tr><td>Supported in Protegrity releases</td><td colspan="8">9.1.0.0</td></tr>
<tr><td>Token specific properties</td><td colspan="8">Tokenization result is Alpha-Numeric, +, /, and =.</td></tr>
</table>

Note:

*1 – If the input and output types of the API are BYTE[], then the customer application should convert the input to and output from the byte array, before calling the API.

*2 - z/OS supports DB/2 database protector.

*3 – The Protegrity MapReduce protector, HBase coprocessor, and Spark protector only support *bytes* converted from the *string* data type. If any other data type is directly converted to *bytes* and passed as input to the MapReduce or Spark API that supports *byte* as input and

provides *byte* as output, then data corruption might occur. If any other data type is directly converted to *bytes* and inserted in an HBase table, which is configured with the Protegrity HBase coprocessor, then data corruption might occur.

*4 – For XCClient, Unicode data element supports 1560 bytes.

*5 – The Protegrity Application Protectors only support *bytes* converted from the *string* data type. If *int*, *short*, or *long* format data is directly converted to *bytes* and passed as input to the Application Protector APIs that support *byte* as input and provide *byte* as output, then data corruption might occur.

*6 - The maximum input length to safely tokenize and detokenize the data is 4096 bytes, which is irrespective of the byte representation.

*7 - The maximum input lengths supported for the Oracle database protector are as described by the following points:

- **Base 64 – Data type : VARCHAR2:** The maximum limit that can be safely tokenized and detokenized back is 3000 bytes.
- **Base 64 – Data type : NVARCHAR2:** The maximum limit that can be safely tokenized and detokenized back is 3000 bytes.

Note:

The DB2 Database Protector supports a maximum input length of 4096 bytes for *pty.ins_unicode_varchar*, *pty.upd_unicode_varchar*, and *pty.sel_unicode_varchar* UDFs.

The minimum and maximum lengths supported for the Big Data Protector are as described by the following points:

- **MapReduce**: The maximum limit that can be safely tokenized and detokenized back is 4096 bytes. The user controls the encoding, as required.
- **Spark**: The maximum limit that can be safely tokenized and detokenized back is 4096 bytes. The user controls the encoding, as required.
- **Hive**: The *ptyProtectUnicode* and *ptyUnprotectUnicode* UDFs convert the data to UTF-16LE encoding internally, which has a minimum requirement of four bytes of data (in UTF-16LE encoding), and a maximum limit of 4096 bytes (in UTF-16LE encoding) to safely tokenize and detokenize the data.

  The *pty_ProtectStr* and *pty_UnprotectStr* UDFs convert the data to UTF-8 encoding internally, which has a minimum requirement of three bytes of data (in UTF-8 encoding), and a maximum limit of 4096 bytes (in UTF-8 encoding) to safely tokenize and detokenize the data.

- **Impala**: The *pty_UnicodeStringIns* and *pty_UnicodeStringSel* UDFs convert the data to UTF-16LE encoding internally, which has a minimum requirement of four bytes of data (in UTF-16LE encoding), and a maximum limit of 4096 bytes (in UTF-16LE encoding) to safely tokenize and detokenize the data.

  The *pty_StringIns* and *pty_StringSel* UDFs convert the data to UTF-8 encoding internally, which has a minimum requirement of three bytes of data (in UTF-8 encoding), and a maximum limit of 4096 bytes (in UTF-8 encoding) to safely tokenize and detokenize the data.

For instance, the respective lengths for UTF-8 and UTF-16LE, in bytes, is described in the following table.

Table 3-52: Lengths for UTF-8 and UTF-16LE

Input Value	UTF-8	UTF-16LE
導字社導字會	18 bytes	12 bytes
Protegrity	10 bytes	20 bytes

The following table shows examples of the way in which a value will be tokenized with the Unicode Base64 token.

Table 3-53: Tokenization for Unicode Base64 Values

Input Value	Tokenized Value	Comments
Москва	BftgxVX0t+O+I8v	Input value contains Cyrillic characters. Tokenization result include alpha numeric characters, =, and +.
Protegrity	9NHI=znyLfgRiRvD	Algorithm is non-length preserving. Tokenized value is longer than initial one.
aĖ	=+bg	Unicode Base64 token element Algorithm is non-length preserving. Tokenized value is longer than initial one.
P+	+BIN	Unicode Base64 token element, Allow Short Data=Yes Algorithm is non-length preserving. Tokenized value is longer than initial one.

3.4.15 Unicode Gen2

The Unicode Gen2 token type can be used to tokenize multi-byte code point character strings. The input Unicode data after protection returns a token value in the same Unicode character format. The Unicode Gen2 token type gives you the liberty to customize how the protected token value is returned. It allows you to leverage existing internal alphabets or create custom alphabets by defining code points. The Unicode Gen2 token type also supports code point length preservation, thus resulting in the protected token length to be exactly the same as the input data, if the length preservation option is selected.

As the token type provides customizations through defining code points and creating custom token values, there are some considerations that must be taken before using such custom alphabets.

> **Note:** For more information about the considerations, refer to *Code Point Range in Unicode Gen2 Token Type*.

The performance benefits of this token type are higher compared to the other Unicode token types.

Table 3-54: Unicode Gen2 Tokenization Type properties for different Protectors

Tokenization Type Properties	Settings
Name	Unicode Gen2
Token type and Format	Application Protectors support UTF-8, UTF-16LE and UTF-16BE encoding. Code points from U+0020 to U+3FFFF excluding D800-DFFF. **Note:** Encoding supported by the Unicode Gen2 data element is *UTF-8*, *UTF-16LE*, and *UTF-16BE*. 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.

<table>
<tr><th>Tokenization Type Properties</th><th colspan="10">Settings</th></tr>
<tr><td>Tokenizer</td><td>Length Preservation</td><td>Allow Short Data</td><td colspan="2">Minimum Length*6</td><td colspan="6">Maximum Length*7</td></tr>
<tr><td rowspan="3">SLT_1_3 *1

SLT_X_1 *1</td><td rowspan="3">Yes</td><td>Yes</td><td colspan="2">1 Code Point</td><td colspan="6">4096 Code Points</td></tr>
<tr><td>No, return input as it is</td><td colspan="2" rowspan="2">3 Code Points</td><td colspan="6" rowspan="2"></td></tr>
<tr><td>No, generate error</td></tr>
<tr><td>Possibility to set Minimum/ Maximum length</td><td colspan="10">No</td></tr>
<tr><td>Left/Right settings</td><td colspan="10">Yes</td></tr>
<tr><td>Internal IV</td><td colspan="10">No</td></tr>
<tr><td>External IV</td><td colspan="10">Yes

Note: The Database Protectors do not support External IV.</td></tr>
<tr><td rowspan="2">Supported input data types (by Application Protectors)*1</td><td colspan="2">AP Python*5</td><td colspan="2">AP Java*5</td><td>AP NodeJS*5</td><td>AP C*5</td><td>AP .Net*5</td><td colspan="2">AP Go*5</td></tr>
<tr><td colspan="2">BYTES

STRING</td><td colspan="2">BYTE[]

CHAR[]

STRING</td><td>STRING

BYTE[]</td><td>BYTE[]</td><td>STRING

BYTE[]</td><td colspan="2">[]BYTE

STRING</td></tr>
<tr><td rowspan="2">Supported input data types (by DB Protectors)</td><td colspan="3">MSSQL Server</td><td colspan="3">Oracle*7</td><td colspan="4">DB/2*2</td></tr>
<tr><td colspan="3">NVARCHAR</td><td colspan="3">VARCHAR2

NVARCHAR2</td><td colspan="4">VARCHAR*2</td></tr>
<tr><td rowspan="2">Supported input data types (by MPP DB Protectors)</td><td colspan="3">Teradata</td><td colspan="3">GPDB</td><td colspan="4">IBM Netezza</td></tr>
<tr><td colspan="3">VARCHAR UNICODE</td><td colspan="3">Not supported</td><td colspan="4">Not supported</td></tr>
<tr><td rowspan="2">Supported input data types (for Big Data Protectors) *1</td><td>MapReduce *3</td><td>Hive</td><td>Pig</td><td>HBase *3</td><td colspan="2">Impala*5</td><td>Spark *3</td><td>Spark SQL</td><td colspan="2">Presto</td></tr>
<tr><td>BYTE[]</td><td>STRING</td><td>Not supported</td><td>BYTE[]</td><td colspan="2">STRING</td><td>BYTE[]

STRING</td><td>STRING</td><td colspan="2">VARCHAR</td></tr>
</table>

Tokenization Type Properties	Settings
Return of Protected value	Yes
SLT_1_3 tokenizer supported in Protegrity releases	From release 8.1.0.0 onwards
SLT_X_1 tokenizer supported in Protegrity releases	From release 9.1.0.0 onwards
Token specific properties	Result is based on the alphabets selected while creating the token.

Note:

*1 – If the input and output types of the API are BYTE[], then the customer application should convert the input to and output from the byte array, before calling the API.

*2 - z/OS supports DB/2 database protector.

*3 – The Protegrity MapReduce protector, HBase coprocessor, and Spark protector only support *bytes* converted from the *string* data type. If any other data type is directly converted to *bytes* and passed as input to the MapReduce or Spark API that supports *byte* as input and provides *byte* as output, then data corruption might occur. If any other data type is directly converted to *bytes* and inserted in an HBase table, which is configured with the Protegrity HBase coprocessor, then data corruption might occur.

The charset or encoding (UTF-8, UTF-16LE, and UTF-16BE) of input data used for byte conversion should match with the default encoding parameter used for respective data element.

*4 – For XCClient, Unicode data element supports 1560 bytes.

*5 – The Protegrity Application Protectors only support *bytes* converted from the *string* data type. If *int*, *short*, or *long* format data is directly converted to *bytes* and passed as input to the Application Protector APIs that support *byte* as input and provide *byte* as output, then data corruption might occur.

*6 - The maximum input length to safely tokenize and detokenize the data is 4096 bytes, which is irrespective of the byte representation.

*7 - The maximum input lengths supported for the Oracle database protector are as described by the following points:

- **Unicode Gen2 – Data type : VARCHAR2:**
  1. If the tokenizer length preservation parameter is selected as *Yes*, then the maximum limit that can be safely tokenized and detokenized is 4000 bytes.
  2. If the tokenizer length preservation parameter is selected as *No*, then the maximum limit that can be safely tokenized and detokenized is 3000 bytes.
- **Unicode Gen2 – Data type : NVARCHAR2:**
  1. If the tokenizer length preservation parameter is selected as *Yes*, then the maximum limit that can be safely tokenized and detokenized is 4000 bytes.
  2. If the tokenizer length preservation parameter is selected as *No*, then the maximum limit that can be safely tokenized and detokenized is 3000 bytes.
- **Unicode Gen2 - Tokenizers**
  - The Unicode Gen2 data element supports SLT_1_3 and SLT_X_1 tokenizers.
  - The SLT_1_3 tokenizer supports small alphabet size from 10-160 code points.
  - The SLT_X_1 tokenizer supports large alphabet size from 161-100K code points.

Note:

The DB2 Database Protector supports a maximum input length of 4096 chars/16384 bytes for *pty.ins_unicode_varchar*, *pty.upd_unicode_varchar*, and *pty.sel_unicode_varchar* UDFs.

The minimum and maximum lengths supported for the Big Data Protector are as described by the following points:

- **MapReduce**: The maximum limit that can be safely tokenized and detokenized back is 4096 bytes. The user controls the encoding, as required.
- **Spark**: The maximum limit that can be safely tokenized and detokenized back is 4096 bytes. The user controls the encoding, as required.
- **Hive**: The *ptyProtectUnicode* and *ptyUnprotectUnicode* UDFs convert the data to UTF-16LE encoding internally, which has a minimum requirement of four bytes of data (in UTF-16LE encoding), and a maximum limit of 4096 bytes (in UTF-16LE encoding) to safely tokenize and detokenize the data.

  The *pty_ProtectStr* and *pty_UnprotectStr* UDFs convert the data to UTF-8 encoding internally, which has a minimum requirement of three bytes of data (in UTF-8 encoding), and a maximum limit of 4096 bytes (in UTF-8 encoding) to safely tokenize and detokenize the data.

- **Impala**: The *pty_UnicodeStringIns* and *pty_UnicodeStringSel* UDFs convert the data to UTF-16LE encoding internally, which has a minimum requirement of four bytes of data (in UTF-16LE encoding), and a maximum limit of 4096 bytes (in UTF-16LE encoding) to safely tokenize and detokenize the data.

  The *pty_StringIns* and *pty_StringSel* UDFs convert the data to UTF-8 encoding internally, which has a minimum requirement of three bytes of data (in UTF-8 encoding), and a maximum limit of 4096 bytes (in UTF-8 encoding) to safely tokenize and detokenize the data.

For instance, the respective lengths for UTF-8, UTF-16LE, and UTF-16BE in bytes, is described in the following table.

Table 3-55: Lengths for UTF-8, UTF-16LE, and UTF-16BE

Input Value	UTF-8	UTF-16LE	UTF-16BE
導字社導字會	18 bytes	12 bytes	12 bytes
Protegrity	10 bytes	20 bytes	20 bytes

The following table shows examples of the way in which a value will be tokenized with the Unicode Gen2 token.

Table 3-56: Tokenization for Unicode Gen2 Values

Input Value	Tokenized Value	Comments
МОСКВ6	Ъыш4х	Input value contains Cyrillic characters. Tokenization results include Cyrillic characters as the data element is created with the Cyrillic alphabet in its definition. The length of the tokenized value is equal to the length of the input data.
Protegrity	93VbLvI12g	Input contains English characters. As the data element include Basic Latin Alpha Numeric characters, the token element includes the same. Algorithm is length preserving. The length of the tokenized value is equal to the length of the input data.
ЕЖ	ao	Input value contains Cyrillic characters. Tokenization results include Cyrillic characters as the data element is created with the Cyrillic alphabet in its definition. Allow Short Data=Yes Algorithm is length preserving. The length of the tokenized value is equal to the length of the input data.

3.4.15.1 Code Point Range in Unicode Gen2 Token Type

When you define a Unicode Gen2 data element, you have an option to either leverage the existing internal alphabets or create custom alphabets. When creating a custom alphabet, a combination of existing alphabets, individual code points or ranges of code points can be used.

While this feature gives you the flexibility to generate token values in Unicode characters, the data element creation does not validate if the code point is defined or undefined. For example, consider that you create a data element that protects Greek and Coptic Unicode block. Though not recommended, a way you might consider to create the custom alphabet would be using the code point range option to include the whole Unicode block that ranges from ***U+0370*** to ***U+03FF***. As seen from the following image, this range includes both defined and undefined code points.

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
U+037x	Ͱ	ͱ	Ͳ	ͳ	ʹ	͵	Ͷ	ͷ			ͺ	ͻ	ͼ	ͽ	;	Ϳ
U+038x					΄	΅	Ά	·	Έ	Ή	Ί		Ό		Ύ	Ώ
U+039x	ΐ	Α	Β	Γ	Δ	Ε	Ζ	Η	Θ	Ι	Κ	Λ	Μ	Ν	Ξ	Ο
U+03Ax	Π	Ρ		Σ	Τ	Υ	Φ	Χ	Ψ	Ω	Ϊ	Ϋ	ά	έ	ή	ί
U+03Bx	ΰ	α	β	γ	δ	ε	ζ	η	θ	ι	κ	λ	μ	ν	ξ	ο
U+03Cx	π	ρ	ς	σ	τ	υ	φ	χ	ψ	ω	ϊ	ϋ	ό	ύ	ώ	Ϗ
U+03Dx	ϐ	ϑ	ϒ	ϓ	ϔ	ϕ	ϖ	ϗ	Ϙ	ϙ	Ϛ	ϛ	Ϝ	ϝ	Ϟ	ϟ
U+03Ex	Ϡ	ϡ	Ϣ	ϣ	Ϥ	ϥ	Ϧ	ϧ	Ϩ	ϩ	Ϫ	ϫ	Ϭ	ϭ	Ϯ	ϯ
U+03Fx	ϰ	ϱ	ϲ	ϳ	ϴ	ϵ	϶	Ϸ	ϸ	Ϲ	Ϻ	ϻ	ϼ	Ͻ	Ͼ	Ͽ

Figure 3-7: Greek and Coptic Code Points

The code point, ***U+0378*** in the defined Greek and Coptic code point range is an undefined code point. When any input data is protected, since the code point range includes both defined and undefined code points, it might result in a corrupted token value if the entire code point range is defined.

It is hence recommended that for Unicode code point ranges where both defined and undefined code points exist, you must create code points ranges excluding any undefined code points. So, in case of the Greek and Coptic characters, a recommended strategy to define alphabets would be to create multiple alphabet entries, such as a range to cover ***U+0371*** to ***U+0377***, another range to cover ***U+037A*** to ***U+037F***, and so on, thus skipping undefined code points.

Important: Only the alphabet characters that are supported by the OS fonts are displayed on the Web UI.

Note: Ensure that code points in the alphabet are supported by the protectors using this alphabet.

3.4.16 Binary

The Binary token type can be used to tokenize binary data with Hex codes from 0x00 to 0xFF. Binary token is supported only by Application Protectors but not by this Protector in z/OS.

Table 3-57: Binary Tokenization Type properties for different Protectors

Tokenization Type Properties	Settings
Name	Binary

Tokenization Type Properties	Settings							
Token type and Format	Hex character codes from 0x00 to 0xFF.							
Tokenizer	**Length Preservation**	**Minimum Length**	**Maximum Length**					
SLT_1_3 SLT_2_3	No	3	4095					
Possibility to set Minimum/ maximum length	No							
Left/Right settings	Yes							
Internal IV	Yes, if Left/Right settings are non-zero.							
External IV	Yes							
Supported input data types (by Application Protectors) *1	**AP Python** *4	**AP Java**[*4]	**AP NodeJS**[*4]	**AP C**[*4]	**AP .Net**[*4]	**AP Go**[*4]		
	BYTES	BYTE[]	BYTE[]	BYTE[]	BYTE[]	[]BYTE		
Supported input data types (by DBs)	**MSSQL Server**	**Oracle**	**DB/2**					
	Not supported	Not supported	Not supported					
Supported input data types (by MPP DBs)	**Teradata**	**GPDB**	**IBM Netezza**					
	Not supported	Not supported	Not supported					
Supported input data types (for Big Data Protectors) *1	**MapReduce** [*2]	**Hive**	**Pig**	**HBase** [*2]	**Impala**	**Spark** [*2]	**Spark SQL**	**Presto**
	BYTE[]	Not supported	Not supported	BYTE[]	Not supported	BYTE[] *3	Not supported	Not supported
Return of Protected value	No							
Supported in Protegrity releases	6.6.x – 9.x.x.x							
Token specific properties	Tokenization result is binary.							

Note:

*1 – If the input and output types of the API are BYTE[], then the customer application should convert the input to and output from the byte array, before calling the API.

*2 – The Protegrity MapReduce protector, HBase coprocessor, and Spark protector only support *bytes* converted from the *string* data type. If any other data type is directly converted to *bytes* and passed as input to the MapReduce or Spark API that supports *byte* as input and provides *byte* as output, then data corruption might occur. If any other data type is directly converted to *bytes* and inserted in an HBase table, which is configured with the Protegrity HBase coprocessor, then data corruption might occur.

*3 – It is recommended to use Binary tokenization only with APIs that accept BYTE[] as input and provide BYTE[] as output. If Binary tokens are generated using APIs that accept BYTE[] as input and provide BYTE[] as output, and uniform encoding is maintained across protectors, then the tokens can be used across various protectors.

*4 – The Protegrity Application Protectors only support *bytes* converted from the *string* data type. If *int*, *short*, or *long* format data is directly converted to *bytes* and passed as input to the Application Protector APIs that support *byte* as input and provide *byte* as output, then data corruption might occur.

The following table shows examples of the way in which a value will be tokenized with the Binary token.

Table 3-58: Tokenization for Binary Values

Input Value	Tokenized Value	Comments
Protegrity	0x05C1CF0C310B2D38ACAD4C	Tokenization result is returned as a binary stream.
123	0x19707E	Tokenization of the value with Minimum supported length.

3.4.17 Email

Email token type allows tokenization of an email address. Email token keeps the domain name along with all characters before the “@” sign, and the “@” sign itself in the clear. The local part (i.e. the part before “@”) gets tokenized.

Table 3-59: Email Tokenization Type Properties different Protectors

Tokenization Type Properties	Settings						
Name	Email						
Token type and Format	Alphabetic and numeric only. The rest characters will be treated as delimiters.						
Tokenizer*3	**Length Preservation**	**Minimum Length**			**Maximum Length**		
		Local	Domain	Entire	**Local**	Domain	Entire
SLT_1_3	No	**1**	1	3	**63**	252	256
SLT_2_3	No	**1**	1	3	**63**	252	256
SLT_1_3	Yes	**3***5	1	5	**64**	252*6	256
SLT_2_3	Yes	**3***5	1	5	**64**	252*6	256
Possibility to set minimum/ maximum length	No						

<table>
<tr><th>Tokenization Type Properties</th><th colspan="9">Settings</th></tr>
<tr><td>Left/Right settings</td><td colspan="9">No</td></tr>
<tr><td>Internal IV</td><td colspan="9">N/A</td></tr>
<tr><td>External IV</td><td colspan="9">Yes</td></tr>
<tr><td rowspan="2">Supported input data types (by Application Protectors) *1</td><td colspan="2">AP Python *4</td><td colspan="2">AP Java *4</td><td>AP NodeJS*4</td><td>AP C*4</td><td>AP .Net*4</td><td colspan="2">AP Go*4</td></tr>
<tr><td colspan="2">STRING

BYTES</td><td colspan="2">STRING

CHAR[]

BYTE[]</td><td>STRING

BYTE[]</td><td>BYTE[]</td><td>STRING

BYTE[]</td><td colspan="2">STRING

[]BYTE</td></tr>
<tr><td rowspan="2">Supported input data types (by DB Protectors)</td><td colspan="3">MSSQL Server</td><td colspan="3">Oracle</td><td colspan="3">DB/2</td></tr>
<tr><td colspan="3">VARCHAR

CHAR</td><td colspan="3">VARCHAR2

CHAR</td><td colspan="3">VARCHAR CHAR</td></tr>
<tr><td rowspan="2">Supported input data types (by MPP DB Protectors)</td><td colspan="3">Teradata</td><td colspan="3">GPDB</td><td colspan="3">IBM Netezza</td></tr>
<tr><td colspan="3">VARCHAR LATIN

CHAR LATIN</td><td colspan="3">VARCHAR</td><td colspan="3">VARCHAR</td></tr>
<tr><td rowspan="2">Supported input data types (for Big Data Protectors) *1</td><td>MapReduce *3</td><td>Hive</td><td>Pig</td><td>HBase *3</td><td>Impala</td><td>Spark *3</td><td>Spark SQL</td><td>Presto</td></tr>
<tr><td>BYTE[]</td><td>CHAR*7

STRING</td><td>CHARARRAY</td><td>BYTE[]</td><td>STRING</td><td>BYTE[]

STRING</td><td>STRING</td><td>VARCHAR</td></tr>
<tr><td>Return of Protected value</td><td colspan="9">Yes</td></tr>
<tr><td>Supported in Protegrity releases</td><td colspan="9">6.6.x – 9.x.x.x</td></tr>
<tr><td>Token specific properties</td><td colspan="9">At least one @ character

The right most @ character defines the delimiter between the local and domain parts</td></tr>
</table>

Note:

*1 – If the input and output types of the API are BYTE[], then the customer application should convert the input to and output from the byte array, before calling the API.

*2 – The table lists minimum and maximum length requirements for this token type, which should be applied for the local part, domain part and the entire e-mail.

*3 – The Protegrity MapReduce protector, HBase coprocessor, and Spark protector only support *bytes* converted from the *string* data type. If any other data type is directly converted to *bytes* and passed as input to the MapReduce or Spark API that supports *byte* as input and provides *byte* as output, then data corruption might occur. If any other data type is directly converted to *bytes* and inserted in an HBase table, which is configured with the Protegrity HBase coprocessor, then data corruption might occur.

*4 – The Protegrity Application Protectors only support *bytes* converted from the *string* data type. If *int*, *short*, or *long* format data is directly converted to *bytes* and passed as input to the Application Protector APIs that support *byte* as input and provide *byte* as output, then data corruption might occur.

*5 – If the settings for short data tokenization is set to *Yes*, then the minimum tokenizable length for the local part of an email is one else it is three.

*6 – If the settings for short data tokenization is set to *Yes*, then the maximum length for the domain part of an email is 253 else it is 252.

*7 – If you are using the *Char* tokenization UDFs in Hive, then ensure that the data elements have length preservation selected. In *Char* tokenization UDFs, using data elements without length preservation selected, is not supported.

Note:

For Non-US, when using *Email* token, the From Codepage (FRCODEPG) parameter in PTYPARM file has to be set to the current system codepage by the system administrator. This is valid for z/OS customers only.

3.4.17.1 Email Token Format

The email address consists of a local part and a domain (*local-part@domain*). The local part can be up to 64 characters and the domain name can be up to 254 characters, but the entire email address cannot be longer than 256 characters.

The following table explains e-mail token format input requirements and tokenized output format:

Table 3-60: Output Values for Email Token Format

Local Part **Input value can consist…**	**Output value can consist…**
Commonly used: • Uppercase and lower case characters through a-z/A-Z • Digits 0-9 • Special characters !#$%&'*+-/=?^_`|}{~ (ASCII: 33, 35-39, 42, 43, 45, 47, 61, 63, 94-96, 123-126) • Comments are allowed with parentheses. Used with restrictions: • dot character (“.”) when it is not the first or the last and it does not appear more than one time consecutively • Special characters (ASCII: 32, 34, 40, 41, 44, 58, 59, 60, 62, 64, 91-93) are allowed with restrictions. They must only be used when contained between quotation marks, and that the three of them (the space (32), backslash (92), and quotation mark (34)) must also be preceded by a backslash (for example, "\ \\\"").	The part before “@” sign will be tokenized. The following will be tokenized: • All valid characters will be tokenized by the same rules as alpha-numeric token • Comments will be tokenized. The following characters will be considered as delimiters and not tokenized: • “.” dot character • “()” left and right parenthesis • Special characters in local part.

Local Part **Input value can consist…**	**Output value can consist…**
• International characters above U+007F are permitted by RFC 6531, though mail systems may restrict which characters to use when assigning local parts.	
@ Part "@" character defines the delimiter between the local and domain parts, and will be left in clear.	
Domain Part *Input value can consist…*	**Output value can consist…**
• Letters and digits • Hyphens and dots • **IP address** within square brackets (for example, *john.smith@[1.1.1.1]*) • Non-ASCII domain (internationalized domain parts) • Comments are allowed within parentheses	The part after "@" sign will not be tokenized.

Note:
Comments are allowed both in local and domain part of the e-mail token, and comments will be tokenized only if they are in the local part. Here are the example of comments usage for the e-mail - *john.smith@example.com*:

- john.smith(comment)@example.com
- "john(comment).smith@example.com"
- john(comment)n.smith@example.com
- john.smith@(comment)example.com
- john.smith@example.com(comment)

The following table shows examples of the way in which a value will be tokenized with the Email token.

Table 3-61: Tokenization for Email Token Formats

Input Value	Tokenized Value	Comments
Protegrity1234@gmail.com	UNfOxcZ51jWbXMq@gmail.com	All characters before @ symbol are tokenized
john.smith!@#@$%$ %^&@gmail.com	hX3p.yDcwD!@#@$%$ %^&@gmail.com	All symbols except alphabetic are distinguish as delimiters
email@protegrity@gmail.com	F00CJ@RjDEX9LMDq@gmail.com	The right most @ character defines the delimter between the local and domain parts
q@a	asj@a	Min 3 symbols in local part for none length preserving tokens
qdd@a	S0Y@a	Min 5 symbols in local part for length preserving tokens

Input Value	Tokenized Value	Comments
a@protegrity.com	o@protegrity.com	Email, SLT_1_3, Length Preservation=Yes, Allow Short Data=Yes The local part of the email has at least one character to tokenize, which meets the minimum length requirement for SLT_1_3 tokenizer when Length Preservation=Yes and Allow Short Data=Yes.
a@protegrity.com email@protegrity.com	a@protegrity.com F00CJ@protegrity.com	Email, SLT_1_3, Length Preservation=Yes, Allow Short Data=No, return input as it is If the input value has less than three characters to tokenize, then it is returned as is else it is tokenized.
a@protegrity.com	Error. Input too short.	Email, SLT_1_3, Length Preservation=Yes, Allow Short Data=No, generate an error The local part of the email has one character to tokenize, which is short for SLT_1_3 tokenizer when Length Preservation=Yes and Allow Short Data=No, generate an error.

Chapter 4

Protegrity Format Preserving Encryption

The Protegrity Format Preserving Encryption (FPE) encrypts input data of specified format and generates output data (ciphertext) of the same format. The input data is encrypted using a block cipher method of encryption, in which a cryptographic key and algorithm are applied to a block of data at once, rather than one bit at a time in a typical encryption process. For example, using FPE, a 16-digit credit card number is encrypted such that the generated ciphertext is another 16-digit number. Since encrypted data retains its original format with FPE, there is no need for any schema related changes to the database or application.

Protegrity supports Format Preserving Encryption (FPE) using NIST-approved FF1 (Format preserving, Feistel based, type 1) mode of operation with AES-256 block cipher encryption algorithm. The input block is divided into two halves and pushed through a Feistel network for several rounds. If a tweak is mentioned for the encryption, then those digits are skipped and the remaining digits are passed to the encryption function that encrypts to different values for each round. For FF1, the minimum proposed rounds by NIST are 10.

4.1 FPE Properties

The FPE properties are specified when creating a data element with FPE method. The following table describes the properties provided by FPE.

Table 4-1: FPE Properties

FPE Property	Description
User configured FPE properties	
Name	Unique name that identifies the FPE data element.
Method	FPE NIST 800-38G NIST 800-38G is the recommended FPE specification by NIST that identifies the supported FPE cipher.
Plaintext Alphabet	Plaintext alphabet type that is to be encrypted. Data Type supported for encryption from the following options:

FPE Property	Description
	• Numeric • Alpha • Alpha-Numeric • Unicode Basic Latin and Latin-1 Supplement Alpha • Unicode Basic Latin and Latin-1 Supplement Alpha-Numeric The plaintext alphabet maps to code points that denotes a range of accepted characters. For more information about code point mappings, refer to the section *Code points*.
Plaintext Encoding	Plaintext encoding type that is to be used. The following are the supported plaintext encodings: • ASCII • UTF-8 • UTF-16LE • UTF-16BE For Unicode Basic Latin and Latin-1 Supplement blocks, ASCII encoding is not supported. **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.
Minimum Input Length	The minimum supported input length is 2 bytes and configurable up to 10 bytes. The default minimum supported input length for Credit Card Number (CCN) is 8 bytes and configurable up to 10 bytes.
Tweak Input Mode	Tweak input can be derived from the following options: • Extract from input message • API Argument For more information about tweak input mode, refer to section *Tweak Input*.
From Left	Number of characters from left to retain in clear.
From Right	Number of characters from right to retain in clear. For more information about left and right settings, refer to section *Left and Right Settings*.
Allow Short Data	Whether short data is supported or not (Possible options are *No, generate error*, or *No, return input as it is*). This is supported by Numeric and Alpha-Numeric data types only. FPE does not support data less than 2 bytes, but you can set the minimum input length value accordingly. For more information about short data support, refer to section *Length Preserving*.
Special numeric alphabet handling	Specific request for numeric data type with following options:

FPE Property	Description
	• None • Credit Card Number (CCN) For more information about numeric alphabet handling, refer to section *Handling Special Numeric Data*.
Read-only FPE properties	
Ciphertext Alphabet	Ciphertext alphabet type that is to be derived. This value is same as the *Plaintext Alphabet* value.
Ciphertext Encoding	Plaintext encoding type that is used. This value is same as the *Plaintext Encoding* that is used.
Key Input	Internally Generated
FPE Mode	Mode of operation for the block cipher algorithm with FF1 as the supported mode.
Pseudorandom Function (PRF)	Block cipher algorithm that is used for encryption with AES-256 as the supported algorithm.
Feistel Rounds	10
Max tweak length	The maximum supported tweak input length is 256 bytes.
Support Delimiters	Any input other than the supported data type is treated as a delimiter. If the input contains only delimiters, then the output value is equal to the input. By default, delimiters are supported for Numeric and Alpha-Numeric data type. Credit Card Number (CCN) data type doesn't support delimiters.
Preserve Length	The length preservation setting is true for: • Numeric • Alpha • Alpha-Numeric • Unicode Basic Latin and Latin-1 Supplement Alpha • Unicode Basic Latin and Latin-1 Supplement Alpha-Numeric The length preservation setting is false for CCN.
Other FPE properties	
Maximum Input Length (including delimiters)	The maximum input length for the supported data types is as follows: • Numeric – 2 GB • Alpha – 2 GB • Alpha-Numeric – 2 GB • Unicode Basic Latin and Latin-1 Supplement Alpha – 2 GB • Unicode Basic Latin and Latin-1 Supplement Alpha-Numeric – 2 GB • Credit Card – 4096 bytes **Note:**

FPE Property	Description
	The recommended maximum input size for the FPE data elements is 4096 characters.

Note:

- The maximum supported input length differs for different protectors based on the input length supported by the protector.
- The maximum input length supported by the *PTY.INS_UNICODENVARCHAR2* UDF for the Oracle Database Protectors is 2000 characters.
- The maximum input length supported by the *PTY_FPEUNICODEVARCHARINS* UDF for the Greenplum Database Protector is 2752 bytes.
- If you are using Format Preserving Encryption (FPE) with Teradata UDFs, you can extend the maximum data length size provided by these UDFs, which is up to 47407 bytes by default. In this case, the maximum data length size to be allocated for the UDFs can be modified in the createobjects.sql file for the following functions:
  - PTY_VARCHARLATININS
  - PTY_VARCHARLATINSEL
  - PTY_VARCHARLATINSELEX

  The REPLACE_UDFVARCHARTOKENMAX parameter value for these functions can be set up to 64000. Teradata supports the maximum row size length of approximately 64000 bytes.

  For more information, refer to the section Installing UDFs for Teradata in the *Installation Guide 9.1.0.0*.
- Masking is not supported for data elements created with FPE method and Unicode, Unicode Base64, and Unicode Gen2 as the plaintext alphabet.
- For FPE data elements, External IV is supported only with ASCII Encoding.
- For more information about empty string and NULL handling by protectors, refer to section *Appendix C: Empty String Handling by Protectors* and *Appendix D: NULL Handling by Protectors*.

Table 4-2: Examples for Format Preserving Encryption

Input Value	Encrypted Value	Comments
123456789012345	187868154999435	Plaintext alphabet – Numeric, Plaintext Encoding – ASCII, Tweak Input – Extract from Input Message, Left=1, Right=1, Allow Short Data = No, return input as it is, Minimum Input Length=3
Protegrity1234567	PyNqSJybYp1234567	Plaintext alphabet – Alpha, Plaintext Encoding – UTF8, Tweak Input – API Argument, Left=1, Right=0,

Input Value	Encrypted Value	Comments
		Allow Short Data = No, generate error, Minimum Input Length=2
Protegrity1234567	ProZSNbyADNoPb2ns	Plaintext alphabet – Alpha-Numeric, Plaintext Encoding – UTF16LE, Tweak Input – Extract from Input Message, Left=3, Right=0, Allow Short Data = No, return input as it is, Minimum Input Length=10
43211234567890	76454340562108	Plaintext alphabet – CCN, Plaintext Encoding – ASCII, Tweak Input – Extract from Input Message, Left=0, Right=0, Allow Short Data = No, generate error, Minimum Input Length=9, Invalid Card Type=True
þrõtégrîtÝ@123456789	þràñTÿwõùÞ@123456789	Plaintext alphabet – Unicode Basic Latin and Latin1 Supplement Alpha, Plaintext Encoding – UTF16LE, Tweak Input – Extract from Input Message, Left=2, Right=1, Allow Short Data = No, generate error, Minimum Input Length=4
þrõtégrîtÝ@123456789	þrWtçjÑHÿÖ@9íKLksvp9	Plaintext alphabet – Unicode Basic Latin and Latin1 Supplement Alpha-Numeric, Plaintext Encoding – UTF16BE, Tweak Input – API Argument, Left=2, Right=1,

Input Value	Encrypted Value	Comments
		Allow Short Data = No, return input as it is, Minimum Input Length=6

4.2 Code Points

The code points are coded character sets, where each character maps to unique numeric values for representation of that character. The Unicode Standard is a character encoding system that supports the processing and representation of the text from diverse languages. The various character encoding schemes, such as UTF-8, UTF-16, etc. accept the character code points as input and using pre-defined formulas, generates the encoded numeric value. The Unicode code space comprises of 17 planes, that is, the basic multilingual plane (BMP) and 16 supplementary planes. The BMP contains the most commonly used characters. FPE supports encryption for BMP with Basic Latin (ASCII) and Latin-1 supplement blocks of characters.

For more information about the Unicode Standard and code points, refer to *http://www.unicode.org/* and *http://www.unicode.org/charts/* respectively.

The following table represents the Unicode code points for FPE-supported plaintext alphabet types and encodings.

Table 4-3: Unicode Code Points for FPE-supported Plaintext Alphabet Types

Plaintext Alphabet	Plaintext Encoding	Codepoint range
Numeric	ASCII UTF-8 UTF-16LE UTF-16BE	U+0030 - U+0039
Alpha	ASCII UTF-8 UTF-16LE UTF-16BE	U+0041 - U+005A U+0061 - U+007A
Alpha-Numeric	ASCII UTF-8 UTF-16LE UTF-16BE	U+0030 - U+0039 U+0041 - U+005A U+0061 - U+007A
Unicode Basic Latin and Latin-1 Supplement Alpha	UTF-8 UTF-16LE	U+0041 - U+005A U+0061 - U+007A

Plaintext Alphabet	Plaintext Encoding	Codepoint range
	UTF-16BE	U+00C0 - U+00FF (excluding U+00D7 and U+00F7)
Unicode Basic Latin and Latin-1 Supplement Alpha-Numeric	UTF-8 UTF-16LE UTF-16BE	U+0030 - U+0039 U+0041 - U+005A U+0061 - U+007A U+00C0 - U+00FF (excluding U+00D7 and U+00F7)

4.3 Tweak Input

The tweak input is derived through either of the following methods:

- Extract from input message - If the tweak is set to be derived from input message, then the left and right property settings are used as a configurable tweak option.
- API argument - If the tweak is set to be derived through API argument, then the tweak value is provided as an input parameter through the API during the protect/unprotect operation.

  The resultant tweak input is zero for the following conditions:

  - When extracting the tweak from input message, the left and right property settings are not set or set to zero.
  - When tweak input is to be derived as an API argument, the tweak input parameter is not specified.

> **Note:** The maximum supported tweak input length is 256 bytes.

4.4 Left and Right Settings

This property indicates the number of characters from left and right that will remain in the clear and are excluded from format preserving encryption.

> **Note:** The sum of left and right properties supports a maximum of 0 through 99 characters. For Unicode multibyte character encoding, the initial 256 bytes are considered as the tweak input.

4.5 Handling Special Numeric Data

The Format Preserving Encryption(FPE) for Credit Card Number (CCN) is handled by configuring numeric data type as the plaintext alphabet. The following default settings for CCN are applicable:

- Credit Card Number (CCN) data type doesn’t support delimiters. Hence, the length preservation is false for CCN.
- Short Data Tokenization is not supported by CCN. CCN supports a minimum input length of 8 bytes.

  For more information about Invalid Card Type (ICT), Invalid Luhn, and Alphabet Indicator validation for CCN, refer to section *Credit Card*.

4.6 Encryption Algorithm

The Protegrity FPE currently supports encryption using AES-256 block cipher algorithm.

For more information about the AES-256 algorithm, refer to section *AES-256*.

Chapter 5

Protegrity Encryption

Encryption is the conversion of data into a ciphertext using an algorithmic scheme. The Protegrity solutions can encode data with the following encryption algorithms:

- **Database and Application Protectors** - Encryption (3DES, AES-128, AES-256, CUSP)
- **Big Data Protectors** - Encryption (3DES, AES-128, AES-256). For more information, refer to the table *Supported Input Data Types by Big Data*.
- **HDFSFP (included in Big Data Protector)** - Encryption (3DES, AES-128, AES-256)
- **File Protectors** - Encryption (3DES, AES-128, AES-256)

> **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.

Encryption algorithms vary by input and output data types they support. Some preserve length, while others do not. Also, the minimum allowable length varies between algorithms.

Table 5-1: Encryption Algorithms by supported Length and Output types

Encryption Algorithm	Output	Preserves Length	Minimum Length	Maximum Length
3DES	Binary		None	Depends on Protector type. For Database protectors depends on Database and data types.
AES-128	Binary		None	
AES-256	Binary		None	
CUSP 3DES, CUSP AES-128,	Binary	Yes	None	

Encryption Algorithm	Output	Preserves Length	Minimum Length	Maximum Length
CUSP AES-256		(No if CRC/Key Id are used)		

For Database protection various data types can be used, depending on encryption algorithm and database. The following table illustrates supported data types by databases.

Table 5-2: Input Data Types Supported by Application Protectors

Encryption Algorithm *1	Application Protector					
	AP Python*3	AP Java	AP C	AP NodeJS	AP .Net	AP Go
3DES AES-128 AES-256 CUSP 3DES CUSP AES-128 CUSP AES-256	STRING BYTES INT LONG FLOAT	string *2 char[] *2 byte[] *2	byte[]	BYTE[]	BYTE[]	BYTE[]

Note:

*1 – If the input and output types of the API are BYTE[], then the customer application should convert the input to and output from the byte array, before calling the API.

*2 – The output type is BYTE[] only. The input type String or Char is supported with the API that provides BYTE[] output type.

*3 – You must pass the *encrypt_to=bytes* keyword argument to the AP Python *protect* API for encrypting the data. However, if you are encrypting or re-encrypting BYTES data, then you do not need to pass the *encrypt_to=bytes* keyword argument to the *protect* and *reprotect* API respectively.

Table 5-3: Input Data Types Supported by Database Protectors

Encryption Algorithm	Database		
	MSSQL Server	Oracle	DB/2
3DES AES-128 AES-256 CUSP 3DES	varchar varbinary binary char	varchar2 char number real	integer varchar2 char number

Encryption Algorithm	Database		
	MSSQL Server	**Oracle**	**DB/2**
CUSP AES-128 CUSP AES-256	numeric real float decimal int bigint smallint tinyint datetime bit nvarchar nchar money smallmoney smalldatetime uniqueidentifier	float date raw blob clob	real float date timestamp long varchar double varchar numeric decimal int bigint smallint tinyint datetime character time blob clob

Table 5-4: Input Data Types Supported by MPP Database Protectors

Encryption Algorithm	Database		
	Pivotal GPDB	**Teradata**	**IBM Netezza**
DES AES-128 AES-256 CUSP 3DES CUSP AES-128	varchar integer decimal date	varchar char integer float decimal	varchar int real date

Encryption Algorithm	Database		
	Pivotal GPDB	Teradata	IBM Netezza
CUSP AES-256		date bigint	

Table 5-5: Input Data Types Supported by Big Data Protectors

Encryption Algorithm *1, *2	Big Data						
	MapReduce	Hive	Pig	HBase	Impala	Spark	Spark SQL
3DES AES-128 AES-256 CUSP 3DES CUSP AES-128 CUSP AES-256	BYTE[]	STRING *5	Not supported	BYTE[]	STRING INT FLOAT DOUBLE	BYTE[] STRING *3	Not supported

Note:

*2 - The customer application should convert the input to and output from byte array.

*3 - The input type STRING is supported with the API that provides the BYTE[] output type.

*5 - The string encryption UDFs for Hive 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.

Note:

- Maximum length for Varchar2 data type supported by Protegrity is as follows for these cases:
  - VARCHAR2 (1991) – AES without IV and CRC
  - VARCHAR2 (1963) – AES with IV and CRC
  - VARCHAR2 (1999) – 3DES without IV, CRC and KID
  - VARCHAR2 (1979) – 3DES with IV, CRC and KID
- The DB2 database protector, supports a maximum input data of 64 KB for BLOB UDFs and 5 KB for CLOB UDFs.
- In DB2, BLOB UDFs are incompatible with videographic file formats.
- When encrypting the Char columns on MS SQL Server, they are converted into Varbinary having limitation up to 2GB of data for each column.
- For Teradata, maximum length is 64000 for Varchar Latin and 32000 for Varchar Unicode data types.

5.1 Encryption Properties (IV, CRC, Key ID)

For Structured Data (Database protection and Application protection) security policy types, (with the exception of No Encryption and Hashing) you can specify additional data encryption properties, such as Initialization Vector (IV), Integrity Check (CRC) and Key ID.

For Unstructured Data (File Protection) security policy type, you can specify the Key ID property.

The following table describes encryption properties.

Table 5-6: Encryption Properties

Feature	Description
Initialization Vector (IV)	A block of bits required to allow a cipher to be executed in any of several streaming modes of operation to produce a unique stream, independent from other streams produced by the same encryption key, without having to go through a (usually lengthy) re-keying process. The size of the IV depends on the encryption algorithm and on the cryptographic protocol in use and is normally as large as the block size of the cipher or as large as the encryption key. The IV must be known to the recipient of the encrypted information to be able to decrypt it. Encrypting the same value with the IV property will result in different crypto text for the same value.
Integrity Check (CRC)	A type of function that takes as input a data stream of any length and produces as output a value of a certain fixed size. A CRC can be used as a checksum to detect alteration of data during transmission or storage.
Key ID	Identifier that associates encrypted data with the protection method so that the data can be decrypted regardless of where it ultimately resides. A data element can have multiple instances of key IDs associated with it. For more information, refer to the following Key IDs section.

5.1.1 Key IDs

Data elements can have key IDs associated with them. Key IDs are a way to correlate a data element with its encrypted data. They facilitate these tasks related to the management of sensitive data: archiving, data movement, and key rotation.

> **Note:** Key IDs can only be used with data elements that use AES, 3DES, or CUSP algorithms for Database and Application protection, and AES, 3DES for File protection. In Database and Application protection, Key ID becomes part of the encrypted value (for details on cipher text format, refer *Data Length and Padding in Encryption*). In File protection, Key ID is separated from the encrypted data and stored in meta info of the encrypted file.

Key IDs make cryptographic text portable, which means that no matter where the encrypted data ultimately resides (such as in an archive), it retains the reference back to ESA. With key IDs, the ESA system will always know what protection methods were applied, so the data can be decrypted.

A data element defined to use key IDs can have multiple instances of keys associated with it. The system retains all key IDs and distinguishes them by their numerical identifier and state. Key IDs remain in the system and cannot be deleted so that no matter what happens to the encrypted data, it can be decrypted by way of its key ID.

The following table describes the key ID states.

Table 5-7: Key ID States

Feature	Description
Pre-Active	The initial state of a key that is created by the Create Key option.
Active	A key becomes Active once it is distributed to a PEP server by deploying the data security policy.
Deactivated	An Active key becomes automatically Deactivated when the data security policy is redeployed with a new Pre-Active key.

For more information about key ID property creation and management, refer to *Protegrity Enterprise Security Administrator Guide 9.1.0.0*.

Table 5-8: Examples of Encryption Properties for AES-256 algorithm (initial value is "Protegrity")

Encryption Property	Encrypted Values	Comments
AES-256-IV	0x1361D69E18A692507895780C2FB26DD7869979CC1BB6612A994B5EA5585FCF0B 0xE2D579E937EE92C67167749151B30809A538CC6A6871B8D9B0C17FBA6F1A8D94	Encrypting the same value with the IV property resulted in different output values. Decrypt will be performed correctly for both values.
AES-256-CRC	0x7A0C701B4B30E6BF141196FE44F125BD 0x3964DD0ACAF5B39D159BE7518B46D84A8DCC0B62F2183B3888FEF82B65C7F87D	The first value is a result of encryption 'Protegrity1'. The second value is a result of encryption 'Protegrity12'. Length of the output value is extended twice though initial value increased only in 1 character.
AES-256-KeyID	0x200936F85C3BD86F008A57C3DF33F200BC42 0x20157C0E98A1C9E4E6F4D1DCB6FE72B2DA69	Key ID of the first value equals to 9 (0x2009 in HEX), key ID of the second value equals to 21 (0x2015 in HEX)

5.2 Data Length and Padding in Encryption

Cipher text produced by the Database and Application Protectors are formatted in a specific way depending on which encryption properties are being used.

Encryption algorithms that operate on blocks of data (block ciphers) require padding. The block size for AES is 16 bytes, and for DES/3DES it is 8 bytes. The input is always padded even if it's already a multiple of the block size. Padding is done in such way that the input data together with the checksum (if such setting is turned on) will be equal to the algorithm's block size.

If the initialization vector (IV) property is turned on, then the system generates an IV value that is used to seed the input before it is encrypted. The IV is then appended to the cipher text.

When the Key ID property is turned on there will be an extra 2 bytes in the beginning of the cipher text. This piece of information contains the reference to the Key ID that was used to produce the cipher text.

5.2.1 Ciphertext Format

The length of encrypted value for non-length-preserving encryption (3DES, AES-128, AES-256) depends on block size of initial data, and encryption properties used (KeyID, CRC, IV).

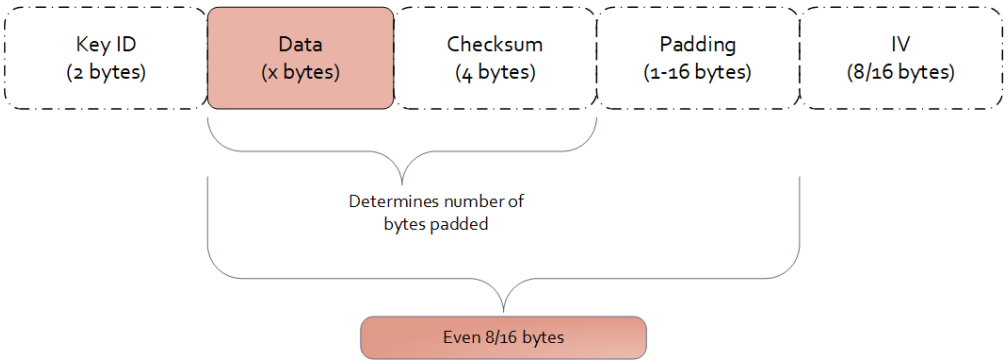

Figure 5-1: Ciphertext format

Examples of data length calculation by column types are provided in *Appendix B: Examples of Column Sizes Calculation for Encryption*

5.3 Encryption Algorithms

The Database, Application, and Big Data Protectors can use 3DES, AES-128, AES-256, and CUSP encryption algorithms. For more information about data types supported by Big Data, refer to the table *Supported Input Data Types by Big Data*.

The 3DES, AES-128, and AES-256 encryption algorithms can also be used with File Protectors and HDFSFP.

> **Note:**
> The Maximum Length that can be encrypted by the different algorithms is mentioned as 2147483610 bytes in the following tables in section *3DES* and section *AES-128 and AES-256*.
>
> Note that this length varies from protector to protector and depends on database configuration, database limitation, and the protection method used.

5.3.1 3DES

The 3DES (Triple Data Encryption Standard) algorithm applies the DES algorithm, the first USA national standard of block ciphering, three times to each data block. The 3DES cipher key size is 168 bits, compared to 56 bits key of DES. The 3DES algorithm, using the DES cipher algorithm, provides a simple method of data protection.

> **Note:** It is recommended to install the IBM z/OS hotfix PI32471 for DB2 before using the FIELDPROC module with non-CUSP algorithms.

Table 5-9: 3DES Encryption Algorithm Properties

Properties	
Name	**3DES**
Operation Mode	EDE3 CBC - triple CBC DES encryption with three keys. CBC = Cipher Block Chaining. EDE = E(ks3,D(ks2,E(ks1,M)))

Properties	
	E=Encrypt D=Decrypt
Encryption Properties (IV, CRC, Key ID)	*Database and Application protectors:* IV, CRC, Key ID (Key ID is the part of the encrypted data) *File protectors (including HDFSFP):* only Key ID (Key ID is separated from the encrypted data and stored in the meta info of encrypted file) **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. *Big Data protectors:* IV, CRC, Key ID (Key ID is the part of the encrypted data)
Length Preservation (padding formula for non-length preserving algorithms)	No For explanation on calculating data length, refer to the section *Data Length and Padding in Encryption*
Minimum Length	None
Maximum Length	2147483610 bytes
Input type /character set	Vary across DBs Refer to the table *Supported Input Data Types by Application Protectors* for supported data types.
Output type /character set	Binary
Return of Protected value	No
Specifics of algorithm	A block cipher with 168 bit key
Supported in Protegrity releases	6.6.x - 8.x.x.x

The following table shows examples of the way in which the value ‘Protegrity’ will be encrypted with the 3DES algorithm.

Table 5-10: 3DES Encryption Examples

Encryption Algorithm	Output Value	Comments
3DES	0x4AA7402C77808D80D093A15A51318D19	The input value of 10 bytes is padded to become 16 (2 blocks of 8 bytes). The output value is 16 bytes.
3DES-CRC	0xF1B7EFD118D27E5568AB192CE2A12E35	The input value of 10 bytes with checksum of 4 bytes is padded to become 16 (2 blocks of 8 bytes). The output value is 16 bytes.
3DES-IV	0x5126D8EB02A213922FB7E6DEDA861ABF661A01AEF7CAEC86	8 bytes IV is added. The output value is 24 bytes (3 blocks of 8 bytes).
3DES-KeyID	0x200479E1CC7983040987362DA49DD68B6E16	2 bytes-Key ID is added. The output value is 18 bytes.
3DES-IV-CRC-KeyID	0x20055B72BF6E9B55B799A9DF51587E93ED8CF42E48A80F9474C0	The input value of 10 bytes with checksum of 4 bytes is padded to 16 bytes. IV of 8 bytes and Key ID of 2 bytes are added. The output value is 26 bytes.

5.3.2 AES-128 and AES-256

The Advanced Encryption Standard (AES) is an encryption algorithm for electronic data that was established by the U.S. National Institute of Standards and Technology (NIST) in 2001.

AES is based on a substitution-permutation network design principle, and is fast in both software and hardware.

AES is used in three versions: AES-128, AES-192, and AES-256. These numbers represent the encryption key sizes (128 bits, 192 bits, and 256 bits) and in their number of rounds (10, 12, and 14, respectively) required to open the vault that is wrapped around the data.

The Protegrity solutions work with AES-128 and AES-256 encryption algorithms.

> **Note:** It is recommended to install the IBM z/OS hotfix PI32471 for DB2 before using the FIELDPROC module with non-CUSP algorithms.

5.3.2.1 AES-128

AES-128 is a version of AES encryption algorithm that has a fixed block size of 16 bytes and a key size of 128 bits.

Table 5-11: AES-128 Encryption Algorithm Properties

Properties	
Name	**AES-128**
Operation Mode	CBC – Cipher Block Chaining

Properties	
Encryption Properties (IV, CRC, Key ID)	*Database and Application protectors:* IV, CRC, Key ID (Key ID is the part of the encrypted data) *File protectors (including HDFSFP):* only Key ID (Key ID is separated from the encrypted data and stored in the meta info of encrypted file) **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. *Big Data protectors:* IV, CRC, Key ID (Key ID is the part of the encrypted data)
Length Preservation (padding formula for non-length preserving algorithms)	No For explanation on calculating data length, refer to the section *Data Length and Padding in Encryption*.
Minimum Length	None
Maximum Length	2147483610 bytes
Input type /character set	Vary across DBs Refer to the table *Supported Input Data Types by Application Protectors* for supported data types.
Output type /character set	Binary
Return of Protected value	No
Specifics of algorithm	A block cipher with 128 bit key
Supported in Protegrity releases	6.6.x – 8.x.x.x

The table shows examples of the way in which the value ‘Protegrity’ will be encrypted with the AES-128 algorithm.

Table 5-12: Encryption Examples

Encryption Algorithm	Output Value	Comments
AES-128	0xA2EA9C9BC53D77BA7F8E85C124296BF3	The input value of 10 bytes is padded to become 16 (1 block of 16 bytes). The output value is 16 bytes.
AES-128-CRC	0x86315F21FA70F7AC1A7D9DB04B22C87A	The input value of 10 bytes with checksum of 4 bytes is padded to become 16 (1 block of 16 bytes). The output value is 16 bytes.

Encryption Algorithm	Output Value	Comments
AES-128-IV	0x0560F196024CCD1CD8213C6657B1BB58CE3047C316EC300BB0BF3C3F5949C157	16 bytes IV is added. The output value is 32 bytes (2 blocks of 16 bytes).
AES-128-KeyID	0x2002E3DE3D7AB6CCBEB96A6A60248559C094	2 bytes Key ID is added. The output value is 18 bytes.
AES-128-IV-CRC-KeyID	0x20031F55A327AFD11DA9E0FEA2499484825E2EABC2B28D162737E867BE0726E7131F	The input value of 10 bytes with checksum of 4 bytes is padded to 16 bytes. IV of 16 bytes and Key ID of 2 bytes are added. The output value is 34 bytes.

5.3.2.2 AES-256

AES-256 is a version of AES encryption algorithm that has a fixed block size of 16 bytes and a key size of 256 bits.

Table 5-13: AES-256 Encryption Algorithm Properties

Properties	
Name	**AES-256**
Operation Mode	CBC – Cipher Block Chaining
Encryption Properties (IV, CRC, Key ID)	*Database and Application protectors:* IV, CRC, Key ID (Key ID is the part of the encrypted data) *File protectors (including HDFSFP):* only Key ID (Key ID is separated from the encrypted data and stored in the meta info of encrypted file) **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. *Big Data protectors:* IV, CRC, Key ID (Key ID is the part of the encrypted data)
Length Preservation (padding formula for non-length preserving algorithms)	No For explanation on calculating data length, refer to section *Data Length and Padding in Encryption*.
Minimum Length	None
Maximum Length	2147483610 bytes
Input type /character set	Vary across DBs

Properties	
	Refer to the table *Supported Input Data Types by Application Protectors* for supported data types.
Output type /character set	Binary
Return of Protected value	No
Specifics of algorithm	A block cipher with 256 bit key
Supported in Protegrity releases	6.6.x – 8.x.x.x

The following table shows examples of the way in which the value ‘Protegrity’ will be encrypted with the AES-256 algorithm.

Table 5-14: AES-256 Encryption Examples

Encryption Algorithm	Output Value	Comments
AES-256	0x0A4771DAD552DA29512BE13BCCF2538A	The input value of 10 bytes is padded to become 16 (1 block of 16 bytes). The output value is 16 bytes.
AES-256-CRC	0x29445B1AEED293D341E9634BD7B7BA4C	The input value of 10 bytes with checksum of 4 bytes is padded to become 16 (1 block of 16 bytes). The output value is 16 bytes.
AES-256-IV	0x2C9D5D8AF80C4614F2C6D063A94BB624C19B14EB40C919F7053DA636ACAE3BEE	16 bytes IV is added. The output value is 32 bytes (2 blocks of 16 bytes).
AES-256-KeyID	0x20157C0E98A1C9E4E6F4D1DCB6FE72B2DA69	2 bytes Key ID is added. The output value is 18 bytes.
AES-256-IV-CRC-KeyID	0x200AA6570EBA6A866F985839C4C189038705C6FC48B2459650940904E76009E300D2	The input value of 10 bytes with checksum of 4 bytes is padded to 16 bytes. IV of 16 bytes and Key ID of 2 bytes are added. The output value is 34 bytes.

5.3.3 CUSP

CUSP (Cryptographic Unit Service Provider) is a special type of CBC mode documented in the z/OS ICSF Application Programmer's Guide (SA22-7522). It is used for handling data with length that is not a multiple of the key block length. It is often used when you want to maintain the original length of the data. The length of encrypted data in CUSP mode will always equal the length of clear text data.

CUSP is best suited for varying types of environments and usage scenarios. For very small-sized data, encrypting with a stream cipher such as CUSP could result in reduced security because it may not include an initialization vector (IV). CUSP is appropriate if the data is greater than one block in size. Larger amounts of data encrypted with CUSP are secure because the CUSP algorithm uses standard chaining block ciphering for the cipher block size pieces of data. For the final data piece less than a cipher block, the CUSP algorithm uses a generated initialization IV only.

The CUSP mode of encryption is not certified by NIST. It is therefore not a part of the NIST standards, or of any other generally accepted body of standards, and has not been formally reviewed by the cryptographic community. Therefore, the use of CUSP mode would be outside the scope of most data security regulations.

Protegrity supports CUSP encryption for z/OS products (prior to version 5.0, CUSP was the default algorithm for z/OS). In the 5.5 release, the support was expanded to Database and Application protectors of all other supported platforms to keep compatibility with z/OS protectors. From 6.6.x, Big Data Protector for Hadoop also supports this encryption method.

Protegrity supports three types of CUSP encryption: CUSP 3DES, CUSP AES-128, and CUSP AES-256. CUSP 3DES uses a 3DES key with the CUSP expansion to the 3DES algorithm. Data is CBC encrypted in 8 byte blocks and any remaining data is stream ciphered using the same 3DES key with an IV of a double encrypted last full block. CUSP AES-128 and CUSP AES-256 CBC encrypt data in 16 byte blocks with any remaining data stream ciphered using the same AES key with an IV of a double encrypted last full block. AES-128 uses a 128 bit key and AES-256 uses a 256 bit key.

> **Note:** If you use the CUSP data element to process long data, then protect and unprotect operations might take more processing time than expected. The CUSP data elements provide a slightly better processing time as compared to the DTP2 data elements.

Table 5-15: CUSP Encryption Algorithm Properties

Properties	
Name	**CUSP 3DES** **CUSP AES-128** **CUSP AES-256**
Operation Mode	CBC – Cipher Block Chaining, combined with ECB - Electronic codebook
Encryption Properties (IV, CRC, Key ID)	CRC, Key ID
Length Preservation (padding formula for non-length preserving algorithms)	Yes (No if CRC and Key ID are used)
Minimum Length	None
Maximum Length	2147483610 bytes
Input type /character set	Vary across DBs Refer to the table *Supported Input Data Types by Application Protectors* for supported data types. On z/OS protectors all input data is treated as binary. For FIELDPROC, DB2 restricts the input data to character or varchar.
Output type /character set	Binary
Return of Protected value	No

Properties	
Specifics of algorithm	A modified block algorithm mainly used in environments where an IBM mainframe is present.
Supported in Protegrity releases	From 6.6.x onwards for z/OS. **Note:** z/OS file protection programs support CUSP, if database or application protector policy is used. From 6.6.x onwards for Database and Application Protectors. From 6.6.x onwards for Big Data Protector for Hadoop.

The following table shows examples of the way in which the value ‘Protegrity’ will be encrypted with the CUSP algorithm.

Table 5-16: CUSP Encryption Examples

Encryption Algorithm	Output Value	Comments
CUSP 3DES	0xD7DE903612B29BA825B4	Length of the output value is the same as input value (10 bytes) as CUSP preserves length.
CUSP AES-128	0x1D95BEFC71590AA7B5C3	
CUSP AES-256	0x1C7244BB85827D36435D	
CUSP 3DES - CRC	0x7920A9AF0CEE96E1C4EDB8F5E9EF	4 bytes checksum is added. The output value is 14 bytes.
CUSP 3DES - KeyID	0x200525200D62B05DCB17E8DB	2 bytes Key ID is added. The output value is 12 bytes.
CUSP 3DES - CRC-KeyID	0x20068C2A54ACB80DB3C3332421B8851B	4 bytes checksum and 2 bytes Key ID are added. The output value is 16 bytes.

Chapter 6

No Encryption

The **No Encryption** protection method when applied lets sensitive data be stored in the clear, but either monitors or masks its usage through a data security policy.

With the **Monitor** method, the sensitive data is accessible by users but its usage is monitored through audit logs that are generated at the protection point and are then delivered to the centrally administered ESA Appliance.

With the **Masking** method, the users who should not use sensitive assets can be prevented from receiving this data, even if the data is stored in the clear.

Both monitoring and masking are controlled by the security officer from the centrally administered ESA Appliance. The **No Encryption** method is highly transparent, which means that the implementation of this method does not cause many changes in the target environment.

The **No Encryption** data element is created in combination with the **Masks** option. The **Masks** option helps define how the masked data output format must be visible to users.

The key difference between masking enabled through a No Encryption data element and masking enabled through a Masking data element is that in the former case, data is always in clear for all roles unless configured to appear masked for unauthorized users, while in the latter case data is always in masked for all users unless configured to appear in clear for authorized users.

For more information about creating masks, refer to the section *Masks*.

Note: If you are reprotecting data using the **No Encryption** method, then the reprotect operation fails in the following scenarios:

- If the data was previously protected using a tokenization or encryption method.
- If the user performing the reprotection of data does not have the *unprotect* privileges on the data element that was used to protect the data.

Table 6-1: No Encryption Algorithm Properties

Properties	
Name	**No Encryption**
Operation Mode	N/A
Encryption Properties (IV, CRC, Key ID)	No
Length Preservation (padding formula for non-length preserving algorithms)	Yes
Minimum Length	None
Maximum Length	≥500 bytes
Input type /character set	Vary across DBs. For supported data types, refer to the table *Supported Input Data Types by Database*.

Properties	
Output type /character set	Output type is the same as the input type, e.g. input type: integer -> output type =integer
Return of Protected value	No
Specifics of algorithm	Does not protect data at rest by changing it. Protection comes from monitoring and masking.
Supported in Protegrity releases	6.6.x-8.x.x.x Supported for Database and Application protection

For Database protection various data types can be used for **No Encryption**. The following table illustrates the supported data types by databases.

Table 6-2: Input Data Types Supported by Application Protectors

Protection Method[*1]	**Application Protectors**					
	AP Python	**AP Java**	**AP C**	**AP .Net**	**AP NodeJS**	**AP Go**
NoEncryption	STRING BYTES FLOAT INT	short int long float double string char[] byte[]	byte[]	STRING BYTE[]	STRING BYTE[]	STRING BYTE[]

Note:
[*1] – If the input and output types of the API are BYTE[], then the customer application should convert the input to and output from the byte array, before calling the API.

Table 6-3: Input Data Types Supported by Database Protectors

Protection Method	**Database**		
	MSSQL Server	**Oracle**	**DB/2**
NoEncryption	varchar varbinary binary	varchar2 char number	varchar2 char number

Protection Method	Database		
	MSSQL Server	**Oracle**	**DB/2**
	char numeric real float decimal int bigint smallint tinyint datetime bit nvarchar nchar money smallmoney smalldatetime uniqueidentifier	real float date raw blob clob	real float date timestamp long varchar double varchar numeric decimal int bigint smallint tinyint datetime character time blob clob

Table 6-4: Input Data Types Supported by MPP Database Protectors

Protection Method	Database		
	Pivotal GPDB	**Teradata**	**IBM Netezza**
NoEncryption	varchar integer decimal date	varchar char integer float decimal	varchar int real date

Protection Method	Database		
	Pivotal GPDB	Teradata	IBM Netezza
		date bigint	

Table 6-5: Input Data Types Supported for Big Data Protectors

Protection Method *1	Big Data						
	MapReduce	Hive	Pig	HBase	Impala	Spark	Spark SQL
NoEncryption	BYTE[] INT LONG	CHAR STRING FLOAT DOUBLE INT BIGINT HIVEDECIMAL	CHARARRAY INT	BYTE[]	STRING INT FLOAT DOUBLE	BYTE[] STRING FLOAT DOUBLE SHORT INT LONG	STRING FLOAT DOUBLE SHORT INT LONG BIGDECIMAL *2

Note:

*1 - The customer application should convert the input to and output from byte array.

*2 - If decimal format data is protected by the Decimal UDFs using the *No Encryption* data element, then the protected data is trimmed to the scale of 18 digits.

The following table shows examples of the way in which a value will be protected with the No Encryption algorithm.

Table 6-6: Output Values for No Encryption Algorithm

Protection Method	Roles in Data Element	Input Value	Output Value	Comments
No Encryption	None	Protegrity	Protegrity	The value is stored in the clear.

Chapter 7

Monitor

The **Monitor** protection method is generally used for auditing. As an organization, if you plan to monitor and assess users that are trying to access the data without protection, choose the Monitor protection method. This element does not restrict any data security operation for any user, but instead audits attempts to add, access, or change data by users. The audit logs generated at the protection point are forwarded to the log management system.

Note:
When you protect data, the following error message is displayed,

```
The user does not have the appropriate permissions to perform the requested operation
```

if:

- masking or monitoring data elements are configured in the policy, and
- username is not specified in the policy

Attention: Due to a limitation, when you unprotect data, a warning severity is returned instead of an error.

Similar to the No Encryption method, implementation of the Monitor method does not cause many changes in the target environment.

Table 7-1: Monitor Algorithm Properties

Properties	
Name	**Monitor**
Operation Mode	N/A
Encryption Properties (IV, CRC, Key ID)	No
Length Preservation (padding formula for non-length preserving algorithms)	Yes
Input type /character set	Vary across DBs. For supported data types, refer to the table *Supported Input Data Types by Database*.
Output type /character set	Output type is the same as the input type, e.g. input type: integer -> output type =integer
Return of Protected value	No
Specifics of algorithm	Does not protect data at rest by changing it. Used for monitoring and auditing.
Supported in Protegrity releases	7.x-8.x.x.x Supported for DSG, Database, Application protectors, and Data Security Gateway

For Database protection, various data types can be used for Monitor method. The following table illustrates the supported data types by databases.

Table 7-2: Input Data Types Supported by Application Protectors

Protection Method[*1]	**Application Protectors**					
	AP Python	**AP Java**	**AP C**	**AP .Net**	**AP NodeJS**	**AP Go**
Monitor	STRING BYTES FLOAT INT	short int long float double string char[] byte[]	byte[]	STRING BYTE[]	STRING BYTE[]	STRING BYTE[]

Note:
[*1] – If the input and output types of the API are BYTE[], then the customer application should convert the input to and output from the byte array, before calling the API.

Table 7-3: Input Data Types Supported by Database Protectors

Protection Method	**Database**		
	MSSQL Server	**Oracle**	**DB/2**
Monitor	varchar varbinary binary char numeric real float decimal int	varchar2 char number real float date raw blob clob	varchar2 char number real float date timestamp long varchar double

Protection Method	Database		
	MSSQL Server	**Oracle**	**DB/2**
	bigint smallint tinyint datetime bit nvarchar nchar money smallmoney smalldatetime uniqueidentifier		varchar numeric decimal int bigint smallint tinyint datetime character time blob clob

Table 7-4: Input Data Types Supported by MPP Database Protectors

Protection Method	Database		
	Pivotal GPDB	**Teradata**	**IBM Netezza**
Monitor	varchar integer decimal date	varchar char integer float decimal date bigint	varchar int real date

Table 7-5: Input Data Types Supported by Big Data Protectors

Protection Method *1	**Big Data**						
	MapReduce	**Hive**	**Pig**	**HBase**	**Impala**	**Spark**	**Spark SQL**
Monitor	BYTE[] INT LONG	CHAR STRING FLOAT DOUBLE INT BIGINT HIVEDECIMAL	CHARARRAY INT	BYTE[]	STRING INT FLOAT DOUBLE	BYTE[] STRING FLOAT DOUBLE SHORT INT LONG	STRING FLOAT DOUBLE SHORT INT LONG BIGDECIMAL *2

Note:

*1 - The customer application should convert the input to and output from byte array.

*2 - If decimal format data is protected by the Decimal UDFs using the *Monitor* data element, then the protected data is trimmed to the scale of 18 digits.

The following table shows examples of the way in which a value will be protected with the Monitor algorithm.

Table 7-6: Output Values for Monitor Algorithm

Protection Method	**Input Value**	**Output Value**	**Comments**
Monitor	Protegrity	Protegrity	The value is stored in the clear. An audit log is generated.

Chapter 8

Masking

For situations where data output restrictions must be applied for users, the Masking method can be used.

As an organization, if you plan to restrict access such that only users with required privileges can view sensitive data, while other users view masked data, the Masking method can be used. Considering the sensitive data is residing in protection endpoint in clear, based on how the Masking data element is configured, users are granted view access. The masking data element as a default considers all users as restricted users and displays masked sensitive data. If any user must be granted access to view clear data, then it must be configured through roles.

For example, consider policy users *user1* and *user2* trying to access CCN data. As default, when the masking policy is created, both users view the CCN data in masked format, such as ****45856655****. If the *user1* is granted privilege to view data in clear, then *user1* view the CCN data in clear while the *user2* still sees masked CCN data.

The key difference between masking enabled through a No Encryption data element and masking enabled through a Masking data element is that in the former case, data is always in clear for all users unless configured to appear masked for unauthorized users, while in the latter case data is always in masked for all users unless configured to appear in clear for authorized users.

Similar to the No Encryption method, implementation of the Masking method does not cause any changes in the target environment.

The Masking data element is created in combination with the **Masks** option. The **Masks** option helps define how the masked data output format must be visible to users.

For more information about creating masks, refer to the section *Masks*.

Note:
When you protect data, the following error message is displayed,

```
The user does not have the appropriate permissions to perform the requested operation
```

if:

- masking or monitoring data elements are configured in the policy, and
- username is not specified in the policy

Attention: Due to a limitation, when you unprotect data, a warning severity is returned instead of an error.

Table 8-1: Masking Algorithm Properties

Properties	
Name	**Masking**

Properties	
Operation Mode	N/A
Encryption Properties (IV, CRC, Key ID)	No
Length Preservation (padding formula for non-length preserving algorithms)	Yes
Input type /character set	Vary across DBs. For supported data types, refer to the table *Supported Input Data Types by Database*.
Output type /character set	Output type is the same as the input type, e.g. input type: integer -> output type =integer
Return of Protected value	No
Specifics of algorithm	Does not protect data at rest by changing it. Protection comes from masking.
Supported in Protegrity releases	6.6.x-8.x.x.x Supported for Database and Application protectors Supported for Data Security Gateway

For Database protection various data types can be used for Masking method. The following table illustrates supported data types by databases.

Table 8-2: Input Data Types Supported by Application Protectors

Protection Method*1	**Application Protectors**					
	AP Python	**AP Java**	**AP C**	**AP .Net**	**AP NodeJS**	**AP Go**
Masking	STRING BYTES FLOAT INT	short int long float double string char[] byte[]	byte[]	STRING BYTE[]	STRING BYTE[]	STRING BYTE[]

Note:

*1 – If the input and output types of the API are BYTE[], then the customer application should convert the input to and output from the byte array, before calling the API.

Table 8-3: Input Data Types Supported by Database Protectors

Protection Method	Database		
	MSSQL Server	**Oracle**	**DB/2**
Masking	varchar varbinary binary char numeric real float decimal int bigint smallint tinyint datetime bit nvarchar nchar money smallmoney smalldatetime uniqueidentifier	varchar2 char number real float date raw blob clob	varchar2 char number real float date timestamp long varchar double varchar numeric decimal int bigint smallint tinyint datetime character time blob clob

Table 8-4: Input Data Types Supported by MPP Database Protectors

Protection Method	Database		
	Pivotal GPDB	**Teradata**	**IBM Netezza**
Masking	varchar	varchar	varchar

Protection Method	Database		
	Pivotal GPDB	**Teradata**	**IBM Netezza**
	integer decimal date	char integer float decimal date bigint	int real date

Table 8-5: Input Data Types Supported for Big Data Protectors

Protection Method *1	Big Data						
	MapReduce	**Hive**	**Pig**	**HBase**	**Impala**	**Spark**	**Spark SQL**
Masking	BYTE[]	CHAR STRING	CHARARRAY	BYTE[]	STRING	BYTE[] STRING	STRING

Note:
*1 - The customer application should convert the input to and output from byte array.

The following table shows examples in which a value will be protected with the Masking algorithm.

Table 8-6: Output Values for Masking Algorithm

Protection Method	Roles in Data Element	Input Value	Output Value	Comments
Masking	None	Protegrity	****egrity	The value is displayed in masked format.
	exampleuser1 with *Unprotect* access and masking selected	Protegrity	• **All users** - ****egrity • **exampleuser1** - Protegrity	The value is stored in the clear. Any other user apart from *exampleuser1* will see masked content.

8.1 Masks

The **Masks** option is a data output restriction that is used in combination with the tokenization, encryption, no encryption, and masking protection methods.

Masks define data output formatting, which means what data to disclose to users that want to view the data. The formatting includes decrypting and transforming the result in a way that part of it is obfuscated. For example, a masked social security number could look like: 12345****, or ***456789.

Using a mask for the output is optional. If none is specified, then all data is returned in the masked output format by default for all users who are not a part of any policy. If users are a part of the policy, then data is shown in the clear in case of a **No Encryption** data element and in masked output format in case of **Masking** data element.

Masks are defined in the ESA and have the following properties:

- Mask name and description
- Number of characters from left
- Number of characters from right
- Whether ‘left’ and ‘right’ should be masked or clear
- Specific mask character (*,#,-,0,1,2,3,4,5,6,7,8, or 9).

The mask definition or how the mask looks like is implemented as a per role and data element combination. This means that one data element can have multiple mask definitions.

When a mask is applied to data that is too short, that is, the data will not match to what has been defined in the mask, everything gets masked. For example, if a mask of 6 from the left and 2 from the right will be applied to data that has a length of 4, such as a name *John*, then all four characters will be masked.

Important:

If a user role is included in multiple policies with masks, then the masks may conflict in one of the following conditions:

- The user has different mask settings for both roles for the same data element. In this case, the user's access rights to the data element with the conflicting masks are revoked.
- The user has the data element with a mask in a role and another with no mask settings in the other role. In this case, the user's access rights to the data element is set to the role with no mask settings.

For detailed information about masking rules for users in multiple roles with different scenarios, refer to the section *4.3.5 Masking Rules for Users in Multiple Roles* in the *Policy Management Guide 9.1.0.0*.

Masking can only apply to character data. If a data element with masking is used on non-character data, then the masking is ignored.

Important:

It is not recommended to use Masking with multibyte encodings, such as UTF-8, UTF-16, and so on, as it might corrupt the data.

Sample Protected Data:
Left and Right Masking settings: L-3 and R-3
Unprotected Data with Mask applied: ##? ?##

Sample Protected Data:
Left and Right Clear settings: L-3 and R-3
Unprotected Data with Mask applied: ?###### #### ##########?

In the above case, the masked unprotected value is distorted as every character in the input is represented by 2 bytes and we are trying to preserve the first 3 bytes from the left and 3 bytes from the right, which results in a distorted output.

The following table shows examples of the way in which Masks can be used in combination with other protection methods.

Table 8-7: Examples of Masks

Protection Method/ Mask	Input Value	Output Value	Comments
CCN 6x4 *Left=6, Right=4, Clear, **	4537432557929840	453743******9840	Pre-defined mask , exposes the first 6 characters and last 4 characters
CCN 12x0 *Left=12, Right=0, Mask, **	4537432557929840	**********9840	Pre-defined mask , hides the first 12 characters
CCN 4x4 *Left=4, Right=4, Clear, **	4537432557929840	4537********9840	Pre-defined mask , exposes the first 4 characters and last 4 characters
CCN 6x4 *Left=6, Right=4, Clear, 1*	4537432557929840	4537431111119840	Pre-defined mask, exposes the first 6 characters and last 4 characters
SSN x-4 *Left=0, Right=4, Clear, **	721-07-4426	*******4426	Pre-defined mask , exposes the last 4 characters
SSN 5-x *Left=5, Right=0, Clear, **	72107-4426	72107*****	Pre-defined mask , exposes the first 5 characters
SSN 5-x *Left=5, Right=0, Clear, 0*	72107-4426	7210700000	Pre-defined mask, exposes the first 5 characters
CustomMask1 *Left=6, Right=0, Mask, #*	721-07-4426	######-4426	Custom mask, illustrates usage of ‘#’ mask character
CustomMask2 *Left=4, Right=4, Mask, -*	4537432557929840	----43255792----	Custom mask, illustrates usage of ‘-’ mask character
CustomMask3 *Left=4, Right=4, Mask, 8*	4537432557929840	8888432557928888	Custom mask, illustrates usage of ‘8’ mask character

Caution: The z/OS Database Protector EDITPROC does not support masking.

Chapter 9

Hashing

Hashing is an alternative to encryption for protecting sensitive data. A hash function is a reproducible method of turning data into a (relatively) small number that can serve as a digital fingerprint of the data. The algorithm "chops and mixes" (for example, substitutes or transposes) the data to create these fingerprints. Protegrity uses the HMAC (Hashed Message Authentication Code) SHA1 algorithm that returns a 160 bit hash value for any data.

Hashing (HMAC-SHA1) is utilized to transform sensitive data. Unlike encryption, transformed (hashed) data is irreversible as it is replaced with a checksum and not stored anywhere as an encrypted value. The original data cannot be retrieved back from the hashed value.

Table 9-1: Hashing Protection Algorithm Properties

Properties	
Name	**HMAC-SHA1**
Operation Mode	N/A
Encryption Properties (IV, CRC, Key ID)	No
Length Preservation (padding formula for non-length preserving algorithms)	No Result is always 20 bytes regardless of input length.
Minimum Length	None
Maximum Length	≥ 500 bytes
Input type /character set	Vary across DBs Refer to the table *Supported Input Data Types by Database* for supported data types.
Output type /character set	Binary
Return of Protected value	No
Specifics of algorithm	Irreversible protection method. Original data is replaced with a checksum and cannot be retrieved back, when decrypted.
Supported in Protegrity releases	6.6.x - 8.x.x.x Supported for Database and Application protection

For Database protection only varchar/char data types can be used for hashing. The following tables illustrate supported data types by Protegrity Protectors.

Table 9-2: Supported Input Data Types by Application Protectors

Protection Method[*1]	Application Protectors					
	AP Python	**AP NodeJS**	**AP Go**	**AP Java**	**AP .Net**	**AP C**
HMAC-SHA1	STRING BYTES	BYTE[]	BYTE[]	float double string char[] byte[]	BYTE[]	byte[]

Note:
[*1] – If the input and output types of the API are BYTE[], then the customer application should convert the input to and output from the byte array, before calling the API.

Table 9-3: Supported Input Data Types by Database Protectors

Protection Method	Database		
	MSSQL Server	**Oracle**	**DB/2**
HMAC-SHA1	varchar char	varchar2 char	varchar2 varchar char

Table 9-4: Supported Input Data Types by MPP Database Protectors

Protection Method	Database		
	Pivotal GPDB	**Teradata**	**IBM Netezza**
HMAC-SHA1	varchar	varchar char integer float	Not supported

Table 9-5: Supported Input Data Types for Big Data

Protection Method [*1]	***Big Data***						
	MapReduce	**Hive**	**Pig**	**HBase**	**Impala**	**Spark**	**Spark SQL**
HMAC-SHA1	BYTE[]	Not supported	Not supported	BYTE[]	Not supported	BYTE[]	Not supported

Note:
[*1] – The customer application should convert the input to and output from byte array.

The following table shows examples of the way in which a value will be replaced with the HMAC-SHA1 hashing type.

Table 9-6: HMAC-SHA1 Hashing Output Values

Protection Method	**Input Value**	**Output Value**	**Comments**
HMAC-SHA1	Protegrity	0x0153DE6D5B43A0C38F8359643775F4 D6D18FCD13	Output value cannot be decrypted

Chapter 10

Appendix A: ASCII Character Codes

Lower ASCII token – character codes 33-126 (*Table A-1*)

Printable token – character codes 32-126 (*Table A-1*), 160-255 (*Table A-2*)

Unicode token – character codes 32-127 (*Table A-1*), 128-255 (*Table A-2*), 0-31 (*Table A-3*)

Binary token – character codes 32-127 (*Table A-1*), 128-255 (*Table A-2*), 0-31 (*Table A-3*)

Table 10-1: ASCII printable characters (character code 32-127)

Character ASCII code		**Character Description**		**Character ASCII code**		**Character Description**	
DEC	**HEX**	**Symbol**	**Description**	**DEC**	**HEX**	**Symbol**	**Description**
32	20	(Space)	Space	80	50	P	Uppercase P
33	21	!	Exclamation mark	81	51	Q	Uppercase Q
34	22	"	Double quotes (or speech marks)	82	52	R	Uppercase R
35	23	#	Number	83	53	S	Uppercase S
36	24	$	Dollar	84	54	T	Uppercase T
37	25	%	Percent sign	85	55	U	Uppercase U
38	26	&	Ampersand	86	56	V	Uppercase V
39	27	'	Single quote	87	57	W	Uppercase W
40	28	(	Open parenthesis (or open bracket)	88	58	X	Uppercase X
41	29	)	Close parenthesis (or close bracket)	89	59	Y	Uppercase Y
42	2A	*	Asterisk	90	5A	Z	Uppercase Z

Character ASCII code		Character Description		Character ASCII code		Character Description	
DEC	**HEX**	**Symbol**	**Description**	**DEC**	**HEX**	**Symbol**	**Description**
43	2B	+	Plus	91	5B	[	Opening bracket
44	2C	,	Comma	92	5C	\	Backslash
45	2D	-	Hyphen	93	5D	]	Closing bracket
46	2E	.	Period, dot or full stop	94	5E	^	Caret - circumflex
47	2F	/	Slash or divide	95	5F	_	Underscore
48	30	0	Zero	96	60	`	Grave accent
49	31	1	One	97	61	a	Lowercase a
50	32	2	Two	98	62	b	Lowercase b
51	33	3	Three	99	63	c	Lowercase c
52	34	4	Four	100	64	d	Lowercase d
53	35	5	Five	101	65	e	Lowercase e
54	36	6	Six	102	66	f	Lowercase f
55	37	7	Seven	103	67	g	Lowercase g
56	38	8	Eight	104	68	h	Lowercase h
57	39	9	Nine	105	69	i	Lowercase i
58	3A	:	Colon	106	6A	j	Lowercase j
59	3B	;	Semicolon	107	6B	k	Lowercase k
60	3C	<	Less than (or open angled bracket)	108	6C	l	Lowercase l
61	3D	=	Equals	109	6D	m	Lowercase m
62	3E	>	Greater than (or close angled bracket)	110	6E	n	Lowercase n
63	3F	?	Question mark	111	6F	o	Lowercase o

Character ASCII code		Character Description		Character ASCII code		Character Description	
DEC	**HEX**	**Symbol**	**Description**	**DEC**	**HEX**	**Symbol**	**Description**
64	40	@	At symbol	112	70	p	Lowercase p
65	41	A	Uppercase A	113	71	q	Lowercase q
66	42	B	Uppercase B	114	72	r	Lowercase r
67	43	C	Uppercase C	115	73	s	Lowercase s
68	44	D	Uppercase D	116	74	t	Lowercase t
69	45	E	Uppercase E	117	75	u	Lowercase u
70	46	F	Uppercase F	118	76	v	Lowercase v
71	47	G	Uppercase G	119	77	w	Lowercase w
72	48	H	Uppercase H	120	78	x	Lowercase x
73	49	I	Uppercase I	121	79	y	Lowercase y
74	4A	J	Uppercase J	122	7A	z	Lowercase z
75	4B	K	Uppercase K	123	7B	{	Opening brace
76	4C	L	Uppercase L	124	7C	|	Vertical bar
77	4D	M	Uppercase M	125	7D	}	Closing brace
78	4E	N	Uppercase N	126	7E	~	Equivalency sign - tilde
79	4F	O	Uppercase O	127	7F	(Delete)	Delete

Table 10-2: Extended ASCII codes (character code 128-255)

Character ASCII code		Character Description		Character ASCII code		Character Description	
DEC	**HEX**	**Symbol**	**Description**	**DEC**	**HEX**	**Symbol**	**Description**
128	80	€	Euro sign	192	C0	À	Latin capital letter A with grave
129	81			193	C1	Á	Latin capital letter A with acute

Character ASCII code		Character Description		Character ASCII code		Character Description	
DEC	**HEX**	**Symbol**	**Description**	**DEC**	**HEX**	**Symbol**	**Description**
130	82	‚	Single low-9 quotation mark	194	C2	Â	Latin capital letter A with circumflex
131	83	*ƒ*	Latin small letter f with hook	195	C3	Ã	Latin capital letter A with tilde
132	84	„	Double low-9 quotation mark	196	C4	Ä	Latin capital letter A with diaeresis
133	85	…	Horizontal ellipsis	197	C5	Å	Latin capital letter A with ring above
134	86	†	Dagger	198	C6	Æ	Latin capital letter AE
135	87	‡	Double dagger	199	C7	Ç	Latin capital letter C with cedilla
136	88	ˆ	Modifier letter circumflex accent	200	C8	È	Latin capital letter E with grave
137	89	‰	Per mille sign	201	C9	É	Latin capital letter E with acute
138	8A	Š	Latin capital letter S with caron	202	CA	Ê	Latin capital letter E with circumflex
139	8B	‹	Single left-pointing angle quotation	203	CB	Ë	Latin capital letter E with diaeresis
140	8C	Œ	Latin capital ligature OE	204	CC	Ì	Latin capital letter I with grave
141	8D			205	CD	Í	Latin capital letter I with acute
142	8E	Ž	Latin captial letter Z with caron	206	CE	Î	Latin capital letter I with circumflex
143	8F			207	CF	Ï	Latin capital letter I with diaeresis
144	90			208	D0	Đ	Latin capital letter ETH
145	91	‘	Left single quotation mark	209	D1	Ñ	Latin capital letter N with tilde

Character ASCII code		Character Description		Character ASCII code		Character Description	
DEC	**HEX**	**Symbol**	**Description**	**DEC**	**HEX**	**Symbol**	**Description**
146	92	’	Right single quotation mark	210	D2	Ò	Latin capital letter O with grave
147	93	“	Left double quotation mark	211	D3	Ó	Latin capital letter O with acute
148	94	”	Right double quotation mark	212	D4	Ô	Latin capital letter O with circumflex
149	95	•	Bullet	213	D5	Õ	Latin capital letter O with tilde
150	96	–	En dash	214	D6	Ö	Latin capital letter O with diaeresis
151	97	—	Em dash	215	D7	×	Multiplication sign
152	98	˜	Small tilde	216	D8	Ø	Latin capital letter O with slash
153	99	™	Trade mark sign	217	D9	Ù	Latin capital letter U with grave
154	9A	š	Latin small letter S with caron	218	DA	Ú	Latin capital letter U with acute
155	9B	›	Single right-pointing angle quotation mark	219	DB	Û	Latin capital letter U with circumflex
156	9C	œ	Latin small ligature oe	220	DC	Ü	Latin capital letter U with diaeresis
157	9D			221	DD	Ý	Latin capital letter Y with acute
158	9E	ž	Latin small letter z with caron	222	DE	Þ	Latin capital letter THORN
159	9F	Ÿ	Latin capital letter Y with diaeresis	223	DF	ß	Latin small letter sharp s - ess-zed
160	A0	(Non-breaking space)	Non-breaking space	224	E0	à	Latin small letter a with grave
161	A1	¡	Inverted exclamation mark	225	E1	á	Latin small letter a with acute

Character ASCII code		Character Description		Character ASCII code		Character Description	
DEC	**HEX**	**Symbol**	**Description**	**DEC**	**HEX**	**Symbol**	**Description**
162	A2	¢	Cent sign	226	E2	â	Latin small letter a with circumflex
163	A3	£	Pound sign	227	E3	ã	Latin small letter a with tilde
164	A4	¤	Currency sign	228	E4	ä	Latin small letter a with diaeresis
165	A5	¥	Yen sign	229	E5	å	Latin small letter a with ring above
166	A6	¦	Pipe, Broken vertical bar	230	E6	æ	Latin small letter ae
167	A7	§	Section sign	231	E7	ç	Latin small letter c with cedilla
168	A8	¨	Spacing dieresis - umlaut	232	E8	è	Latin small letter e with grave
169	A9	©	Copyright sign	233	E9	é	Latin small letter e with acute
170	AA	ª	Feminine ordinal indicator	234	EA	ê	Latin small letter e with circumflex
171	AB	«	Left double angle quotes	235	EB	ë	Latin small letter e with diaeresis
172	AC	¬	Not sign	236	EC	ì	Latin small letter i with grave
173	AD	(Soft hyphen)	Soft hyphen	237	ED	í	Latin small letter i with acute
174	AE	®	Registered trade mark sign	238	EE	î	Latin small letter i with circumflex
175	AF	¯	Spacing macron - overline	239	EF	ï	Latin small letter i with diaeresis
176	B0	°	Degree sign	240	F0	ð	Latin small letter eth
177	B1	±	Plus-or-minus sign	241	F1	ñ	Latin small letter n with tilde

Character ASCII code		Character Description		Character ASCII code		Character Description	
DEC	**HEX**	**Symbol**	**Description**	**DEC**	**HEX**	**Symbol**	**Description**
178	B2	²	Superscript two - squared	242	F2	ò	Latin small letter o with grave
179	B3	³	Superscript three - cubed	243	F3	ó	Latin small letter o with acute
180	B4	´	Acute accent - spacing acute	244	F4	ô	Latin small letter o with circumflex
181	B5	µ	Micro sign	245	F5	õ	Latin small letter o with tilde
182	B6	¶	Pilcrow sign - paragraph sign	246	F6	ö	Latin small letter o with diaeresis
183	B7	·	Middle dot - Georgian comma	247	F7	÷	Division sign
184	B8	¸	Spacing cedilla	248	F8	ø	Latin small letter o with slash
185	B9	¹	Superscript one	249	F9	ù	Latin small letter u with grave
186	BA	º	Masculine ordinal indicator	250	FA	ú	Latin small letter u with acute
187	BB	»	Right double angle quotes	251	FB	û	Latin small letter u with circumflex
188	BC	¼	Fraction one quarter	252	FC	ü	Latin small letter u with diaeresis
189	BD	½	Fraction one half	253	FD	ý	Latin small letter y with acute
190	BE	¾	Fraction three quarters	254	FE	þ	Latin small letter thorn
191	BF	¿	Inverted question mark	255	FF	ÿ	Latin small letter y with diaeresis

Table 10-3: ASCII control characters (character code 0-31)

Character ASCII code		**Character Description**		**Character ASCII code**		**Character Description**	
DEC	HEX	Symbol	Description	DEC	HEX	Symbol	Description
0	0	NUL	Null char	16	10	DLE	Data Line Escape
1	1	SOH	Start of Heading	17	11	DC1	Device Control 1 (oft. XON)
2	2	STX	Start of Text	18	12	DC2	Device Control 2
3	3	ETX	End of Text	19	13	DC3	Device Control 3 (oft. XOFF)
4	4	EOT	End of Transmission	20	14	DC4	Device Control 4
5	5	ENQ	Enquiry	21	15	NAK	Negative Acknowledgement
6	6	ACK	Acknowledgment	22	16	SYN	Synchronous Idle
7	7	BEL	Bell	23	17	ETB	End of Transmit Block
8	8	BS	Back Space	24	18	CAN	Cancel
9	9	HT	Horizontal Tab	25	19	EM	End of Medium
10	0A	LF	Line Feed	26	1A	SUB	Substitute
11	0B	VT	Vertical Tab	27	1B	ESC	Escape
12	0C	FF	Form Feed	28	1C	FS	File Separator
13	0D	CR	Carriage Return	29	1D	GS	Group Separator
14	0E	SO	Shift Out / X-On	30	1E	RS	Record Separator
15	0F	SI	Shift In / X-Off	31	1F	US	Unit Separator

Chapter 11

Appendix B: Examples of Column Sizes Calculation for Encryption

Table 11-1: Column Sizes Calculation for 3DES encryption

Data Type	Size (bytes)	3DES	3DES-CRC	3DES-IV	3DES-IV-CRC	3DES-IV-CRC-KeyID
Maximum padding size		**8**	**8**	**8**	**8**	**8**
Checksum size		**0**	**4**	**0**	**4**	**4**
IV Size		**0**	**0**	**8**	**8**	**8**
SMALLINT	2	8	8	16	16	18
INTEGER	4	8	16	16	24	26
BIGINT	8	16	16	24	24	26
DATE	4	8	16	16	24	26
DECIMAL(1..2)	1	8	8	16	16	18
DECIMAL(3..4)	2	8	8	16	16	18
DECIMAL(5..9)	4	8	16	16	24	26
DECIMAL(10..18)	8	16	16	24	24	26
DECIMAL(19..38)	16	24	24	32	32	34
FLOAT, REAL	8	16	16	24	24	26
Latin CHAR / VARCHAR	5	8	16	16	24	26
Unicode CHAR / VARCHAR	5	16	16	24	24	26

Table 11-2: Column Sizes Calculation for AES encryption (AES-256 and AES-128)

Data Type	Size (bytes)	AES	AES-CRC	AES-IV	AES-IV-CRC	AES-IV-CRC-KeyID
Maximum padding size	-	16	16	16	16	16
Checksum size	-	0	4	0	4	4
IV Size	-	0	0	16	16	16
SMALLINT	2	16	16	32	32	34
INTEGER	4	16	16	32	32	34
BIGINT	8	16	16	32	32	34
DATE	4	16	16	32	32	34
DECIMAL(1..2)	1	16	16	32	32	34
DECIMAL(3..4)	2	16	16	32	32	34
DECIMAL(5..9)	4	16	16	32	32	34
DECIMAL(10..18)	8	16	16	32	32	34
DECIMAL(19..38)	16	32	32	48	48	50
FLOAT, REAL	8	16	16	32	32	34
Latin CHAR / VARCHAR	5	16	16	32	32	34
Unicode CHAR / VARCHAR	5	16	16	32	32	34

The sizes of database native data types may vary, but the column sizes calculation provided in the above tables is generic.

Chapter 12

Appendix C: Empty String Handling by Protectors

Empty strings can be protected by tokenization and encryption. When these protected empty strings are unprotected, the output may differ depending on how the Protector is configured. Protegrity Protectors provide empty string handling support specifically with UDFs for varchar data types. For other data types, empty string handling depends on how the database itself represents the empty strings.

You can configure the expected behavior of your Protector using the *pepserver.cfg* configuration file.

The following table explains the Protectors behavior with respect to the parameter selected in the *pepserver.cfg* file.

Table 12-1: Empty String handling by Data Protectors on Open Systems

empty string setting in pepserver.cfg	**Operation with empty string ‘’**	**Handling by different Protectors on Open Systems**		
		MSSQL Server	**DB2**	**Oracle**
NULL (default)	encrypt ‘’ (by all DEs)	NULL	NULL	NULL
	encrypt ‘’ by FPE	NULL	NULL	NULL
	tokenize ‘’	NULL	NULL	NULL
	decrypt encrypted ‘’	NULL	NULL	NULL
encrypt	encrypt ‘’ by AES-128, AES-256, 3DES	encrypted value	encrypted value	NULL
	encrypt ‘’ by CUSP	error message	error message	NULL
	encrypt ‘’ by HMAC-SHA1	NULL	encrypted value	NULL
	encrypt ‘’ by No_ENC	EMPTY	EMPTY	NULL
	encrypt ‘’ by FPE	NULL	EMPTY	NULL
	tokenize ‘’	NULL	EMPTY	NULL
	decrypt encrypted ‘’	EMPTY	EMPTY	NULL
empty	encrypt ‘’ by AES-128, AES-256, 3DES	EMPTY	EMPTY	NULL

empty string setting in pepserver.cfg	Operation with empty string ‘’	Handling by different Protectors on Open Systems		
		MSSQL Server	DB2	Oracle
	encrypt ‘’ by CUSP	EMPTY	EMPTY	NULL
	encrypt ‘’ by FPE	EMPTY	EMPTY	NULL
	tokenize ‘’	EMPTY	EMPTY	NULL
	decrypt encrypted ‘’	EMPTY	EMPTY	NULL
	Hashing " value by HMAC-SHA1	NULL		

Note:

Due to inconsistent behavior, changing the default behavior for empty strings handling (**emptystring = NULL**) is only recommended for DB2.

Emptystring setting affects only ‘’ value. NULL and any other values are not impacted by this setting.

Oracle handles empty string ‘’ as NULL, thus emptystring setting is not applicable in Oracle.

MSSQL Server PTY functions implementation returns NULL after decryption. Thus, with emptystring=encrypt empty string ‘’ as initial value will be encrypted, but decrypted as NULL.

Note:

For Database and MPP Database Protectors, a column protected in a table should allow NULL. If a column does not allow NULL, then attempts to encrypt with default **emptystring = NULL** setting will result in error.

With **emptystring=encrypt** empty string ‘’ will be encrypted by the following data elements: AES-256, AES-128, 3DES, CUSP, NoEnc, HMAC-SHA1. For token data elements, the following error is returned: `Invalid input data`.

Note:

Users of XC cannot distinguish if a NULL or an empty string is returned from a decrypt operation, that is why with setting **emptystring=NULL**, NULL is returned. NULL is also returned if a user lacks permissions to decrypt the data.

Before changing the emptystring setting in *pepserver.cfg*, it is recommended to decrypt the necessary values using the setting in which they were encrypted.

This may be needed because in some cases you may get "integrity failed” error, for example, on attempt to decrypt a value with **emptystring=encrypt**, encrypted with **emptystring=empty**.

Table 12-2: Empty String handling by Application Protectors on Open Systems

Emptystring setting in pepserver.cfg	Operation with empty string ''	Handling by different Application Protectors on Open Systems					
		AP C	AP NodeJS	AP .Net	AP Java	AP Python	AP Golang
NULL (default)	protect '' by NoEnc	NULL	NULL	NULL	NULL	None	EMPTY
	encrypt '' by AES-256	NULL	NULL	NULL	NULL	None	EMPTY
	encrypt '' by FPE	NULL	NULL	NULL	NULL	None	EMPTY
	tokenize '' by any Token Element	NULL	NULL	NULL	NULL	None	EMPTY
	decrypt encrypted '' by data element that was used to encrypt	NULL	NULL	NULL	NULL	None	EMPTY
	detokenize tokenized '' by token element that was used to protect	NULL	NULL	NULL	NULL	None	EMPTY
	unprotect protected '' by NoEnc data element that was used to protect	NULL	NULL	NULL	NULL	None	EMPTY
	reprotect protected '' by any Encryption Data element	NULL	NULL	NULL	NULL	None	EMPTY
	reprotect protected '' by any Token element	NULL	NULL	NULL	NULL	None	EMPTY
	reprotect protected '' by NoEnc	NULL	NULL	NULL	NULL	None	EMPTY
encrypt	protect '' by NoEnc	NULL	NULL	NULL	NULL	NULL	NULL
	encrypt '' by AES-256	encrypted value	encrypted value	encrypted value	encrypted value	encrypted value	encrypted value
	encrypt '' by HMAC-SHA1, CUSP	CUSP: ERROR MESSAGE: 44, The	CUSP: ERROR MESSAGE: 44, The	CUSP: ERROR MESSAGE: 44, The	CUSP: ERROR MESSAGE: 44, The	CUSP: ERROR MESSAGE: 44, The	CUSP: ERROR MESSAGE: 44, The

Emptystring setting in pepserver.cfg	Operation with empty string ''	Handling by different Application Protectors on Open Systems					
		AP C	AP NodeJS	AP .Net	AP Java	AP Python	AP Golang
		content of the input data is not valid HMAC-SHA1: hashed value	content of the input data is not valid HMAC-SHA1: hashed value	content of the input data is not valid HMAC-SHA1: hashed value	content of the input data is not valid HMAC-SHA1: hashed value	content of the input data is not valid HMAC-SHA1: hashed value	content of the input data is not valid HMAC-SHA1: hashed value
	encrypt '' by FPE	NULL	NULL	NULL	NULL	NULL	EMPTY
	tokenize '' by any Token element	NULL	NULL	NULL	NULL	NULL	EMPTY
	detokenize tokenized '' by token element that was used to tokenize	NULL	NULL	NULL	NULL	NULL	EMPTY
	decrypt encrypted '' by AES-256 by encryption data element that was used to encrypt	EMPTY	EMPTY	EMPTY	EMPTY	EMPTY	EMPTY
	unprotect protected '' by NoEnc data element that was used to protect	NULL	NULL	NULL	NULL	NULL	EMPTY
	reprotect protected '' by AES-256	Encrypted value	Encrypted value	Encrypted value	Encrypted value	Encrypted value	Encrypted value
	reprotect protected '' by NoEnc	NULL	NULL	NULL	NULL	NULL	EMPTY
	reprotect protected '' by any Token element	NULL	NULL	NULL	NULL	NULL	EMPTY
empty	protect '' by NoEnc	EMPTY	EMPTY	EMPTY	EMPTY	EMPTY	EMPTY
	encrypt '' AES-256	EMPTY	EMPTY	EMPTY	EMPTY	EMPTY	EMPTY

Emptystring setting in pepserver.cfg	Operation with empty string ''	Handling by different Application Protectors on Open Systems					
		AP C	AP NodeJS	AP .Net	AP Java	AP Python	AP Golang
	encrypt '' by FPE	EMPTY	EMPTY	EMPTY	EMPTY	EMPTY	EMPTY
	tokenize '' by any Token element	EMPTY	EMPTY	EMPTY	EMPTY	EMPTY	EMPTY
	decrypt encrypted '' by data element that was used to encrypt	EMPTY	EMPTY	EMPTY	EMPTY	EMPTY	EMPTY
	detokenize tokenized '' by token element that was used to tokenize	EMPTY	EMPTY	EMPTY	EMPTY	EMPTY	EMPTY
	unprotect protected '' by NoEnc data element that was used to protect	EMPTY	EMPTY	EMPTY	EMPTY	EMPTY	EMPTY
	reprotect protected '' by any Enc data element	EMPTY	EMPTY	EMPTY	EMPTY	EMPTY	EMPTY
	reprotect protected '' by any Token element	EMPTY	EMPTY	EMPTY	EMPTY	EMPTY	EMPTY
	reprotect protected '' by NoEnc Token element	EMPTY	EMPTY	EMPTY	EMPTY	EMPTY	EMPTY

Table 12-3: Empty String handling by MPP Data Protectors on Open Systems

empty string setting in pepserver.cfg	Operation with empty string ‘’	Handling by different Protectors on Open Systems		
		Teradata	GPDB	IBM Netezza
NULL (default)	encrypt ‘’ (by all DEs)	NULL	NULL	NULL
	encrypt ‘’ by FPE	‘?’	NULL	NA
	tokenize ‘’	NULL	NULL	NULL

empty string setting in pepserver.cfg	Operation with empty string ‘’	Handling by different Protectors on Open Systems		
		Teradata	GPDB	IBM Netezza
	decrypt encrypted ‘’	NULL	NULL	NULL
encrypt	encrypt ‘’ by AES-128, AES-256, 3DES	encrypted value	encrypted value	encrypted value
	encrypt ‘’ by CUSP	encrypted value	encrypted value	Not supported
	encrypt ‘’ by HMAC-SHA1	encrypted value	encrypted value	N/A
	encrypt by No_ENC	EMPTY	EMPTY	EMPTY
	encrypt ‘’ by FPE	EMPTY	EMPTY	NA
	tokenize ‘’	EMPTY	EMPTY	EMPTY
	decrypt encrypted ‘’	EMPTY	EMPTY	EMPTY
empty	encrypt ‘’ by AES-128, AES-256, 3DES	EMPTY	EMPTY	EMPTY
	encrypt ‘’ by CUSP. HMAC-SHA1	EMPTY	EMPTY	EMPTY
	encrypt ‘’ by FPE	EMPTY	EMPTY	NA
	tokenize ‘’	EMPTY	EMPTY	EMPTY
	decrypt encrypted ‘’	EMPTY	EMPTY	EMPTY

Note:

Due to inconsistent behavior, changing the default behavior for empty strings handling (**emptystring = NULL**) is only recommended for Teradata and DB2.

Emptystring setting affects only ‘’ value. NULL and any other values are not impacted by this setting.

For Database and MPP Database Protectors, a column protected in a table should allow NULL. If a column does not allow NULL, then attempts to encrypt with default **emptystring = NULL** setting will result in error.

Note:

With **emptystring=encrypt** empty string ‘’ will be encrypted by the following data elements: AES-256, AES-128, 3DES, CUSP, NoEnc, HMAC-SHA. For token data elements, the following error is returned: `Invalid input data`.

Users of XC cannot distinguish if a NULL or an empty string is returned from a decrypt operation, that is why with setting **emptystring=NULL**, NULL is returned. NULL is also returned if a user lacks permissions to decrypt the data.

Note:

Before changing the *emptystring* setting in *pepserver.cfg*, it is recommended to decrypt the necessary values using the setting in which they were encrypted.

This may be needed because in some cases you may get "integrity failed” error, for example, on attempt to decrypt a value with **emptystring=encrypt**, encrypted with **emptystring=empty**.

Note:

*1 – If empty strings need to be loaded, then the Pig functions replace the empty strings with null value.

*2– Since HBase does not support *Null* values, it does not support the empty string setting as *Null*.

Table 12-4: Empty String handling by Big Data Protectors on Open Systems

Emptystring setting in pepserver.cfg	Operation with empty string ‘’	Handling by Big Data Protectors on Open Systems							
		MapReduce	Hive	Pig *1	HBase *2	Impala	Spark	Spark SQL	Presto
NULL (default)	protect ‘’ by NoEnc	NULL	NULL	NA	NA	NULL	NULL	NULL	Not supported
	encrypt ‘’ by AES-256, AES-128, 3DES, HMAC-SHAI, CUSP	NULL	NULL	N/A	N/A	NULL	NULL	N/A	Not supported
	encrypt ‘’ by FPE	NULL	NULL	NULL	NULL	NULL	NULL	NULL	Not supported
	tokenize ‘’ by any	NULL	NULL	N/A	N/A	NULL	NULL	NULL	NULL

Emptystring setting in pepserver.cfg	Operation with empty string ‘’	Handling by Big Data Protectors on Open Systems							
		MapReduce	Hive	Pig *1	HBase *2	Impala	Spark	Spark SQL	Presto
	Token Element								
	decrypt encrypted ‘’ by data element that was used to encrypt	NULL	NULL	N/A	N/A	NULL	NULL	NULL	Not supported
	detokenize tokenized ‘’ by token element that was used to protect	NULL	NULL	N/A	N/A	NULL	NULL	NULL	NULL
	unprotect protected ‘’ by NoEnc data element that was used to protect	NULL	NULL	N/A	N/A	NULL	NULL	NULL	Not supported
	reprotect protected ‘’ by any Encryption Data element	NULL	NULL	N/A	N/A	N/A	NULL	N/A	Not supported
	reprotect protected ‘’ by any Token element	NULL	NULL	N/A	N/A	N/A	NULL	NULL	NULL
	reprotect protected ‘’ by NoEnc	NULL	NULL	N/A	N/A	N/A	NULL	NULL	Not supported
encrypt	encrypt ‘’	encrypted value	encrypted value	N/A	encrypted value	encrypted value	encrypted value	encrypted value	Not supported

Emptystring setting in pepserver.cfg	Operation with empty string ‘’	Handling by Big Data Protectors on Open Systems							
		MapReduce	Hive	Pig *1	HBase *2	Impala	Spark	Spark SQL	Presto
	by AES-128, AES-256, 3DES								
	encrypt ‘’ by HMAC-SHA1, CUSP	CUSP: ERROR MESSAGE: 44, The content of the input data is not valid HMAC-SHA1: hashed value	CUSP: ERROR MESSAGE: 44, The content of the input data is not valid HMAC-SHA1: Not supported	N/A	CUSP: ERROR MESSAGE: 44, The content of the input data is not valid HMAC-SHA1: hashed value	CUSP: ERROR MESSAGE: Invalid input data	CUSP: ERROR CODE: 44 is returned HMAC-SHA1: hashed value	CUSP: ERROR MESSAGE: 44, The content of the input data is not valid HMAC-SHA1: Not supported	Not supported
	encrypt ‘’ by FPE	EMPTY	N/A	N/A	EMPTY	EMPTY	EMPTY	N/A	Not supported
	protect ‘’, NoEnc	EMPTY	EMPTY	N/A	EMPTY	EMPTY	EMPTY	EMPTY	Not supported
	tokenize ‘’ by any Token element	NULL	NULL	N/A	NULL	NULL	NULL	NULL	Not supported
	detokenize tokenized ‘’ by token element that was used to tokenize	NULL	NULL	N/A	NULL	NULL	NULL	NULL	Not supported
	decrypt encrypted ‘’ by AES-128, AES-256, 3DES by encryption data element that was used to encrypt	EMPTY	EMPTY	N/A	EMPTY	EMPTY	EMPTY	EMPTY	Not supported
	unprotect protected ‘’	EMPTY	EMPTY	N/A	EMPTY	EMPTY	EMPTY	EMPTY	Not supported

Emptystring setting in pepserver.cfg	Operation with empty string ‘’	Handling by Big Data Protectors on Open Systems							
		MapReduce	Hive	Pig *1	HBase *2	Impala	Spark	Spark SQL	Presto
	by NoEnc data element that was used to protect								
	reprotect protected ‘’ by AES-128, AES-256, 3DES	encrypted value	encrypted value	N/A	N/A	N/A	encrypted value	encrypted value	Not supported
	reprotect protected ‘’ by NoEnc	EMPTY	EMPTY	N/A	N/A	N/A	EMPTY	EMPTY	Not supported
	reprotect protected ‘’ by any Token element	NULL	NULL	N/A	N/A	N/A	NULL	NULL	Not supported
empty	protect ‘’ by NoEnc	EMPTY	EMPTY	N/A	EMPTY	EMPTY	EMPTY	EMPTY	Not supported
	encrypt ‘’ AES-128, AES-256, 3DES, HMAC-SHA1, CUSP	EMPTY	EMPTY	N/A	EMPTY	EMPTY	EMPTY	N/A	Not supported
	encrypt ‘’ by FPE	EMPTY	N/A	N/A	EMPTY	EMPTY	EMPTY	N/A	Not supported
	tokenize ‘’ by any Token element	EMPTY	EMPTY	N/A	EMPTY	EMPTY	EMPTY	EMPTY	EMPTY
	decrypt encrypted ‘’	EMPTY	EMPTY	N/A	EMPTY	EMPTY	EMPTY	N/A	Not supported

Emptystring setting in pepserver.cfg	Operation with empty string ‘’	Handling by Big Data Protectors on Open Systems							
		MapReduce	Hive	Pig *1	HBase *2	Impala	Spark	Spark SQL	Presto
	by data element that was used to encrypt								
	detokenize tokenized ‘’ by token element that was used to tokenize	EMPTY	EMPTY	N/A	EMPTY	EMPTY	EMPTY	EMPTY	EMPTY
	unprotect protected ‘’ by NoEnc data element that was used to protect	EMPTY	EMPTY	N/A	EMPTY	EMPTY	EMPTY	EMPTY	Not supported
	reprotect protected ‘’ by any Enc data element	EMPTY	EMPTY	N/A	N/A	N/A	EMPTY	N/A	Not supported
	reprotect protected ‘’ by any Token element	EMPTY	EMPTY	N/A	N/A	N/A	EMPTY	EMPTY	EMPTY
	reprotect protected ‘’ by NoEnc Token element	EMPTY	EMPTY	N/A	N/A	N/A	EMPTY	EMPTY	Not supported

The following table explains the expected behavior of the Protectors on z/OS.

Table 12-5: Empty String handling by Protectors on z/OS

Emptystring setting in pepserver.cfg	**Operation with empty string ‘’**	**Handling by different Protectors on z/OS**				
		DB2 **UDFs**	**EDITPROC**	**FIELDPROC**	**XC PEP** **XC Client** **XC Lite**	**File Utility**
NULL (default)	encrypt ‘’ (all DEs)	NULL	N/A	NULL	NULL	Not supported
	tokenize ‘’	NULL	N/A	NULL	NULL	Not supported
	decrypt encrypted ‘’	NULL	N/A	NULL	NULL	Not supported
encrypt	encrypt ‘’ (all DE)	encrypted value	N/A	N/A	encrypted value	error message
	tokenize ‘’	error message	N/A	N/A	error message	error message
	decrypt encrypted ‘’	EMPTY	N/A	N/A	NULL	error message
empty	encrypt ‘’ (all DE)	EMPTY	N/A	N/A	NULL	error message
	tokenize ‘’	EMPTY	N/A	N/A	NULL	error message
	decrypt encrypted ‘’	EMPTY	N/A	N/A	NULL	error message

Chapter 13

Appendix D: NULL Handling by Protectors

NULL is neither an empty string nor a zero value. When a variable has no value it is considered to be NULL. When you provide a NULL input, it results in different return values depending on the Protectors. Hence you cannot configure the expected behavior of the Protectors using *pepserver.cfg* configuration file.

The following table explains the behavior of the Protectors when NULL is the input.

Table 13-1: NULL handling by different protectors

Operation with NULL	**NULL Handling by different Protectors**				
	Handling by Database Protectors on Open Systems				
	MSSQL Server	**DB2**			**Oracle**
tokenize NULL by any token element	NULL	‘-’			NULL
detokenize NULL by any token element	NULL	‘-’			NULL
encrypt NULL by any encryption data element	NULL	‘-’			NULL
decrypt encrypted NULL	NULL	‘-’			NULL
encrypt NULL by FPE	NULL	‘-’			NULL
	Handling by Application Protectors on Open Systems				
	AP C	**AP NodeJS**	**AP .Net**	**AP Java**	**AP Python**
tokenize NULL by any token element	NULL	NULL	NULL	NULL	None

Operation with NULL	**NULL Handling by different Protectors**				
detokenize NULL by any token element	NULL	NULL	NULL	NULL	None
encrypt NULL by any encryption data element	NULL	NULL	NULL	NULL	None
encrypt NULL by FPE	NULL	NULL	NULL	NULL	None
decrypt encrypted NULL	NULL	NULL	NULL	NULL	None

	Handling by MPP Database Protectors on Open Systems		
	Teradata	**GPDB**	**IBM Netezza**
tokenize NULL by any token element	‘?’	EMPTY	EMPTY
detokenize NULL by any token element	‘?’	EMPTY	EMPTY
encrypt NULL by any encryption data element	‘?’	EMPTY	EMPTY
encrypt NULL by FPE	‘?’	NULL	NA
decrypt encrypted NULL	‘?’	EMPTY	EMPTY

	Handling by Big Data Protectors on Open Systems							
	MapReduce	**Hive**	**Pig**	**HBase**	**Impala**	**Spark**	**Spark SQL** *1	**Presto**
tokenize NULL by any token element	NULL	NULL	EMPTY	N/A	NULL	NULL	NULL	NULL
detokenize NULL using	NULL	NULL	EMPTY	N/A	NULL	NULL	NULL	NULL

Operation with NULL	NULL Handling by different Protectors							
data element that was used to tokenize								
encrypt NULL by any encryption data element	NULL	NULL	N/A	N/A	NULL	NULL	NULL	Not supported
encrypt NULL by FPE data element	NULL	N/A	N/A	N/A	NULL	NULL	N/A	Not supported
decrypt encrypted NULL using data element that was used to encrypt	NULL	NULL	N/A	N/A	NULL	NULL	NULL	Not supported
reprotect NULL using any token element	NULL	NULL	N/A	N/A	N/A	NULL	NULL	NULL
reprotect NULL using any encryption data element	NULL	NULL	N/A	N/A	N/A	NULL	NULL	NULL

Note:
*1 – Null handling is supported by the Spark SQL protector with the String and Unicode UDFs only.

Note:
Null handling is not supported by the AP Go.

z/OS Protectors do not support Null handling due to the following reasons:

- UDFs do not support NULL handling.
- As EDITPROC handles rows and a NULL row holds no value, NULL handling is not applicable.
- FIELDPROC is never invoked for a NULL value.
- File Utility and z/OS APIs do not support NULL handling.

Chapter 14

Appendix E: Hashing Functions and Examples

Hashing is accomplished by two functions at the Policy Enforcement Point (PEP), an Insert hash function and an Update hash function. Both functions take the same parameters and return a hash value that is always a 160 bit binary value. The difference between the functions is the access rights that they check.

Here is the functions syntax example, applicable to an Oracle database:

```
FUNCTION ins_hash_varchar2(dataelement IN CHAR, cdata IN VARCHAR, SCID IN BINARY_INTEGER) RETURN
RAW;
FUNCTION upd_hash_varchar2(dataelement IN CHAR, cdata IN VARCHAR, SCID IN BINARY_INTEGER) RETURN
RAW;
```

Table 14-1: Functions Syntax Example

Where...	Is...
dataelement	The data element name.
cdata	The data.
SCID	The security ID.

There is no decrypt function since a hash is a checksum and not data.

14.1 Hash Data column size

A hash value is always 160 bits or 20 bytes long regardless of what data it’s calculated on. Basically you should have a table with a binary column of 20 bytes for the hash value.

Here is an example of an Oracle table with hash value instead of name:

```
CREATE TABLE NAMETABLE ( ident NUMBER PRIMARY KEY,
                  name RAW(20));
```

14.2 Using Hashing Triggers and View

You use protection functions in triggers in the same manner as encryption.

Oracle example:

```
CREATE OR REPLACE TRIGGER SCOTT.NAMETABLE_INS
INSTEAD OF INSERT ON SCOTT.NAMETABLE
```

```
FOR EACH ROW
DECLARE
NAME_ RAW(2000) := NULL;

BEGIN
           NAME_:=PTY.INS_HASH_VARCHAR2('HashDE', :new.NAME, 0);

           INSERT INTO SCOTT.NAMETABLE_ENC(IDENT, NAME)
           VALUES(:new.IDENT, NAME_);
END;


CREATE OR REPLACE TRIGGER SCOTT.NAMETABLE_UPD
INSTEAD OF UPDATE ON SCOTT.NAMETABLE
FOR EACH ROW
DECLARE
NAME_ RAW(2000) := NULL;

BEGIN
           PTY.SEL_CHECK('HashDE');

           NAME_:=PTY.UPD_HASH_VARCHAR2('HashDE', :new.NAME, 0);

           IF: old.IDENT = :new.IDENT THEN
                      UPDATE NAMETABLE_ENC SET
                      NAME= NAME_,
                      WHERE IDENT=:old.IDENT;
           ELSE
                      UPDATE NAMETABLE_ENC SET
                      IDENT=:new.IDENT,
                      NAME= NAME_,
                      WHERE IDENT=:old.IDENT;
           END IF;
END;
```

The view selects the hash value directly from the table instead of running a decrypt function. To make this work as a normal trigger/view solution, the binary data type is cast into the original data type. In Oracle it should be VARCHAR2. The data type must be cast to insert data through the view as usual.

```
CREATE OR REPLACE VIEW SCOTT.NAMETABLE(IDENT,
NAME)
AS SELECT IDENT, utl_raw.cast_to_varchar2(NAME))
FROM SCOTT.NAMETABLE_ENC;
```

The application handles the return value, which will now be a 20 byte binary string converted into a character string.

Chapter 15

Appendix F: Codebook Re-shuffling in the Data Security Gateway (DSG)

You can enable the Codebook Re-shuffling in the Data Security Gateway (DSG) for all the tokenization data elements to generate unique tokens for protected values across the tokenization domains. A tokenization domain or a token domain can be defined as a business unit, or a geographical location, or a subsidiary organization where protected data is stored. The data protected by enabling Codebook Re-shuffling cannot be unprotected outside the tokenization domain.

For more information about the Codebook Re-shuffling for the Data Security Gateway (DSG), refer to section *Codebook Re-shuffling* in the *Protegrity Cloud Gateway User Guide 2.5.0.0*.

> **Note:**
> As the Codebook Re-shuffling feature is an advanced functionality, you must contact the Professional Services team for more information about the usage.

Index

E

H

I

P

T