

# **S T R O N G K E Y**

**Tellaro KeyAppliance (KA)  
Demo Client Guide**

**Version 4.0**

## Copyrights and Notices

---

Copyright 2001–2019 StrongAuth, Inc. (d/b/a StrongKey), 20045 Stevens Creek Blvd. Suite 2A, Cupertino, CA 95014, U.S.A. All rights reserved.

StrongAuth, Inc. has intellectual property rights relating to technology embodied in the product that is described in this document. In particular, and without limitation, these intellectual property rights may include one or more U.S. patents or pending patent applications in the U.S. and in other countries.

U.S. Government Rights—Commercial software. Government users are subject to the StrongAuth, Inc. standard license agreement and applicable provisions of the Federal Acquisition Regulations and its supplements.

This distribution may include materials developed by third parties.

StrongAuth, StrongKey, StrongKey Lite, StrongKey CryptoCabinet, StrongKey CryptoEngine, the StrongAuth logo, the StrongKey logo, the StrongKey Lite logo, the StrongKey CryptoCabinet logo and the StrongKey CryptoEngine logo are trademarks or registered trademarks of StrongAuth, Inc. or its subsidiaries in the U.S. and other countries.

Products covered by and information contained in this publication are controlled by U.S. Export Control laws and may be subject to the export or import laws in other countries. Nuclear, missile, chemical or biological weapons or nuclear maritime end uses or end users, whether direct or indirect, are strictly prohibited. Export or reexport to countries subject to U.S. embargo or to entities identified on U.S. export exclusion lists, including, but not limited to, the denied persons and specially designated nationals lists is strictly prohibited.

DOCUMENTATION IS PROVIDED “AS IS” AND ALL EXPRESS OR IMPLIED CONDITIONS, REPRESENTATIONS AND WARRANTIES, INCLUDING ANY IMPLIED WARRANTY OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT, ARE DISCLAIMED, EXCEPT TO THE EXTENT THAT SUCH DISCLAIMERS ARE HELD TO BE LEGALLY INVALID.

---

## Contents

<b>1—Preface.....</b>	<b>1</b>
1.1—Default Paths and Filenames.....	1
1.2—Third-party Website References.....	1
1.3—StrongKey Welcomes Your Comments!.....	1
<b>2—Introduction.....</b>	<b>2</b>
2.1—Regulatory Compliance.....	2
2.2—Architecture.....	4
2.2.1—KAM Encryption Mechanics.....	5
2.2.2—KAM Decryption Mechanics.....	6
2.2.3—Notes on the Mechanics.....	7
2.2.4—KAM Deletion Mechanics.....	8
2.2.5—KAM Search Mechanics.....	8
2.2.6—KAM Entropy Mechanics.....	9
2.2.7—KAM Batch Operations.....	9
2.2.8—KAM Relay Mechanics.....	10
2.2.9—HTTPS Interface.....	12
2.2.10—SOAP Interface.....	12
<b>3—sakaclient.....</b>	<b>13</b>
3.1—Installing the Demo Client Application.....	14
3.2—Displaying Help Options.....	14
3.3—Operations.....	16
3.3.1—Encryption.....	16
3.3.2—Decryption.....	17
3.3.3—Encryption and Decryption.....	18
3.3.4—Failed Encryption and Decryption.....	19
3.3.5—Encryption and Failed Decryption.....	19
3.3.6—Deletion.....	20
3.3.7—Re-encryption after Deletion.....	21
3.3.8—Search.....	21
3.3.9—Encryption 2.....	22
3.3.10—Encryption and Decryption of a String.....	22
3.3.11—Random Numbers from TRNG.....	23
3.3.12—Encrypt and Decrypt with a Stored Key.....	24

<b>4—sakagclient.....</b>	<b>26</b>
4.1—Installing the Demo Client Application.....	26
4.2—Displaying Help Options.....	26
4.3—Encryption (CBC Mode).....	28
4.4—Decryption (CBC Mode).....	30
4.5—Encryption (GCM Mode).....	31
4.6—Decryption (GCM Mode).....	32
<b>5—KMS Client.....</b>	<b>33</b>
5.1—Installing the Demo Client Application.....	34
5.2—Displaying Help Options.....	34
5.3—List Manufacturer.....	37
5.4—Load Key Component.....	39
5.5—Load BDK.....	39
5.6—Generate Initial Key.....	40
5.7—Store ANSI Key (BDK or LTMK).....	40
5.9—Replace ANSI Key (TMK or TPK).....	41
5.10—Update ANSI Key.....	41
5.11—Delete ANSI Key.....	42
5.12—Generate Asymmetric Key.....	42
<b>6—CCS Client.....</b>	<b>43</b>
6.1—Installing the Demo Client Application.....	44
6.2—Prerequisites.....	44
6.3—Displaying Help Options.....	44
6.4—List Manufacturer.....	46
6.5—Generate Swipe Data.....	46
6.6—Get Card Capture Data.....	47
6.7—DUKPT Encrypt.....	48
6.8—DUKPT Decrypt.....	48
<b>7—SEDOS.....</b>	<b>49</b>
7.1—The SEDOS Client Software Distribution.....	51
7.2—Installing the SEDMDB Schema.....	51
7.3—Installing the SEDOS Software Distribution.....	53
7.4—Using the SEDOS Shell Script Manually.....	55

<b>8—skceclient.....</b>	<b>58</b>
<b>8.1—Parameters.....</b>	<b>59</b>
8.1.1—svcinfo.....	59
8.1.2—fileinfo.....	59
8.1.3—Encinfo.....	59
8.1.4—Authzinfo.....	59
8.1.5—storageinfo.....	60
<b>8.2—Encrypt.....</b>	<b>60</b>
8.2.1—Decrypt.....	61
8.2.2—Encrypt to Cloud.....	61
8.2.3—Decrypt from Cloud.....	62
8.2.4—Ping.....	63

# 1–Preface



This document provides information about the *Tellaro KeyAppliance™ (KA) 4.0* Demo Client.


## 1.1–Default Paths and Filenames

The following table describes the default paths and filenames used in this book.

Placeholder	Description
GLASSFISH_HOME	The directory where the Payara Application Server is installed. Default location is <code>STRONGAUTH_HOME</code>
JAVA_HOME	The directory where the Java Development Kit is installed. Default location is <code>STRONGAUTH_HOME</code>
MYSQL_HOME	The directory where the MariaDB Relational Database is installed. Default location is <code>STRONGAUTH_HOME</code>
SAKA_HOME	The directory where Tellaro KeyAppliance™ related files are installed. Default location is <code>STRONGAUTH_HOME/saka</code>
STRONGAUTH_HOME	The directory where Tellaro KeyAppliance™ components are installed. Default location is <code>/usr/local/strongauth</code>
STRONGKEYLITE_HOME	The directory where Tellaro KeyAppliance™ related files are installed. Default location is <code>STRONGAUTH_HOME/strongkeylite</code>

## 1.2–Third-party Website References

Third-party URLs are referenced herein and provide additional, related information.

 **NOTE:** StrongKey is not responsible for third-party websites mentioned in this document. StrongKey does not endorse and is not responsible or liable for any content, advertising, products, or other materials available on or through such sites or resources. StrongKey will not be responsible or liable for any actual or alleged damage or loss caused or alleged to be caused by or in connection with use of or reliance on any such content, goods, or services available on or through such resources.

## 1.3–StrongKey Welcomes Your Comments!

StrongKey is interested in improving its documentation and welcomes your comments and suggestions. To share your comments, please email [info@strongkey.com](mailto:info@strongkey.com) referencing the title of this document in your email with your comments.

# 2—Introduction



StrongKey's *Tellaro KeyAppliance™* (KA)—StrongKey's flagship product—has undergone a radical change in the 4.0 release. KA 4.0 retains all the great features of the 3.0 release, but also adds significant new capabilities, as follows:

1. It updates to the latest releases all underlying components, such as the *Java Virtual Machine (JVM)*, the relational database, the application server, and the cryptographic libraries. While this not only fixes many bugs, it also enables regulatory compliance by addressing residual vulnerabilities from previous releases.
2. It replaces the standard BouncyCastle library with the latest BouncyCastle FIPS library. BC-FIPS is a [Federal Information Processing Standards \(FIPS\)](#) 140-2 level 1 certified module used to perform all cryptography. To manage this, all cryptographic code has been collapsed into a single crypto-module in the code.
3. All public key cryptography has moved from RSA to Elliptic Curve cryptography.
4. Updates support from Trusted Platform Module (TPM) 1.2 to TPM 2.0. TPM 2.0 is a FIPS 140-2 Level 2 certified cryptographic module.

Customers of existing KA (formerly SAKA and SKLES)—1.0, 2.0, or 3.0—with current Support contracts are entitled to free upgrades to the 4.0 release. Please contact us at [support@strongkey.com](mailto:support@strongkey.com) to learn how to upgrade an existing SAKA (or SKLES).

## 2.1—Regulatory Compliance

The *Payment Card Industry Data Security Standard*<sup>1</sup> (PCI DSS) mandates encryption of *Personal Account Numbers (PAN)* when stored on a computerized device. The encryption keys used for cryptographic processing of the credit cards are required to be managed with “appropriate key management” operations as defined in the PCI DSS *Key Management (KM)* section of the *Detailed PCI DSS Requirements and Security Assessment Procedures*, Version 3.2, dated April 2016.

The State of Massachusetts in the USA passed a law in 2008, popularly known as *Standards for the Protection of Personal Information of Residents of the Commonwealth*<sup>2</sup> (201 CMR 17.00) which mandates the encryption of *Personally Identifiable Information (PII)* of the residents of the Commonwealth when stored or maintained on a computer and when transmitted over public networks. The regulation states: *Every person who owns, licenses, stores or maintains personal information about a resident of the Commonwealth shall be in full compliance with 201 CMR 17.00 on or before January 1, 2010.*

Similarly, the State of Washington in the USA amended their breach disclosure law through *House Bill 1149*<sup>3</sup>, “and to permit financial institutions to recoup data breach costs associated with the re-issuance from large businesses and card processors who are negligent in maintaining or transmitting card data.” The amendment, however, states that “Processors, businesses and vendors are not liable under this section if (a) the account information was encrypted at the time of the breach, or (b) ... was certified compliant with the PCI DSS ... and in force at the time of the breach.”

<sup>1</sup> [https://www.pcisecuritystandards.org/security\\_standards/pci\\_dss.shtml](https://www.pcisecuritystandards.org/security_standards/pci_dss.shtml)


<sup>2</sup> <https://www.mass.gov/regulations/201-CMR-17-standards-for-the-protection-of-personal-information-of-residents-of-the>

<sup>3</sup> <http://apps.leg.wa.gov/documents/billdocs/2009-10/Pdf/Bills/Session%20Laws/House/1149-S2.SL.pdf>

StrongKey's KA is a collection of technology (packaged as an appliance for convenience) to assist companies with addressing PCI DSS, 201 CMR 17.00, HB-1149 and similar regulations requiring encryption of sensitive data. It does this in the following ways:

1. Securely generating, storing, using and controlling access to cryptographic keys within the system using a FIPS 140-2 Level 3 certified cryptographic *Hardware Security Module (HSM)* or *Trusted Platform Module (TPM)*. These devices are designed to erase cryptographic key material rather than give it up when they sense they are being attacked. Keys generated on these devices never leave the device unless they are encrypted using other cryptographic keys.
2. Using only one cryptographic algorithm—the *Advanced Encryption System (AES)*—with a choice of 128-bit, 192-bit or 256-bit symmetric keys for the encryption and decryption of PANs or PII.
3. Using only one cryptographic algorithms for generating *Hashed Message Authentication Codes (HMAC)*—while providing a choice of key sizes for the HMAC: the *HmacSHA256* algorithm—with a 256-bit cryptographic key, *HmacSHA224*, *HmacSHA384* and *HmacSHA512* for preserving the integrity of encrypted data (*ciphertext*) in the system.
4. Storing ciphertext on the appliance system—never allowing it to leave KA—while returning a pseudo-number of the PAN, generally known as a “token” as a unique reference for the PAN/PII.

By choosing the strongest algorithm and cryptographic components recommended by PCI DSS and the U.S. *National Institute of Standards and Technology (NIST)*, by localizing all cryptographic processing on KA and by storing ciphertext on the appliance, KA narrows the scope of the application system at risk, and consequently, the scope for the PCI DSS/201 CMR 17.00/HB-1149 audit.

 **NOTE:** It must be emphasized that technology alone is incapable of addressing the KM requirements completely. To ensure compliance, companies implementing KA **must** supplement the technology with appropriate policies and procedures to establish controls over the cryptographic keys in KA. Without such policies and procedures, KA can be improperly implemented and managed, thereby negating the security of the system.



## 2.2—Architecture

KA is a *Java Enterprise Edition 7 (JEE7)* application encapsulated within an appliance and provides secure web services to perform many different cryptographic functions. It consists of the following components:

- A JEE7 Application Server that hosts multiple web service applications
- A Relational Database that stores the ciphertext and accompanying metadata
- A cryptographic Trusted Platform Module or Hardware Security Module that performs cryptographic functions
- A replication architecture that automatically replicates all transactions to every KA node defined within a cluster
- A Lightweight Directory Access Protocol (LDAP) Server for authenticating and authorizing requesters of web services. In the event a site already has an LDAP directory server – such as Active Directory—the KA application can authenticate requesters against this directory server
- A FIDO-enabled web application that enables end users to encrypt/decrypt files while storing cryptographic keys in the KA module of the KA.

In the current release, the KA supports the following versions of underlying components:

Component	Name	Version
Operating System	CentOS Linux (64-bit)	7.5
Java Virtual Machine	Open Java Development Kit	[Latest]
Relational Database	MariaDB RDBMS	10.2.13
JEE7 Application Server	Payara	4.1.2.174
HSM Software	SafeNet Protect Toolkit C	4.3
	Utimaco CryptoServer	4.20.0.4
HSM Java Software	SafeNet Protect Toolkit J	4.3
	Utimaco CryptoServer JCE	4.20.0.4
Replication Software	JeroMQ	0.4.3
LDAP Service	OpenDJ	3.0.0
	Microsoft Active Directory	Windows 2008

In the current release, KA supports the following cryptographic algorithms and sizes. StrongKey has chosen to restrict the algorithms and key sizes to the strongest ones available. As guidelines from PCI and/or NIST evolve, so will KA support the recommended algorithms and key sizes.

Algorithm	Purpose	Size
Elliptic Curve (EC)	Key Encryption	256 bits
	Key Custodian Authentication	256 bits
	Domain Administrator Authentication	256 bits
Advanced Encryption Standard	Data Encryption	128, 192, and 256 bits
Hashed Message Authentication Code	Message Integrity	224, 256, 384 and 512 bits

## 2.2.1—KAM Encryption Mechanics

The *KeyAppliance Module (KAM)* web service application works by having client applications—whether they are e-commerce, payment processing, business intelligence, health care or other applications that deal with sensitive data—send the sensitive data to KA via a standard *Simple Object Access Protocol (SOAP)*-based web service over the *Secure Hyper Text Transfer Protocol (HTTPS)*.

For encrypting sensitive data, the KAM web service call requires four parameters:

Parameter	Explanation
<b>did</b>	The unique Encryption Domain identifier. This is a numeric integer that logically represents the context within which the data is encrypted and tokenized.
<b>username</b>	The username in the encryption domain with authorization to call this web service.
<b>password</b>	The password of the username to authenticate the credential of the requester.
<b>plaintext</b>	The sensitive data that must be encrypted and tokenized.

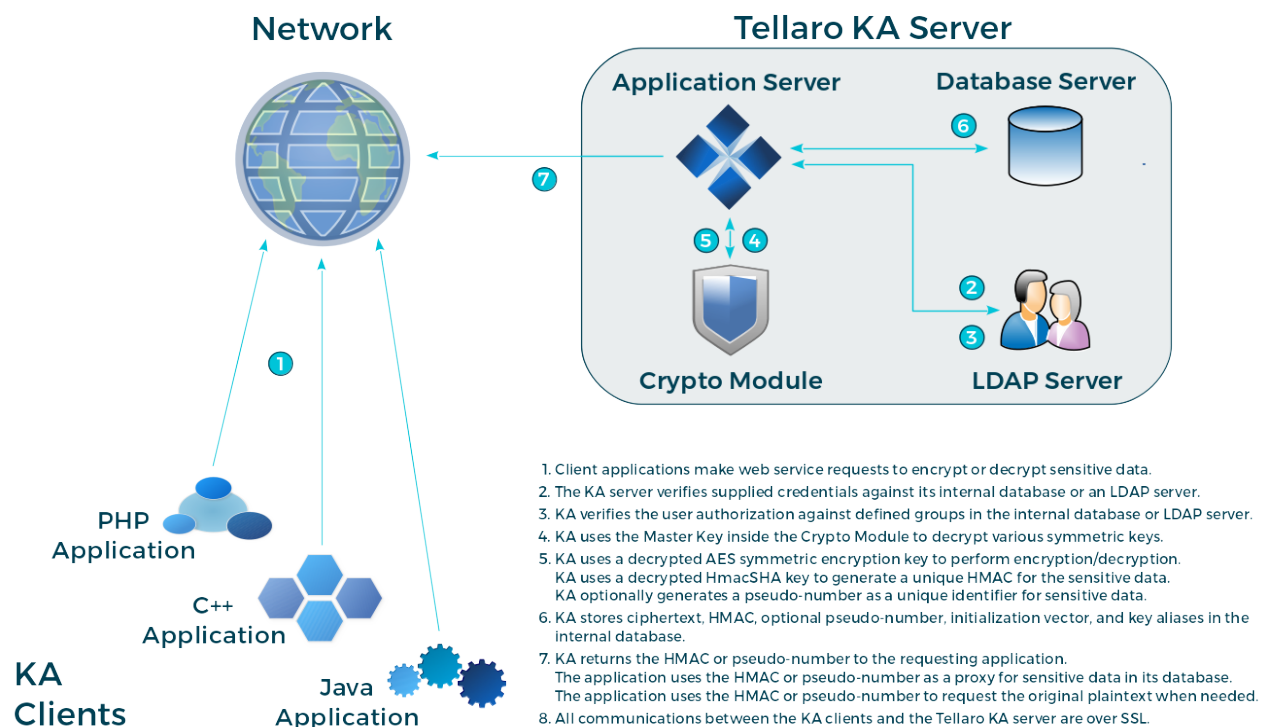
When KA receives the request, it first verifies the credentials presented against its internal database or an LDAP directory server (depending on which is configured), and then determines their authorization to request the encryption service by determining if they belong to an *EncryptionAuthorized* group. Note that if using LDAP, this group and its members must be created in the LDAP directory as a distinct task of the KA installation and configuration process; when using the KA internal database, this is performed automatically.

If the requester is authorized, KA uses an encryption key to encrypt the plaintext. It also uses a separate HMAC key to generate a unique HMAC of the plaintext. After performing these two cryptographic operations, it stores the ciphertext, the HMAC, the key identifiers, and metadata about the request in the RDBMS. The KA *never* stores plaintext in the database or anywhere on the system—plaintext is discarded immediately after the transaction. By default, KA also generates a numerical *Token* that is characteristically similar to the plaintext (16 digits for a credit card number, 9 digits for a social security number, etc.), and can be used as a substitute for the plaintext data in applications.

Upon storing the data, KA replicates the transaction object to all other KA nodes within the same cluster and returns the Token to the requester. The requesting application may use this Token as a unique identifier for the plaintext and store it within its own database. The Token can be configured to be of the identical length as the plaintext data, up to a maximum length of 64 digits.

The replication latency between nodes within a cluster depends on the network capacity between nodes and the saturation of the network at the time of replication. The demo appliance provided by StrongKey on the internet has 1GbE ports and the network has light-to-moderate traffic on it; as a result, transactions have been observed to be replicated from one node to the other in an average of 1–2 seconds.

An illustration of the web service process is shown in the following diagram; to focus on the application's perspective, replication details are not shown:



## 2.2.2—KAM Decryption Mechanics

For decrypting ciphertext, the web service call requires four parameters:

Parameter	Explanation
<b>did</b>	The unique Encryption Domain identifier.
<b>username</b>	The username in the encryption domain with authorization to call this web service.
<b>password</b>	The password of the username to authenticate the credential of the requester.
<b>token</b>	The token—by default, a 16-digit number—referencing the object; given to applications during the original encryption call.

When KA receives the request, it first verifies the credentials presented against its internal database or an optional LDAP directory server and then determines their authorization to request the decryption service by determining if they belong to a *DecryptionAuthorized* group. If using LDAP, this group and its members must be created in the LDAP directory as a distinct task of the KA installation and configuration process; when using the internal database on the Tellaro, this is performed automatically.

If the requester is authorized, KA searches its RDBMS for the Token, determines the identifier of the key that was used to encrypt it originally, and uses that key to decrypt the ciphertext. If the key was rotated at some point due to PCI DSS or other security requirements, KA uses the rotated key to decrypt the ciphertext—applications are neither aware nor concerned about cryptographic key management operations when requesting encryption and/or tokenization services from KA.

After the decryption process, KA retrieves the identifier of the HMAC key originally used by the application and with the HMAC key, recalculates a new HMAC with the just-decrypted plaintext. It then compares the original HMAC (stored after the successful encryption operation) with the freshly calculated HMAC; if the two HMACs are identical, KA knows the decryption process was successful.

KA logs the decryption request (*without* storing or logging the sensitive plaintext), replicates the transaction object to all other KA nodes within the same cluster, and returns the plaintext to the requester.

### 2.2.3—Notes on the Mechanics

1. Two separate keys are used for the encryption and HMAC calculation.
2. The default size of the AES encryption key is 256 bits. However, this can be customized to use either a 128-bit or 192-bit AES key by modifying the KA *properties* (see the KA Configuration chapter in the KA Reference Manual for details).
3. The default size of the HMAC key is 256 bits. However, this can be customized to use either a 224-bit, 384-bit or 512-bit key by modifying the KA *properties* file (see the KA Configuration chapter in the KA Reference Manual for details).
4. The default duration for the usage of the encryption key is one (1) month, while that of the HMAC key is one (1) year. At the start of a new month – starting with the first encryption request past midnight—KA starts using a new encryption key that it generates automatically based on configured policies; a new HMAC key is generated on the first day of a new calendar year. These durations can be customized to use keys on either a *daily*, *weekly*, *monthly*, or an *annual* basis in the KA properties file (see the KA Configuration chapter in the KA Reference Manual for details).

## 2.2.4—KAM Deletion Mechanics

KA can delete encrypted records on demand from its internal database. For deleting ciphertext, the web service call requires four parameters:

Parameter	Explanation
<b>did</b>	The unique Encryption Domain identifier.
<b>username</b>	The username in the encryption domain with authorization to call this web service.
<b>password</b>	The password of the username to authenticate the credential of the requester.
<b>token</b>	The token referencing the object (given to applications during the original encryption call).

When KA receives the request, it first verifies the credentials presented against its internal database or an optional LDAP directory server and then determines their authorization to request the deletion service by determining if they belong to a *DeletionAuthorized* group. If using LDAP, this group and its members must be created in the LDAP directory as a distinct task of the KA installation and configuration process; when using the internal database on the Tellaro, this is performed automatically.

If the requester is authorized, KA searches its RDBMS for the Token; if found, the record is deleted and the deletion logged. The deletion is replicated to other nodes of the cluster before a response is returned to the calling application indicating success or failure.

Once deleted, the original plaintext cannot be returned to any calling application. While the key that encrypted the original plaintext might still be present in the KA—potentially, to decrypt other records encrypted by the same key—a successful call to the deletion web service permanently removes the record from the database. This does not imply that sites using KA may not have other copies of the encrypted record on backup tapes. It remains the site's responsibility to ensure compliance to its data retention policy.

## 2.2.5—KAM Search Mechanics

Certain applications may have a need to search the KA internal database to determine if a specific piece of sensitive data exists. KA provides a web service method for performing this task. The method requires four parameters:

Parameter	Explanation
<b>did</b>	The unique Encryption Domain identifier. This is a numeric integer that logically represents the context within which the data is encrypted and tokenized.
<b>username</b>	The username in the encryption domain with authorization to call this web service.
<b>password</b>	The password of the username to authenticate the credential of the requester.
<b>plaintext</b>	The sensitive data for which to search.

When KA receives the request, it verifies the credentials presented against its internal database or an optional LDAP directory server and then determines their authorization to request the search service by determining if they are a member of a *SearchAuthorized* group. Note that if using LDAP, this group and its members must be created in the LDAP directory as a distinct task of the installation process of the KA; when using the internal database on the Tellaro, this group is created automatically.

If the requester is authorized, KA converts the plaintext to an HMAC and searches its RDBMS for the HMAC; if found, the Token is returned to the caller and the search is logged and replicated to other nodes. A non-NULL return value to the calling application indicates the search was successful.

## 2.2.6—KAM Entropy Mechanics

Certain applications may have a need for true random numbers generated from a certified hardware-based *random number generator (RNG)*. Since the KA includes such a hardware RNG, it provides a web service for requesting and receiving true random numbers from its underlying RNG. The method requires four parameters:

Parameter	Explanation
<b>did</b>	The unique Encryption Domain identifier. This is a numeric integer that logically represents the context within which the data is encrypted and tokenized.
<b>username</b>	The username in the encryption domain with authorization to call this web service.
<b>password</b>	The password of the username to authenticate the credential of the requester.
<b>bytes</b>	The number of bytes of entropy requested, returned as Base64-encoded text.

When KA receives the request, it verifies the credentials presented against its internal database or an optional LDAP directory server. It then determines their authorization to request the entropy service by determining if they belong to an *EncryptionAuthorized* group. If using LDAP, this group and its members must be created in the LDAP directory as a distinct task of the KA installation process; when using the internal database on the Tellaro, this group is created automatically.

If the requester is authorized, KA gathers the requested number of bytes of entropy from its cryptographic hardware module, Base64-encodes them, then returns the encoded bytes to the calling application. While most applications are likely to Base64-decode the encoded bytes—perhaps to seed a *Pseudo Random Number Generator (PRNG)* in their application/system—some may choose to use the Base64-encoded text as-is—perhaps as truly random passwords or session identifiers for web sessions, etc.

## 2.2.7—KAM Batch Operations

Some business operations require cryptographic operations on millions of records periodically. While the standard web services are capable of receiving said millions of requests individually, the operations can be made significantly more efficient by submitting the input data in an *eXtensible Markup Language (XML)* file and performing the operation on the appliance without authenticating and authorizing each request (except for the first), and by eliminating the network round trips for each web service call.

The Tellaro provides four (4) web service methods for encrypting, decrypting, deleting and searching for sensitive data using XML-based files in batch mode.

The XML input file conforms to the *SKLESBatchInput* element, defined in the KA *XML Schema Definition (XSD)* file supplied with the appliance. Any number of records may be processed through batch files more efficiently; the only limitation to the number of records in such a batch file would be the appliance's operating system limit on the file size.

**NOTE:** An XML input file with one million 16-digit credit card numbers (and conforming to the KA XSD) uses a little less than 40 MB of space, or approximately 25,000 records per MB of space. Based on this, a 1 GB file can store 25 million input records. An input file with a maximum file size limitation of 8 Terabytes can accommodate 200 billion credit card numbers.

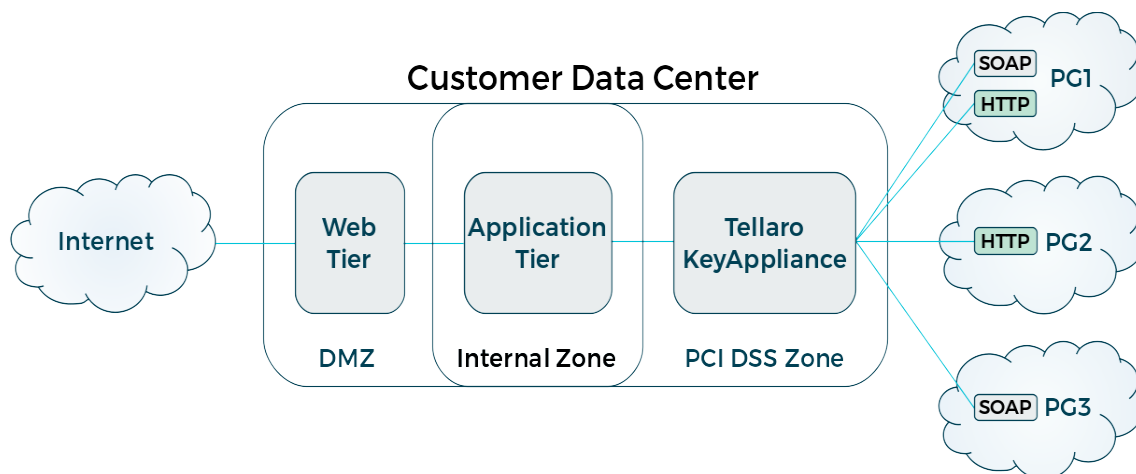
The appliance processes the input file in batch mode: it performs just a single authentication and authorization check, a single verification of the Encryption Domain's status, and proceeds to execute the requested cryptographic operation for each record in the input file. It writes the result to a different XML file corresponding to the *SKLESBatchOutput* element in the KA XSD. The input and output files may be transferred to and from the appliance using the *Secure File Transfer Protocol (SFTP)*, secure NFS or SAMBA over TLS.

## 2.2.8—KAM Relay Mechanics

To minimize decryption of sensitive PAN data within the application network, KA supports a “Relay” web service method permitting applications to relay a transaction to a *payment gateway (PG)*. This is accomplished through an HTTPS POST method, or a SOAP action within an HTTPS POST method.

The business benefit of using the Relay web service is that the application dealing with the PG does not need to decrypt credit card numbers before sending transaction to the gateway—the KA performs this service (functioning like a proxy) on behalf of the application and, thus, reduces or eliminates the need for decrypting credit card numbers. However, to relay transactions, KA must have direct network connectivity to the PG's web server, either over the internet or through a *virtual private network (VPN)*.

Here is a sample representation of how an infrastructure might look when configured to use the Relay web service to multiple payment gateways:



In this configuration, the site has:

- A web tier in the *demilitarized zone (DMZ)* receiving customer transactions from the internet
- An application tier with servers and databases representing the business logic and data
- A PCI zone containing the Tellaros
- Three payment gateways: PG1, which offers both an HTTPS and SOAP interface to their transaction gateway; PG2, which only offers an HTTPS interface; and PG3, which only offers a SOAP interface.



One may connect any number of payment gateways to the Tellaro, as long as the gateways offer standard HTTPS or SOAP interfaces to their services.

When KA receives the request, it verifies the credentials presented against its internal database or an optional LDAP directory server, then determines their authorization to request the relay service by determining if they are a member of two groups—*RelayAuthorized* group and *DecryptionAuthorized*. This is the only service requiring authorized requesters to be part of two groups; this is because the security of the appliance requires that tokens are decrypted and substituted for actual sensitive information and relayed to the payment gateway.

To use the relay web service, applications that normally communicate with the PG must be modified to communicate with the KA which requires seven (7) parameters to perform its task. The seven parameters are listed here:

Parameter	Explanation
<b>did</b>	The unique Encryption Domain identifier. This is a numeric integer that logically represents the context in which your data is encrypted and tokenized. You must specify the DID to ensure the relay service can not only authenticate the request correctly, but also decrypt any tokens before relaying the transaction.
<b>username</b>	The username within the encryption domain that has authorization to call the web service. At most sites, this is a service credential used by the application communicating with KA.
<b>password</b>	The password of the username within the encryption domain to authenticate the credential of the requester.
<b>relayurl</b>	The URL of the payment gateway that receives transactions. The appliance checks to see if the supplied URL matches—completely or partially—URLs configured and authorized for use by the appliance for relaying transactions. For example, if <a href="https://test.authorize.net">https://test.authorize.net</a> is configured as an authorized URL for this appliance, supplying the URL <a href="https://test.authorize.net/gateway/transact.dll">https://test.authorize.net/gateway/transact.dll</a> in this parameter will allow the transaction to be relayed to the gateway (assuming all other checks pass). However, <a href="https://test.authorize.com/gateway/transact.dll">https://test.authorize.com/gateway/transact.dll</a> will not.
<b>relayprotocol</b>	The web service only accepts HTTP or SOAP based relay requests; as such, this parameter <i>must</i> specify either HTTP or SOAP (uppercase) depending on the interface the PG offers.
<b>relayencoding</b>	While future versions of the web service will support additional encoding schemes, the appliance currently accepts only UTF-8 in this parameter.
<b>relaycontent</b>	This parameter carries XML content that conforms to the schema defined in <i>SKLESRelaySchema.xsd</i> . This schema definition—supplied with the software that implements this service—defines the syntax of the transaction that is translated by the KA and relayed to the payment gateway.

Depending on the type of protocol specified in the *relayprotocol* parameter, the appliance builds a standard HTTPS POST message—or one encapsulating a SOAP request—and posts it to the specified URL. As part of relaying the transaction, the appliance can decrypt any tokens specified in the *relaycontent* parameter and substitute decrypted content for the tokens before posting it to the payment gateway's web site

The appliance waits for a response from the gateway and transmits the response back to the calling application without interpreting the response. All relay transactions are logged in the application server's logs (however, sensitive data is never logged).



Since the Relay web service works on standard SOAP over HTTPS, any programming environment that supports these two protocols can consume the service. StrongKey supplies two sample Java clients that show how to consume the service using the HTTPS and the SOAP protocols.

## 2.2.9—HTTPS Interface

A few notes about the HTTPS interface for the Relay web service:

1. The Relay service only performs HTTP POSTs; GETs are not supported at this time.
2. The service allows for specifying any number of HTTP headers, HTTP parameters, and KA tokens that need decrypting and substituting in the relay request to the gateway.
3. All headers, parameters, and tokens are specified in XML elements that *must* conform to the supplied `SKLESRelaySchema.xsd`. If in doubt about your XML, test your sample XML with *xmllint* (on the Linux platform) against the XSD file. Fix any errors before sending the XML to the appliance.
4. The appliance will *not* print any sensitive decrypted/detokenized information in the server log.

## 2.2.10—SOAP Interface

A few notes about the SOAP interface for the Relay web service:

1. The SOAP message sent to the Tellaro for the relay request *embeds* another SOAP envelope containing the message to be relayed to the payment gateway. This might be confusing initially, but is acceptable to the appliance. Just make sure that samples of XML created for testing pass validation tests using *xmllint* against the XSD defined in `SKLESRelaySchema.xsd`.
2. The Relay service only performs HTTP POSTs; GETs are not supported at this time.
3. The service allows for specifying any number of HTTP headers; HTTP parameters are *not* supported in this interface.
4. The appliance will not print any sensitive decrypted/detokenized information in the server log.

# 3—sakaclient



To test the KA web services, the appliance comes with a Java-based client application to test the web service operations of the KA Module. The Java client—called `sakaclient.jar`—can be used to call web service operations on a DEMO appliance on the internet—[demo4.strongkey.com](https://demo4.strongkey.com)—or on your own Tellaro cluster within your network.

The demo KA uses software identical to that of the appliances delivered to customers. This allows application development teams to verify code without having to wait for an internal KA implementation. Once verified against the KA Demo Client, the identical code will work against your internal KA implementation without changing a single line of code. The only difference will be the URL of the host and the parameters passing at runtime. We recommend parameterizing the URL of the appliances being called; this will make it easier to point applications to the KA during deployment to production.

The client application is a simple Java application that calls the KA web service to perform one of many operations; it simulates what applications must do to call the *EncryptionService* on the Tellaro. While your client application will certainly be different based on the business functions it implements, the programming language, and the application model it uses, ultimately it *must* perform the same web service functions as the test client application bundled with KA. Because the client application focuses only on testing the implementation of the KA service, it is ideal for ensuring the correctness of the implementation.

The currently supported operations of the Demo Client are:

1. Encrypt simulated credit card numbers.
2. Decrypt the ciphertext for previously encrypted credit card numbers.
3. Encrypt and decrypt a credit card number within a single transaction.
4. Delete a credit card number from the cluster.
5. Search for a credit card number on the cluster.
6. Return bytes of entropy generated on a *True Random Number Generator (TRNG)*.
7. Submit a job to encrypt credit card numbers in batch mode.
8. Submit a job to decrypt credit card numbers in batch mode.
9. Submit a job to delete credit card numbers in batch mode.
10. Submit a job to search for credit card numbers in batch mode.
11. Relay a payment transaction to [authorize.net](https://authorize.net) using HTTPS POST (you will need your own account and access keys to [authorize.net](https://authorize.net) to perform this test).
12. Encrypt a plaintext using a stored symmetric key.
13. Decrypt a ciphertext using a stored symmetric key.

The client application distinguishes between these operations to allow sites to test different levels of authorization for the web services:

1. Users who are authorized to only encrypt.
2. Users who are authorized to only decrypt.
3. Users who are authorized to encrypt *and* decrypt.
4. Users who are authorized to only delete.

5. Users who are authorized to only search.
6. Users who are authorized to only relay transactions to a payment gateway.
7. Users who are authorized to perform all operations.
8. Users who are *NOT* authorized to perform any operation.

## 3.1—Installing the Demo Client Application

StrongKey distributes the demo application as a .ZIP file. After unzipping the *democlients* distribution, the *sakaclient* executable and source will be in a folder called *sakaclient*.

The *sakaclient.jar* file contains pre-built executable code. The *sakaclient-src.zip* file contains the source for this executable as a project folder. Build the executable .JAR file from scratch or modify the client to integrate into a custom application.

Change directory to the *sakaclient* directory in the location where the *democlients* are installed. Once in that directory, the .JAR file may be run without specifying its location. For the rest of this document, we will assume that *sakaclient.jar* is in your current directory.

## 3.2—Displaying Help Options

Type the following command to verify the client application is executable; also to show a list of operations it can perform and the parameters it expects for each operation:

```
java -jar sakaclient.jar
```

You will see the following output:

```

C:\Users\StrongAuth\Desktop\sakaclient>java -jar sakaclient.jar

Usage: java -jar sakaclient.jar https://<host:port> <did> <username> <password> E <last-8-digits-of-PAN> <# of iterations>
java -jar sakaclient.jar https://<host:port> <did> <username> <password> EE <16-digits-of-PAN> <# of iterations>
java -jar sakaclient.jar https://<host:port> <did> <username> <password> ES <string>
java -jar sakaclient.jar https://<host:port> <did> <username> <password> D <hmac/psn>
java -jar sakaclient.jar https://<host:port> <did> <username> <password> B <last-8-digits-of-PAN> <# of iterations>
java -jar sakaclient.jar https://<host:port> <did> <username> <password> BB <16-digits-of-PAN> <# of iterations>
java -jar sakaclient.jar https://<host:port> <did> <username> <password> L <hmac/psn>
java -jar sakaclient.jar https://<host:port> <did> <username> <password> S <16-digit-PAN>
java -jar sakaclient.jar https://<host:port> <did> <username> <password> R <# of bytes to retrieve>

java -jar sakaclient.jar https://<host:port> <did> <username> <password> GE <gpktoken> <plaintext> <algorithm> <encoding> [iv] [aad]
java -jar sakaclient.jar https://<host:port> <did> <username> <password> GD <gpktoken> <ciphertext> <algorithm> <encoding> [iv] [aad]
java -jar sakaclient.jar https://<host:port> <did> <username> <password> BE <input-filename>
java -jar sakaclient.jar https://<host:port> <did> <username> <password> BD <input-filename>
java -jar sakaclient.jar https://<host:port> <did> <username> <password> BL <input-filename>
java -jar sakaclient.jar https://<host:port> <did> <username> <password> BS <input-filename>

C:\Users\StrongAuth\Desktop\sakaclient>

```

The explanation of the parameters to the demo client application is as follows:

Parameter	Explanation
https://<host:port>	<a href="https://demo4.strongkey.com">https://demo4.strongkey.com</a>
<did>	The unique Encryption Domain identifier. This was provided to you separately by StrongKey as your unique domain for testing.
<username>	The username in the encryption domain with authorization to call the web service. Usernames were provided separately by StrongKey for testing.
<password>	The password of the username above. These passwords were provided separately by StrongKey for testing.

Parameter		Explanation
E	Encryption	Encryption of 16-digit PANs where you only need to specify the last 8 digits of the PAN on the command line; the first 8 digits are programmed into the client application as a convenience for quick testing.
EE	Encryption	Encryption of 16-digit PANs where you specify all 16 digits of the PAN on the command line.
ES	Encryption	Encryption of a string of indefinite length.
D	Decryption	Decryption of a token.
B	Both Encryption and Decryption	Encryption and decryption of 16-digit PANs where you specify the last 8 digits of the PAN on the command line; the first 8 digits are programmed into the client application as a convenience for quick testing.
BB	Both Encryption and Decryption	Encryption and decryption of 16-digit PANs where you specify all 16 digits of the PAN on the command line.
L	Deletion	Deletion of an object based on the specified token.
S	Search	Search for a 16-digit PAN where you specify all 16 digits.
R	Entropy	Gather a certain number of bytes of entropy from the KA's <i>True Random Number Generator (TRNG)</i> .
BE	Batch Encryption	Encryption of PANs defined in an XML file conforming to the <code>SKLES.xsd</code> schema; results are placed in an XML file containing the original PAN and the token associated with it.
BD	Batch Decryption	Decryption of tokens defined in an XML file conforming to the <code>SKLES.xsd</code> schema; results are in an XML file with the original token and resulting PAN.
BL	Batch Deletion	Deletion of PANs based on tokens defined in an XML file conforming to the <code>SKLES.xsd</code> schema; results are placed in an XML file containing the token and a result value of <i>True</i> or <i>False</i> .
BS	Batch Search	Searches for PANs defined in an XML file conforming to the <code>SKLES.xsd</code> schema; results are placed in an XML file containing the original PAN and the token associated with it if it exists or <i>NULL</i> if it does not.
GE	GPK Encrypt	Encrypts a plaintext using the key specified by the <i>Group Public Key (GPK)</i> token.
GD	GPK Decrypt	Decrypts a ciphertext using the key specified by the GPK token.
<last-8-digits-of-PAN>		The demo client is programmed with the first 8 digits of a sample credit card number—11112222; only the last 8 digits are required for the test. However, these last 8 digits <i>must</i> be between 10000000 and 99999999.
<16-digits-of-PAN>		With this option you specify 16 digits (or any other number of your choice); KA does not modify the supplied value, processing the value as is.
<hmac/psn>		For decryption and deletion operations, the client expects the full HMAC or Pseudo Number (a.k.a. Token).
<# of iterations>		The Demo Client can perform multiple operations in each run of the client by incrementing the <i>last-8-digits-of-PAN</i> value and calling the web service again; specify the number of iterations between 1 and 99999999.
<input-filename>		The name of the XML input file containing the information to be transformed by the batch functions.

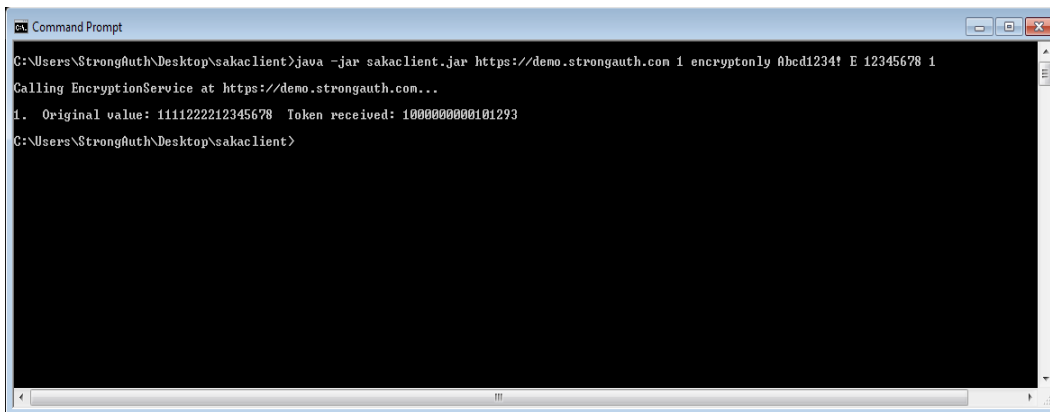
## 3.3—Operations

### 3.3.1—Encryption

To encrypt a test PAN, **type the following command**. In this example the URL parameter is <https://demo4.strongkey.com>, the domainID is the numeral 1, the username is *encryptonly* and the password is *Abcd1234!*. Since this example uses the *E* option, specify only the last 8 digits of the test PAN—the first 8 are specified by the *sakaclient.jar* program. The last parameter indicates to only execute one iteration of this transaction. Had we chosen to specify more than one transaction, the Java program would have incremented the specified 8-digit value in each transaction for the required number of transactions:

```
java -jar sakaclient.jar https://demo4.strongkey.com 1 encryptonly  
Abcd1234! E 12345678 1
```

The following image shows the successful return of a token (1000000000101192) when the operation returns. The actual token returned by your demo encryption domain may be different, but in principle, they resemble the value returned.



If you see an error message that resembles the following, it implies the JDK needs updating. You must use JDK 8 Update 112 or higher to make a web service connection to StrongKey's Demo Client when using Java.

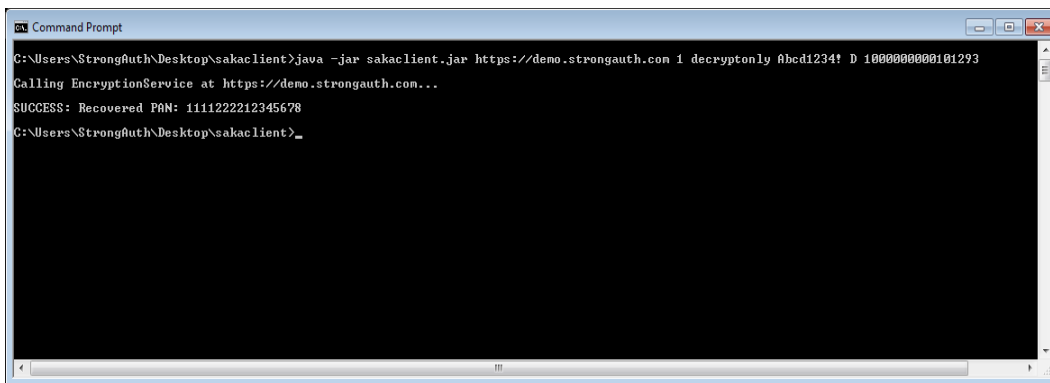
```
Exception in thread "main" javax.xml.ws.WebServiceException: Failed to access the WSDL at:  
https://demo4.strongkey.com/strongkeyliteWAR/EncryptionService?wsdl. It failed with:  
sun.security.validator.ValidatorException: PKIX path building failed:  
sun.security.provider.certpath.SunCertPathBuilderException: unable to find valid certification path to requested  
target.  
at com.sun.xml.internal.ws.wsdl.parser.RuntimeWSDLParser.tryWithMex(RuntimeWSDLParser.java:151)  
at com.sun.xml.internal.ws.wsdl.parser.RuntimeWSDLParser.parse(RuntimeWSDLParser.java:133)  
at com.sun.xml.internal.ws.client.WSServiceDelegate.parseWSDL(WSServiceDelegate.java:254)  
at com.sun.xml.internal.ws.client.WSServiceDelegate.<init>(WSServiceDelegate.java:217)  
at com.sun.xml.internal.ws.client.WSServiceDelegate.<init>(WSServiceDelegate.java:165)  
at com.sun.xml.internal.ws.spi.ProviderImpl.createServiceDelegate(ProviderImpl.java:93)  
at javax.xml.ws.Service.<init>(Service.java:56)  
at com.strongauth.strongkeylite.web.EncryptionService.<init>(EncryptionService.java:79)  
at strongkeyliteClient.Main.main(Main.java:109)
```

### 3.3.2—Decryption

To test decryption, call KA with the *D* option and use the token from your encryption test as the parameter. You will also be using a different username—*decryptonly*—to demonstrate the use of a different service privilege, as shown below.

```
java -jar sakaclient.jar https://demo4.strongkey.com 1 decryptonly  
Abcd1234! D 1000000000101293
```

The following image shows the successful return of the original PAN (1111222212345678) when the operation returns. The operation may take a little while because cryptographic keys may need to be loaded from the internal database and decrypted before the actual encrypted object can be decrypted. However, once the keys are decrypted, they are cached for a short duration (~5 minutes); if you repeat the same decryption operation immediately, you'll notice an immediate response.



**NOTE:** The first web service request for encryption or decryption within a user's session will see a long delay because of the following factors:

- The latency of the Internet
- The verification of the user's authorization (which causes the decryption of a hierarchy of keys in the hardware module to verify the user's password)
- The decryption of the symmetric key before it is available for use
- Finally, the actual encryption/decryption of the PAN data

A key-caching feature on the KA (5 minutes by default) enables subsequent operations to see significant improvements in performance. However, at the end of the default caching period, the keys are flushed from memory and the first cryptographic operation after the flush will see a similar delay as keys are decrypted and loaded into cache memory. The caching window can be configured for your business needs.

From a performance point of view, the current generation of KA, using a TPM, delivers approximately 200–220 *web service operations per second (WSOPS)* when multi-threaded client applications make requests. In a clustered environment where all nodes in the cluster may service client applications, the overall WSOPS will be higher than for an individual KA, but will be difficult to predict in advance without understanding the nature of the network and applications in question.

StrongKey has provided nine (9) credentials—usernames and passwords—for your test encryption domain on the DEMO appliance. These credentials can be used by the Demo `sakaclient.jar` application to test the authentication and authorization capabilities of the appliance. The privileges of these credentials for cryptographic web services are as follows:

Username	Encryption Authorized	Decryption Authorized	Deletion Authorized	Relay Authorized	Search Authorized
encryptonly	Yes	No	No	No	No
decryptonly	No	Yes	No	No	No
encryptdecrypt	Yes	Yes	No	No	No
deleteonly	No	No	Yes	No	No
relay	No	Yes	No	Yes	No
searchonly	No	No	No	No	Yes
all	Yes	Yes	Yes	Yes	Yes
none	No	No	No	No	No
pinguser	No	Yes	No	No	No

### 3.3.3—Encryption and Decryption

To test how *encryptdecrypt* can encrypt and decrypt—execute the following command in a Command Prompt (or Shell) window:

```
java -jar sakaclient.jar https://demo4.strongkey.com 1 encryptdecrypt Abcd1234! B 33331001 5
```

You are essentially asking the KA hosted on [demo4.strongkey.com](https://demo4.strongkey.com) to perform five (5) encryption web service transactions of PANs, starting with 1111222233331001 through 1111222233331005 and, upon receiving the Token, to decrypt them immediately by calling the *decryption* web service. The output of the command, when successful, is shown below:

```

C:\Users\StrongAuth\Desktop\sakaclient>java -jar sakaclient.jar https://demo.strongauth.com 1 encryptdecrypt Abcd1234! B 33331001 5
Calling EncryptionService at https://demo.strongauth.com...
1. Original value: 1111222233331001 Token received: 1000000000101259
SUCCESS: Original and recovered PANs are identical!
2. Original value: 1111222233331002 Token received: 1000000000101255
SUCCESS: Original and recovered PANs are identical!
3. Original value: 1111222233331003 Token received: 1000000000101256
SUCCESS: Original and recovered PANs are identical!
4. Original value: 1111222233331004 Token received: 1000000000101257
SUCCESS: Original and recovered PANs are identical!
5. Original value: 1111222233331005 Token received: 1000000000101258
SUCCESS: Original and recovered PANs are identical!
C:\Users\StrongAuth\Desktop\sakaclient>

```



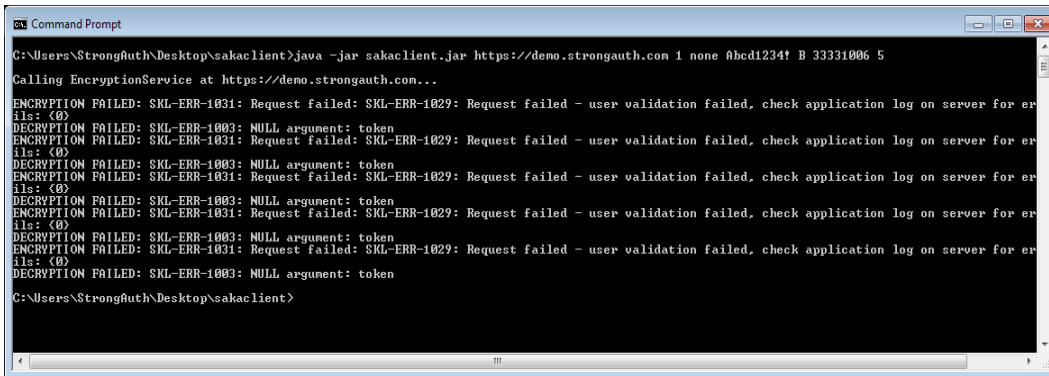
### 3.3.4—Failed Encryption and Decryption

To test how the user *none* can neither encrypt nor decrypt—execute the following command, where you are asking KA to perform five (5) encryption web service transactions of PANs, starting with 1111222233331006 through 1111222233331010 and, upon receiving the Token, to decrypt them immediately by calling the decryption web service.

However, because the user *none* has no authorization to perform any operation, the operation will fail with a response of “Invalid user”. The decryption operation also fails, but shows a different error message: “NULL argument: token.”

When the encryption operation failed due to lack of authorization, instead of a token being returned, the `sakaclient.jar` program received an error message. However, the client program was designed to re-send the response as a parameter to the decryption operation. Since the sanity checks on the KA determined this was not a normal token, it fails with the appropriate message.

```
java -jar sakaclient.jar https://demo4.strongkey.com 1 none Abcd1234! B 33331006 5
```

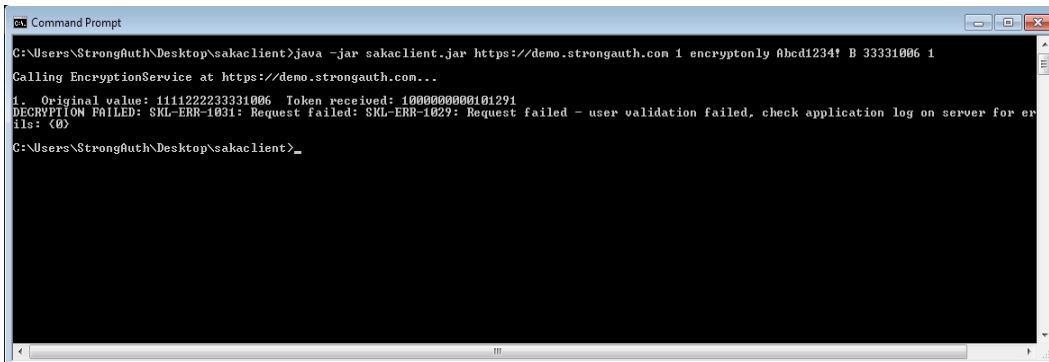


```
Command Prompt
C:\Users\StrongAuth\Desktop\sakaclient>java -jar sakaclient.jar https://demo.strongauth.com 1 none Abcd1234! B 33331006 5
Calling EncryptionService at https://demo.strongauth.com...
ENCRYPTION FAILED: SKL-ERR-1031: Request failed: SKL-ERR-1029: Request failed - user validation failed, check application log on server for er
ils: {}
DECRYPTION FAILED: SKL-ERR-1003: NULL argument: token
ENCRYPTION FAILED: SKL-ERR-1031: Request failed: SKL-ERR-1029: Request failed - user validation failed, check application log on server for er
ils: {}
DECRYPTION FAILED: SKL-ERR-1003: NULL argument: token
ENCRYPTION FAILED: SKL-ERR-1031: Request failed: SKL-ERR-1029: Request failed - user validation failed, check application log on server for er
ils: {}
DECRYPTION FAILED: SKL-ERR-1003: NULL argument: token
ENCRYPTION FAILED: SKL-ERR-1031: Request failed: SKL-ERR-1029: Request failed - user validation failed, check application log on server for er
ils: {}
DECRYPTION FAILED: SKL-ERR-1003: NULL argument: token
ENCRYPTION FAILED: SKL-ERR-1031: Request failed: SKL-ERR-1029: Request failed - user validation failed, check application log on server for er
ils: {}
DECRYPTION FAILED: SKL-ERR-1003: NULL argument: token
C:\Users\StrongAuth\Desktop\sakaclient>
```

### 3.3.5—Encryption and Failed Decryption

The following example shows how *encryptonly* can encrypt but not decrypt:

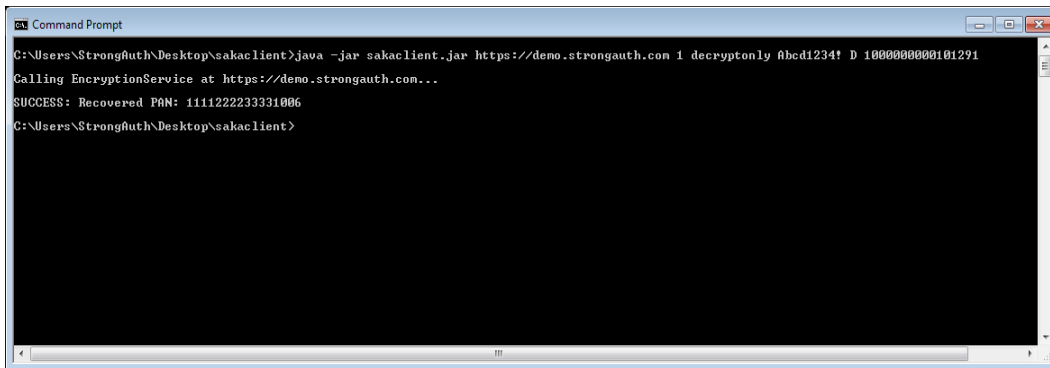
```
java -jar sakaclient.jar https://demo4.strongkey.com 1 encryptonly Abcd1234! B 33331006 1
```



```
Command Prompt
C:\Users\StrongAuth\Desktop\sakaclient>java -jar sakaclient.jar https://demo.strongauth.com 1 encryptonly Abcd1234! B 33331006 1
Calling EncryptionService at https://demo.strongauth.com...
1. Original value: 1111222233331006 Token received: 1000000000101291
DECRYPTION FAILED: SKL-ERR-1031: Request failed: SKL-ERR-1029: Request failed - user validation failed, check application log on server for er
ils: {}
C:\Users\StrongAuth\Desktop\sakaclient>
```



However, if you copy and paste the returned token into the next example using the *decryptonly* credential for the operation, you'll see the decryption succeed:

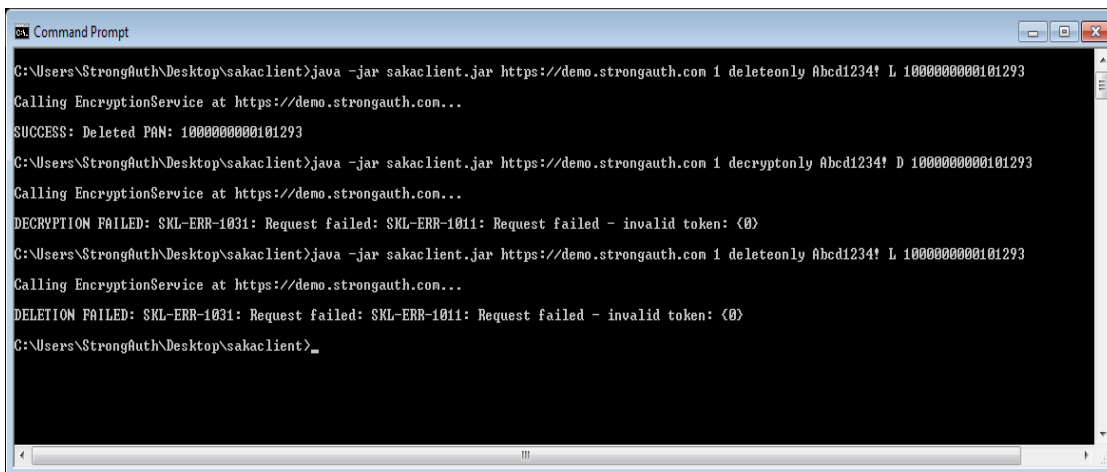


```
C:\Users\StrongAuth\Desktop\sakaclient>java -jar sakaclient.jar https://demo.strongauth.com 1 decryptonly Abcd1234! D 1000000000101291
Calling EncryptionService at https://demo.strongauth.com...
SUCCESS: Recovered PAN: 1111222233331006
C:\Users\StrongAuth\Desktop\sakaclient>
```

### 3.3.6—Deletion

To delete an object from KA, send the request as follows using a token returned by an earlier encryption operation. After a successful deletion, try decrypting the same token to verify the deletion was successful—you will see an error message returned. You can also attempt deleting the same token again; this should also return an error.

```
java -jar sakaclient.jar https://demo4.strongkey.com 1 deleteonly
Abcd1234! L 1000000000101293
java -jar sakaclient.jar https://demo4.strongkey.com 1 decryptonly
Abcd1234! D 1000000000101293
java -jar sakaclient.jar https://demo4.strongkey.com 1 deleteonly
Abcd1234! L 1000000000101293
```



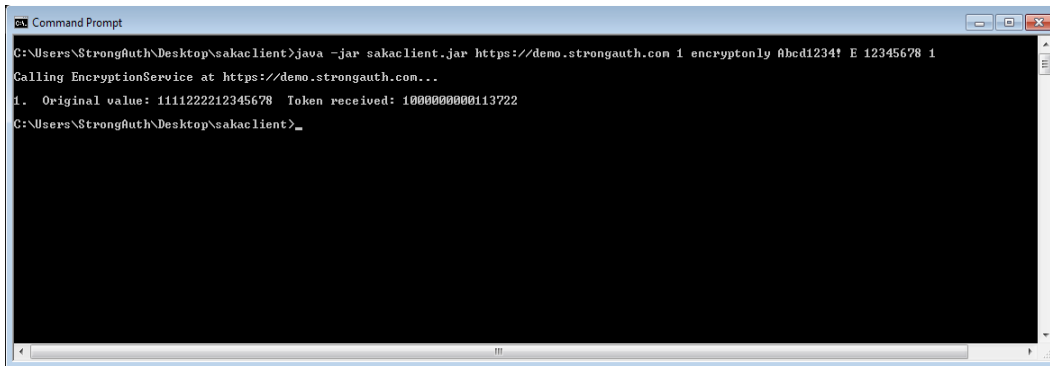
```
C:\Users\StrongAuth\Desktop\sakaclient>java -jar sakaclient.jar https://demo.strongauth.com 1 deleteonly Abcd1234! L 1000000000101293
Calling EncryptionService at https://demo.strongauth.com...
SUCCESS: Deleted PAN: 1000000000101293
C:\Users\StrongAuth\Desktop\sakaclient>java -jar sakaclient.jar https://demo.strongauth.com 1 decryptonly Abcd1234! D 1000000000101293
Calling EncryptionService at https://demo.strongauth.com...
DECRYPTION FAILED: SKL-ERR-1031: Request failed: SKL-ERR-1011: Request failed - invalid token: {0}
C:\Users\StrongAuth\Desktop\sakaclient>java -jar sakaclient.jar https://demo.strongauth.com 1 deleteonly Abcd1234! L 1000000000101293
Calling EncryptionService at https://demo.strongauth.com...
DELETION FAILED: SKL-ERR-1031: Request failed: SKL-ERR-1011: Request failed - invalid token: {0}
C:\Users\StrongAuth\Desktop\sakaclient>
```

### 3.3.7—Re-encryption after Deletion

The following example shows how a PAN (encrypted in [Section 3.3.1](#) and deleted in [Section 3.3.6](#)), when re-encrypted, gets a completely new token number—1000000000000124.

This is because, once a token number is assigned to a sensitive data object, it cannot be re-used again. However, if the sensitive data has not been deleted from the encrypted records in the appliance, the same sensitive data will result in the same token number:

```
java -jar sakaclient.jar https://demo4.strongkey.com 1 encryptonly  
Abcd1234! E 12345678 1
```

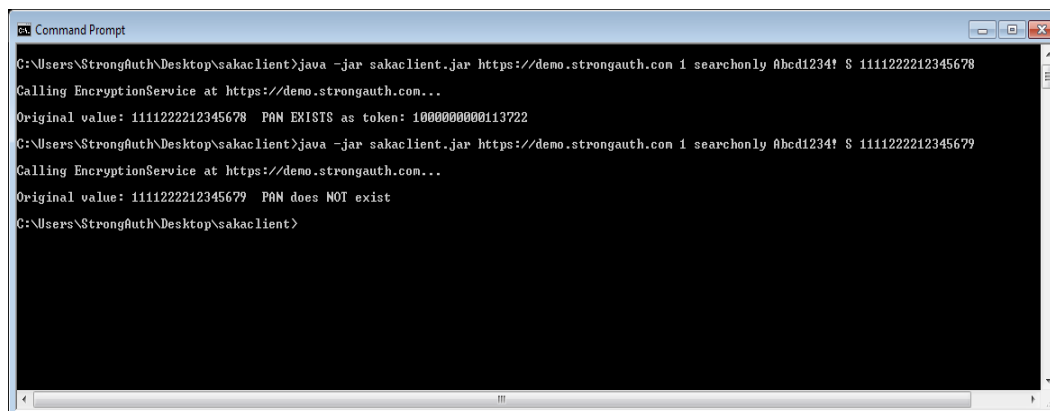


```
Command Prompt
C:\Users\StrongAuth\Desktop\sakaclient>java -jar sakaclient.jar https://demo.strongauth.com 1 encryptonly Abcd1234! E 12345678 1
Calling EncryptionService at https://demo.strongauth.com...
1. Original value: 1111222212345678 Token received: 1000000000113722
C:\Users\StrongAuth\Desktop\sakaclient>
```

### 3.3.8—Search

To test the search web services, encrypt a test PAN on the appliance first; it should return a token. Search for the PAN using the same plaintext data; it will return the same token as from the encryption step. Finally, search for a PAN which is not encrypted; it should return a response indicating the PAN does not exist. The commands and output are shown here:

```
java -jar sakaclient.jar https://demo4.strongkey.com 1 searchonly  
Abcd1234! S 1111222212345678  
  
java -jar sakaclient.jar https://demo4.strongkey.com 1 searchonly  
Abcd1234! S 1111222212345679
```

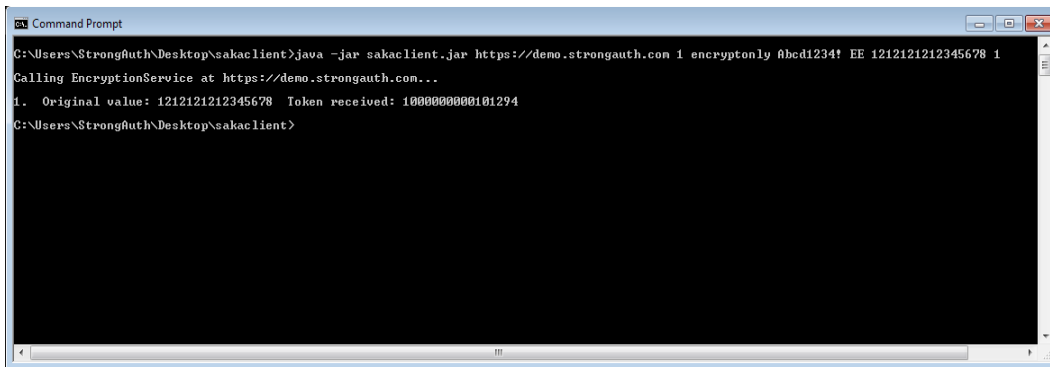


```
Command Prompt
C:\Users\StrongAuth\Desktop\sakaclient>java -jar sakaclient.jar https://demo.strongauth.com 1 searchonly Abcd1234! S 1111222212345678
Calling EncryptionService at https://demo.strongauth.com...
Original value: 1111222212345678 PAN EXISTS as token: 1000000000113722
C:\Users\StrongAuth\Desktop\sakaclient>java -jar sakaclient.jar https://demo.strongauth.com 1 searchonly Abcd1234! S 1111222212345679
Calling EncryptionService at https://demo.strongauth.com...
Original value: 1111222212345679 PAN does NOT exist
C:\Users\StrongAuth\Desktop\sakaclient>
```

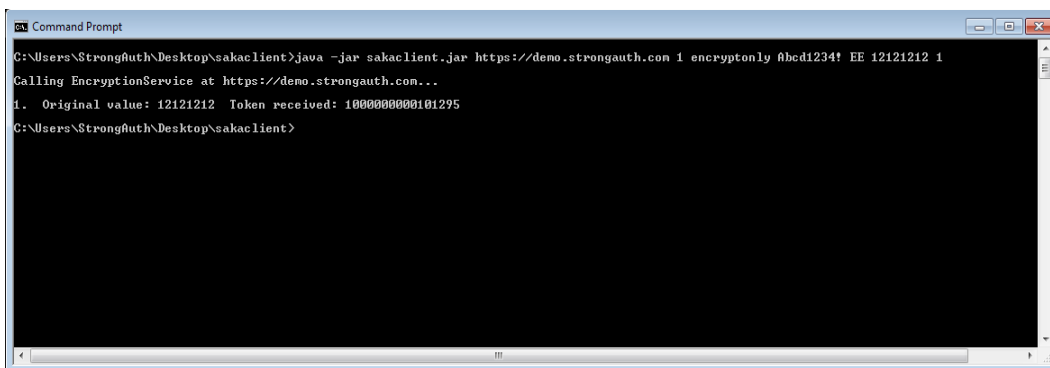
### 3.3.9—Encryption 2

While the *E* (encrypt) option allows you to test quickly by specifying only the last 8 digits on the command line, the *EE* option allows testing encryption with numbers between 1 and 19 digits long. The following two examples show successful encryptions of a 16-digit and an 8-digit number:

```
java -jar sakaclient.jar https://demo4.strongkey.com 1 encryptdecrypt  
Abcd1234! EE 1212121212345678 1
```



```
java -jar sakaclient.jar https://demo4.strongkey.com 1 encryptdecrypt  
Abcd1234! EE 12121212 1
```

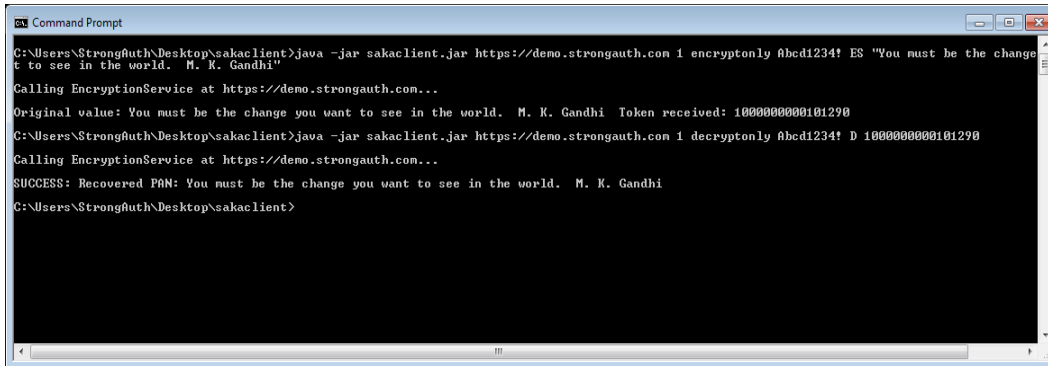


### 3.3.10—Encryption and Decryption of a String

So far, the *sakaclient* has encrypted numeric values that might represent *Primary Account Numbers (PANs)* for credit card holders. The next example shows how to encrypt a string of arbitrary length. The *KA Module (KAM)* can encrypt strings of up to 10,000 characters; as such, the entire string—"You must be the change you want to see in the world. M. K. Gandhi" is encrypted and stored on KA exactly like the PANs in previous examples. The resulting token can be used to decrypt the entire string as is shown in the same figure:

```
java -jar sakaclient.jar https://demo4.strongkey.com 1 encryptonly  
Abcd1234! ES "You must be the change you want to see in the world. M. K.  
Gandhi"
```

```
java -jar sakaclient.jar https://demo4.strongkey.com 1 decryptonly  
Abcd1234! D 1000000000101290
```



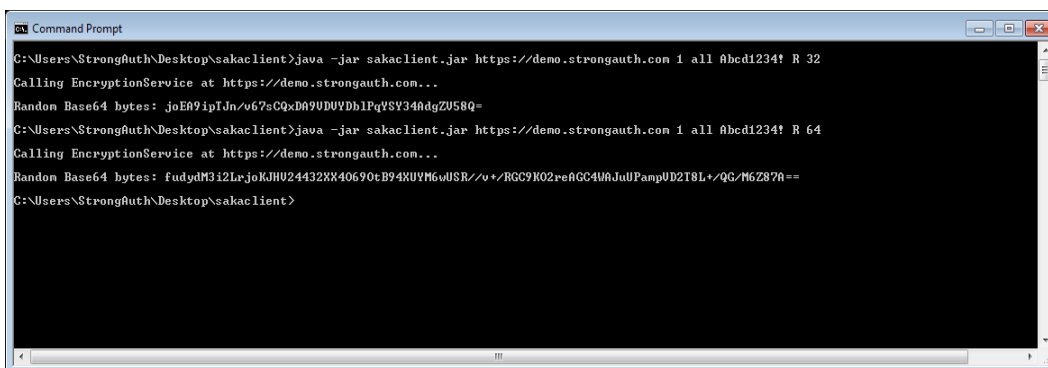
```
Command Prompt
C:\Users\StrongAuth\Desktop\sakaclient>java -jar sakaclient.jar https://demo.strongauth.com 1 encryptonly Abcd1234! ES "You must be the change  
t to see in the world. M. K. Gandhi"
Calling EncryptionService at https://demo.strongauth.com...
Original value: You must be the change you want to see in the world. M. K. Gandhi Token received: 1000000000101290
C:\Users\StrongAuth\Desktop\sakaclient>java -jar sakaclient.jar https://demo.strongauth.com 1 decryptonly Abcd1234! D 1000000000101290
Calling EncryptionService at https://demo.strongauth.com...
SUCCESS: Recovered P&N: You must be the change you want to see in the world. M. K. Gandhi
C:\Users\StrongAuth\Desktop\sakaclient>
```

Since the KAM can encrypt anything up to 10,000 characters, applications can store a wide variety of information: JSON, XML, Base64-encoded check images, and/or QR codes (as long as they are fewer than 10,000 characters in length).

### 3.3.11—Random Numbers from TRNG

While the KAM can protect many different kinds of information, you may occasionally have a need for entropy (random numbers/bytes) generated on a certified TRNG. All Tellaros are outfitted with a certified TRNG as a standard component. A new web service—entropy—allows requests for varying lengths of entropy—up to a maximum of 1024 bytes per web service request—from KA, as the following two examples show. The result may be Base64-decoded and used to seed local *Pseudo-Random Number Generators (PRNGs)*, used as extremely complex passwords for password resets, or any other purpose where applications need random data.

```
java -jar sakaclient.jar https://demo4.strongkey.com 1 all Abcd1234! R 32  
java -jar sakaclient.jar https://demo4.strongkey.com 1 all Abcd1234! R 64
```

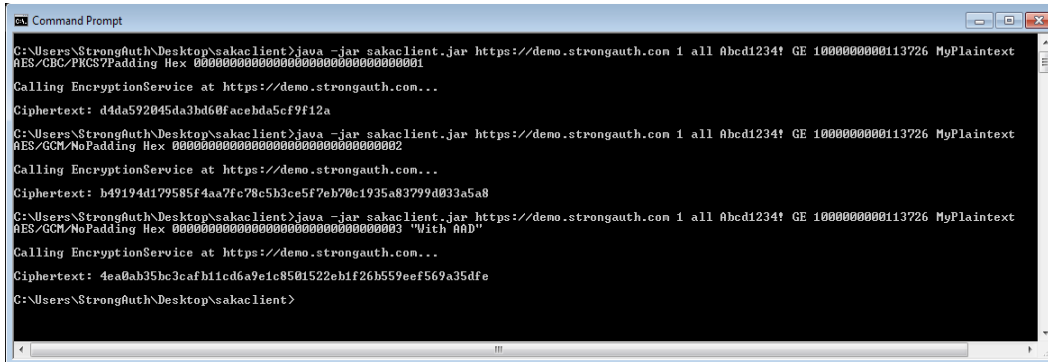


```
Command Prompt
C:\Users\StrongAuth\Desktop\sakaclient>java -jar sakaclient.jar https://demo.strongauth.com 1 all Abcd1234! R 32
Calling EncryptionService at https://demo.strongauth.com...
Random Base64 bytes: joEa9ipTJn/v67sCQxDa9UDUYDh1PqVSV34AdgZU58Q=
C:\Users\StrongAuth\Desktop\sakaclient>java -jar sakaclient.jar https://demo.strongauth.com 1 all Abcd1234! R 64
Calling EncryptionService at https://demo.strongauth.com...
Random Base64 bytes: fudydM3i2LrJoKJHU24432XX40690tB94XUYM6wUSR/v+/RGC9K02reAGC4UaJuUPampUD2I8L+/QG/M6Z87a==
C:\Users\StrongAuth\Desktop\sakaclient>
```

### 3.3.12—Encrypt and Decrypt with a Stored Key

1. As a prerequisite to these commands, a *General Public Key (GPK)* must be stored using the *kmsclient*.
2. Using a stored GPK, the appliance can perform many types of encryption and decryption using keys previously tokenized on the appliance. The web service supports AES keys in ECB, CBC, OFB, CFB, and GCM modes. In ECB and CBC mode, the web service supports *ZeroBytePadding*, *PKCS7Padding*, *TBCPadding*, *X9.23Padding*, *ISO7816-4Padding*, *ISO10126-2Padding*, and *NoPadding* (when the plaintext input is the correct block size). In GCM mode, optional *Additional Authenticated Data (AAD)* can be provided which will be included in the MAC generated over the ciphertext. This can be used to cryptographically bind an unencrypted message to a ciphertext. The following examples use the GPK token 1000000000113726 which has previously been stored.

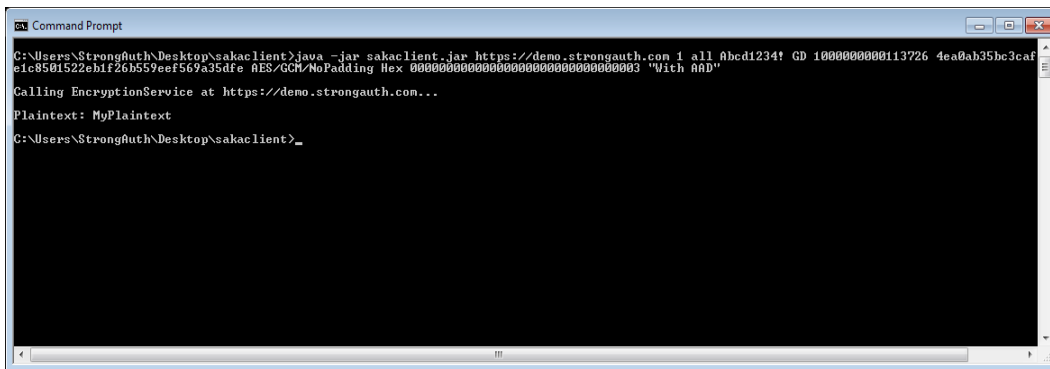
```
java -jar sakaclient.jar https://demo4.strongkey.com 1 all Abcd1234! GE 1000000000113726 MyPlaintext AES/CBC/PKCS7Padding Hex 00000000000000000000000000000001
java -jar sakaclient.jar https://demo4.strongkey.com 1 all Abcd1234! GE 1000000000113726 MyPlaintext AES/GCM/NoPadding Hex 00000000000000000000000000000002
java -jar sakaclient.jar https://demo4.strongkey.com 1 all Abcd1234! GE 1000000000113726 MyPlaintext AES/GCM/NoPadding Hex 00000000000000000000000000000003 "With AAD"
```



```
Command Prompt
C:\Users\StrongAuth\Desktop\sakaclient>java -jar sakaclient.jar https://demo.strongauth.com 1 all Abcd1234! GE 1000000000113726 MyPlaintext AES/CBC/PKCS7Padding Hex 00000000000000000000000000000001
Calling EncryptionService at https://demo.strongauth.com...
Ciphertext: d4da592045da3bd6f0facebda5cf9f12a
C:\Users\StrongAuth\Desktop\sakaclient>java -jar sakaclient.jar https://demo.strongauth.com 1 all Abcd1234! GE 1000000000113726 MyPlaintext AES/GCM/NoPadding Hex 00000000000000000000000000000002
Calling EncryptionService at https://demo.strongauth.com...
Ciphertext: b49194d179585f4aa7fc70c5b3ce5f7eb70c1935a83799d033a5a8
C:\Users\StrongAuth\Desktop\sakaclient>java -jar sakaclient.jar https://demo.strongauth.com 1 all Abcd1234! GE 1000000000113726 MyPlaintext AES/GCM/NoPadding Hex 00000000000000000000000000000003 "With AAD"
Calling EncryptionService at https://demo.strongauth.com...
Ciphertext: 4ea0ab35bc3cafb11cd6a9e1c8501522eb1f26b559eef569a35dfe
C:\Users\StrongAuth\Desktop\sakaclient>
```

- The following example shows how to decrypt one of the ciphertexts generated with a GPK key.

```
java -jar sakaclient.jar https://demo4.strongkey.com 1 all Abcd1234! GD 10000000000113726 4ea0ab35bc3cafb11cd6a9c92f0b0f1cb937871ba4f7244baf4e17 AES/GCM/NoPadding Hex 00000000000000000000000000000003 "With AAD"
```



In this manner, the *sakaclient* can be used to test various KAM web services.

# 4—sakagclient



While the *sakaclient* demonstrates most of the common web services of the *KeyAppliance Module (KAM)*, the *sakagclient* primarily demonstrates how to encrypt arbitrary length strings and text content using the newer *Galois Counter Mode (GCM)* algorithm. GCM itself is not a unique feature of KA, but the cryptographic key, once generated and escrowed on the KAM by the client application, can encrypt using various algorithms.

The *sakagclient* depends on the same web services the *sakaclient* does; applications consuming the KAM's web services require similar privileges on the KA as *sakagclient*. The currently supported operations of the *sakagclient* are:

- Encrypt a string provided on the command line
- Decrypt the ciphertext for previously encrypted strings
- Encrypt and decrypt a string within a single transaction

## 4.1—Installing the Demo Client Application

StrongKey distributes the demo application as a .ZIP file. After unzipping the *democlients* distribution, the *sakagclient* executable and source will be contained within a folder called *sakagclient*.

The *sakagclient.jar* file contains pre-built executable code. The *sakagclient-src.zip* file contains the source for this executable as a project folder. You can build the executable .JAR file from scratch if you wish, or you can modify the client to integrate into your own application.

Change directory to the *sakagclient* directory in the location where the *democlients* are installed. Once you've changed to that directory, you may execute the JAR file without specifying its location. For the rest of this document, we will assume that *sakagclient.jar* is in your current directory.

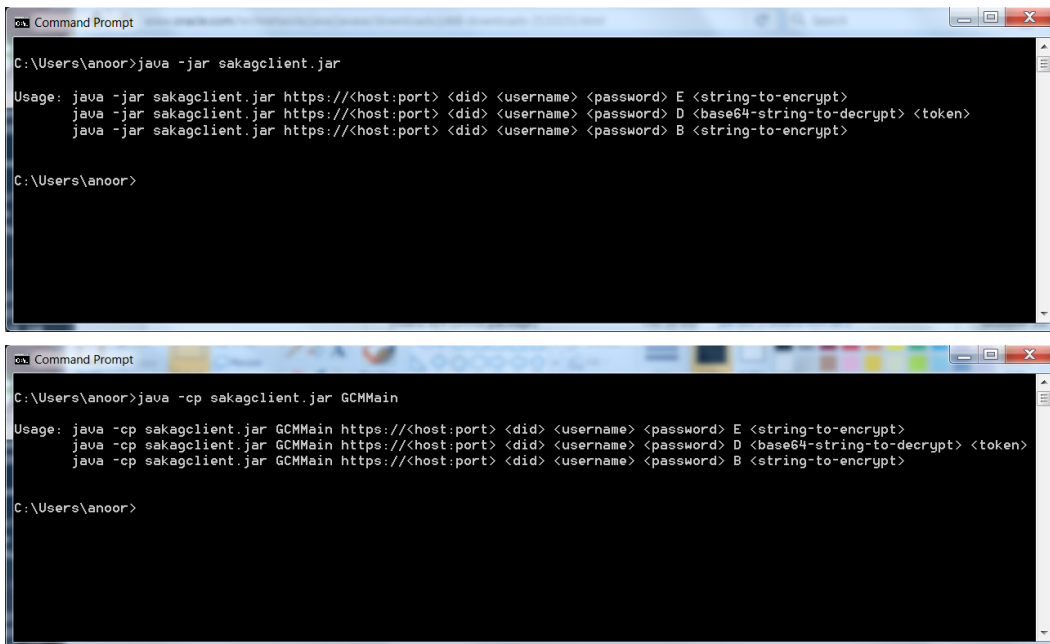
## 4.2—Displaying Help Options

Type the following commands to verify that the client application is executable. It also shows a helpful prompt of the operations it can perform and the parameters it expects for each operation.

The second command—containing GCM Main—reflects a second executable program within the .JAR file that implements the GCM mode for encryption. Since the default executable program uses the CBC mode of operation, the GCM version must be called slightly differently (as shown below). The options for both programs are identical.

```
java -jar sakagclient.jar
java -cp sakagclient.jar GCMMain
```

You will see the following output:



```
C:\Users\anoor>java -jar sakagclient.jar

Usage: java -jar sakagclient.jar https://<host:port> <did> <username> <password> E <string-to-encrypt>
       java -jar sakagclient.jar https://<host:port> <did> <username> <password> D <base64-string-to-decrypt> <token>
       java -jar sakagclient.jar https://<host:port> <did> <username> <password> B <string-to-encrypt>

C:\Users\anoor>

C:\Users\anoor>java -cp sakagclient.jar GCMMain

Usage: java -cp sakagclient.jar GCMMain https://<host:port> <did> <username> <password> E <string-to-encrypt>
       java -cp sakagclient.jar GCMMain https://<host:port> <did> <username> <password> D <base64-string-to-decrypt> <token>
       java -cp sakagclient.jar GCMMain https://<host:port> <did> <username> <password> B <string-to-encrypt>

C:\Users\anoor>
```

An explanation of the *sakagclient.jar* Demo Client parameters follows:

Parameter		Explanation
<b>https://&lt;host:port&gt;</b>		Use <a href="https://demo4.strongkey.com/">https://demo4.strongkey.com/</a> for your testing.
<b>&lt;did&gt;</b>		The unique Encryption Domain identifier. This was provided separately by StrongKey as a unique domain for testing.
<b>&lt;username&gt;</b>		The username in the encryption domain with authorization to call the web service. These usernames were provided separately by StrongKey for testing.
<b>&lt;password&gt;</b>		The password of the username within the encryption domain with authorization to call the web service. These passwords were provided separately by StrongKey for testing.
<b>The demo client supports the following operations:</b>		
<b>E</b>	<b>Encryption</b>	Encryption of 16-digit PANs where only the last 8 digits of the PAN are required on the command line; the first 8 digits are programmed into the client application as a convenience for quick testing.
<b>D</b>	<b>Decryption</b>	Decryption of a token.
<b>B</b>	<b>Both Encryption and Decryption</b>	Encryption and decryption of 16-digit PANs where you specify the last 8 digits of the PAN on the command line; the first 8 digits are programmed into the client application as a convenience for quick testing.
<b>&lt;string-to-encrypt&gt;</b>		A UTF-8 string of arbitrary length. If the string has blank spaces or special punctuation marks, please enclose the string in double quotes and place a backslash in front of the special punctuation mark to escape it from being processed by the Linux shell (if using Linux to test this tool).
<b>&lt;token&gt;</b>		For decryption, the client expects the Token that was returned by the KAM when encrypting the string.



## 4.3—Encryption (CBC Mode)

To encrypt a test string, type the following commands—separately. In this example the URL parameter is <https://demo4.strongkey.com>, the *domainID* is the numeral 1, the *username* is **all** and the password is **Abcd1234!**. The string being encrypted is enclosed within double quotes:

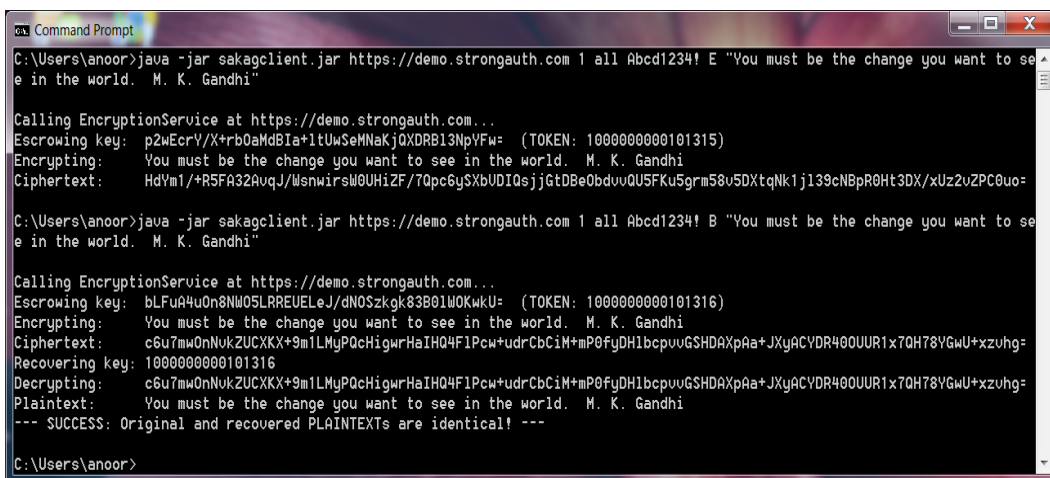
```
java -jar sakagclient.jar https://demo4.strongkey.com 1 all Abcd1234! E "You must be the change you want to see in the world. M. K. Gandhi"
java -jar sakagclient.jar https://demo4.strongkey.com 1 all Abcd1234! B "You must be the change you want to see in the world. M. K. Gandhi"
```

The top half of the image below shows the sequence of activities performed by *sakagclient*: a new AES encryption key was generated by the client (shown as a Base64-encoded string on the first line) and escrowed on the KAM, which returned the token 1000000000101315. Using the token as the *initialization vector (IV)* for the CBC mode of operation, *sakagclient* encrypts the string within double quotes and displays the Base64-encoded ciphertext on the last line of the output.

The bottom half of the results in [3.4—Decryption \(CBC Mode\)](#) shows a variation of the same command, but this time the command has specified the *B* option, which encrypts the supplied string, then immediately decrypts the ciphertext in the same operation and compares the decrypted plaintext with the original to determine if they are the same.

The second command also generates a new AES key, escrows it—resulting in a new token: 1000000000101316—recovers the escrowed key from the KAM, and then decrypts the ciphertext to compare with the original. As the output shows, the original and decrypted plaintext are identical.

The KAM has the ability to escrow cryptographic keys in this manner—TDES, AES, RSA, ECDSA. Almost any cryptographic (or binary) object that can be Base64-encoded to a UTF-8 string of length less than 10,000 characters can be encrypted and stored on KA as a secure vault. KA have been successfully used to store more than 50 million objects in Production use at customer sites.



```
C:\Users\anoor>java -jar sakagclient.jar https://demo.strongauth.com 1 all Abcd1234! E "You must be the change you want to see in the world. M. K. Gandhi"

Calling EncryptionService at https://demo.strongauth.com...
Escrowing key: p2wEcrY/X+rb0aMdBIa+ItUwSeMNaKjQXDRB13NpYFw= (TOKEN: 1000000000101315)
Encrypting: You must be the change you want to see in the world. M. K. Gandhi
Ciphertext: HdYm1/+R5FA32AuqJ/WsnwirsW0UHiZF/7Qpc6ySXbUDIqsjGtDBe0bduvQU5FKuSgrmS8v5DXtqNk1j139cNBpR0Ht3DX/xUz2vZPC0uo=

C:\Users\anoor>java -jar sakagclient.jar https://demo.strongauth.com 1 all Abcd1234! B "You must be the change you want to see in the world. M. K. Gandhi"

Calling EncryptionService at https://demo.strongauth.com...
Escrowing key: bLFuA4u0n8NW0SLRREUELeJ/dNOSzkgk83B01WOKwkU= (TOKEN: 1000000000101316)
Encrypting: You must be the change you want to see in the world. M. K. Gandhi
Ciphertext: c6u7mw0nNukZUCXX+9m1LMypQcHigwrHaIHQ4F1Pcw+udrCbCiM+mp0fyDH1bcpuvGSHDAXpAa+JXyACVDR400UUR1x7QH78YgWU+xzvhg=
Recovering key: 1000000000101316
Decrypting: c6u7mw0nNukZUCXX+9m1LMypQcHigwrHaIHQ4F1Pcw+udrCbCiM+mp0fyDH1bcpuvGSHDAXpAa+JXyACVDR400UUR1x7QH78YgWU+xzvhg=
Plaintext: You must be the change you want to see in the world. M. K. Gandhi
--- SUCCESS: Original and recovered PLAINTEXTS are identical! ---

C:\Users\anoor>
```

If you see an error message similar to the following, it implies an outdated version of JDK. You must use *JDK 8 Update 112 or higher* to make a web service connection to StrongKey's Demo Client when using Java.

```
Exception in thread "main" javax.xml.ws.web serviceException: Failed to
access the WSDL at:
https://demo4.strongkey.com/strongkeyliteWAR/EncryptionService?wsdl. It
failed with:
  sun.security.validator.ValidatorException: PKIX path building failed
  sun.security.provider.certpath.SunCertPathBuilderException: unable to find
  valid certification path to requested target.
  at
com.sun.xml.internal.ws.wsdl.parser.RuntimeWSDLParser.tryWithMex(RuntimeWSDLP
arser.java:151)
  at
com.sun.xml.internal.ws.wsdl.parser.RuntimeWSDLParser.parse(RuntimeWSDLParser
.java:133)
  at
com.sun.xml.internal.ws.client.WSServiceDelegate.parseWSDL(WSServiceDelegate.
java:254)
  at
com.sun.xml.internal.ws.client.WSServiceDelegate.<init>(WSServiceDelegate.jav
a:217)
  at
com.sun.xml.internal.ws.client.WSServiceDelegate.<init>(WSServiceDelegate.jav
a:165)
  at
com.sun.xml.internal.ws.spi.ProviderImpl.createServiceDelegate(ProviderImpl.j
ava:93)
  at javax.xml.ws.Service.<init>(Service.java:56)
  at
com.strongauth.strongkeylite.web.EncryptionService.<init>(EncryptionService.j
ava:79)
  at strongkeyliteClient.Main.main(Main.java:109)
```

If you see a different error that resembles the following, it implies that a .JAR library—*bcprov-jdk15on-1.54.jar*—is not in the *lib* subdirectory where *sakagclient* exists.

Make sure there is a subdirectory called *lib* in the same folder where the *sakagclient.jar* file exists, and that the *BouncyCastle JCE Provider 1.54* exists in the *lib* subdirectory. Then, attempt the command again.

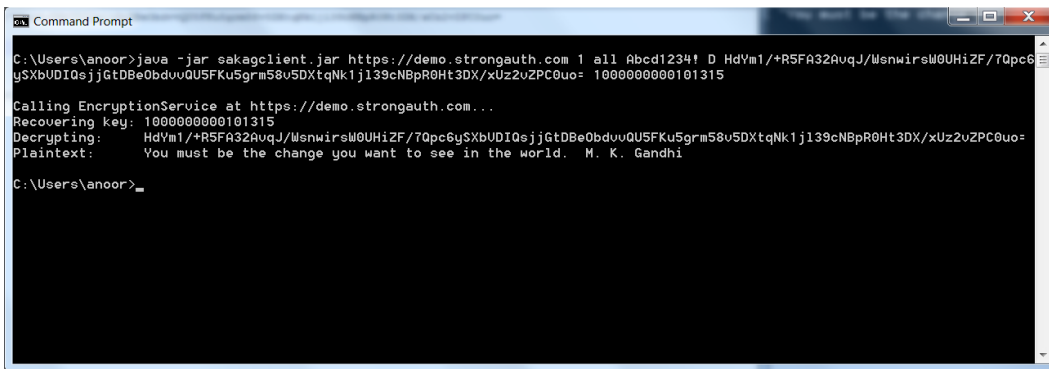
```
Exception in thread "main" java.lang.NoClassDefFoundError:
org/bouncycastle/util/encoders/Base64
  at GCMMain.decryptText(GCMMain.java:324)
  at GCMMain.main(GCMMain.java:237)
Caused by: java.lang.ClassNotFoundException:
org.bouncycastle.util.encoders.Base64
  at java.net.URLClassLoader.findClass(URLClassLoader.java:381)
  at java.lang.ClassLoader.loadClass(ClassLoader.java:424)
  at sun.misc.Launcher$AppClassLoader.loadClass(Launcher.java:331)
  at java.lang.ClassLoader.loadClass(ClassLoader.java:357)
  ... 2 more
```

## 4.4—Decryption (CBC Mode)

To test decryption, call *sakagclient* with the *D* option and use the ciphertext from your encryption transaction (from the transaction similar to the top half of the results in [3.4—Encryption \(CBC Mode\)](#) on your computer—it is sure to be different from the ciphertext shown in this document) as well as the token from your encryption test as parameters. Make sure you copy the ciphertext *exactly* as it shows in the output of the encryption command:

```
java -jar sakagclient.jar https://demo4.strongkey.com 1 all Abcd1234! D HdYm1/+R5FA32AvqJ/WsnwirsW0UHiZF/7Qpc6ySxbVDIQsjjGtDBe0bdvvQU5FKu5grm58v5DXtqNk1j139cNBpR0Ht3DX/xUz2vZPC0uo= 1000000000101315
```

[1.5—Encryption \(GCM Mode\)](#) shows the successful return of the original string (“You must be the change you want to see in the world. M. K. Gandhi”). The operation may take a little while because cryptographic keys may need to be loaded from the internal database and decrypted before the actual encrypted object can be decrypted. However, once the keys are decrypted, they are cached for a short duration (15 minutes); if you repeat the same decryption operation immediately, you'll notice an immediate response.



```
C:\Users\anoor>java -jar sakagclient.jar https://demo.strongauth.com 1 all Abcd1234! D HdYm1/+R5FA32AvqJ/WsnwirsW0UHiZF/7Qpc6ySxbVDIQsjjGtDBe0bdvvQU5FKu5grm58v5DXtqNk1j139cNBpR0Ht3DX/xUz2vZPC0uo= 1000000000101315

Calling EncryptionService at https://demo.strongauth.com...
Recovering key: 1000000000101315
Decrypting: HdYm1/+R5FA32AvqJ/WsnwirsW0UHiZF/7Qpc6ySxbVDIQsjjGtDBe0bdvvQU5FKu5grm58v5DXtqNk1j139cNBpR0Ht3DX/xUz2vZPC0uo=
Plaintext: You must be the change you want to see in the world. M. K. Gandhi

C:\Users\anoor>
```

**NOTE:** The first web service request for encryption or decryption within a user's session will see a long delay because of the following factors:

- The latency of the Internet
- The verification of the user's authorization (which causes the decryption of a hierarchy of keys in the hardware module to verify the user's password)
- The decryption of the symmetric key before it is available for use
- Finally, the actual encryption/decryption of the PAN data

A key-caching feature on KA (5 minutes by default) enables subsequent operations to see significant improvements in performance. However, at the end of the default caching period, the keys are flushed from memory and the first cryptographic operation after the flush will see a similar delay as keys are decrypted and loaded into cache memory. The caching window can be configured for your business needs.

From a performance point of view, the current generation of KA, using a TPM, delivers approximately 200–220 *web service operations per second (WSOPS)* when multi-threaded client applications make requests. In a clustered environment where all nodes in the cluster may service client applications, the overall WSOPS will be higher than for an individual KA, but will be difficult to predict in advance without understanding the nature of the network and applications in question.

## 4.5—Encryption (GCM Mode)

To test the GCM of authenticated encryption, type the following command:

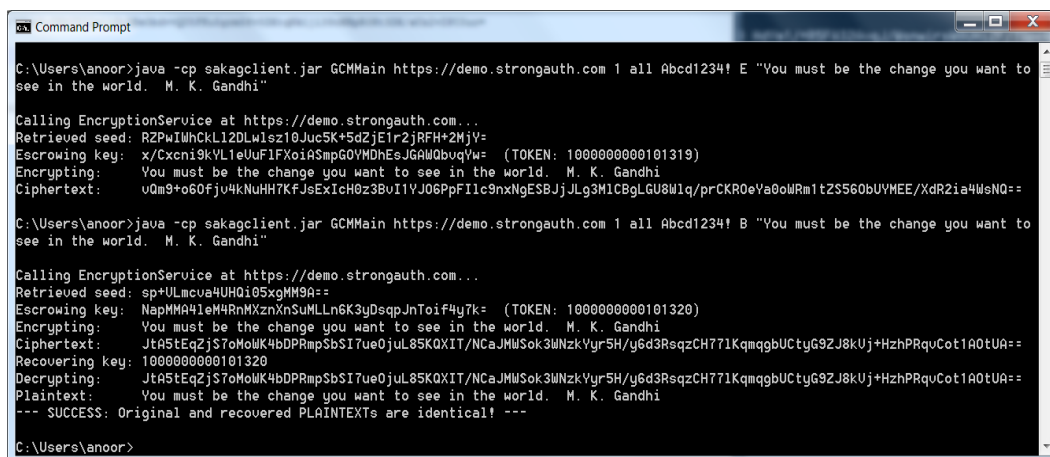
```
java -cp sakagclient.jar GCMMain https://demo4.strongkey.com 1 all
Abcd1234! E "You must be the change you want to see in the world. M. K.
Gandhi"

java -cp sakagclient.jar GCMMain https://demo4.strongkey.com 1 all
Abcd1234! E "You must be the change you want to see in the world. M. K.
Gandhi"
```

This calls *sakagclient* to encrypt the string specified on the command line using the GCM mode of operation. The result of this execution is shown in the top half of the image in [1.6—Decryption \(GCM Mode\)](#).

In this execution, *sakagclient* first calls the KAM to retrieve some entropy from the KA TRNG (shown in the Retrieved seed line). It then generates a new AES key, escrows it on the KAM, and gets a token (1000000000101319). Using the last 96 bits of the token for the *Additional Authenticated Data (AAD)*, *sakagclient* then encrypts the supplied string in the GCM mode of operation to display the resulting ciphertext.

In the bottom half of [1.6—Decryption \(GCM Mode\)](#), the same string is encrypted and decrypted by specifying the *B* (both) option. As one can see, it performs the same operations as for the encrypt operation, but goes on to recover the original cryptographic key by recovering it from the KAM, decrypting the ciphertext, and comparing the original plaintext with the decrypted value.



```
C:\Users\anoor>java -cp sakagclient.jar GCMMain https://demo.strongauth.com 1 all Abcd1234! E "You must be the change you want to
see in the world. M. K. Gandhi"

Calling EncryptionService at https://demo.strongauth.com...
Retrieved seed: RZPwIWhCKL12DLw1sz10Juc5K+5dZjE1r2jRFH+2HjY=
Escrowing key: x/Cxcni9kVL1eUuF1FXoiASmpGOYMDhEsJGAWQbuqVw= (TOKEN: 1000000000101319)
Encrypting: You must be the change you want to see in the world. M. K. Gandhi
Ciphertext: vQm9+o60fju4kNuHH7KfJsExIcH0z3BvI1VJ06PpFIlc9nxNgESBJJLg3M1CBgLGU8W1q/prCKR0eVa0oWRm1tZS560bUYMEE/XdR2ia4W6NQ=:

C:\Users\anoor>java -cp sakagclient.jar GCMMain https://demo.strongauth.com 1 all Abcd1234! B "You must be the change you want to
see in the world. M. K. Gandhi"

Calling EncryptionService at https://demo.strongauth.com...
Retrieved seed: sp+ULmcua4UH0i05xgMM9A=:
Escrowing key: NapMMA41eM4RnMXznXnSuMLLn6K3yDsqpJnToif4y7k= (TOKEN: 1000000000101320)
Encrypting: You must be the change you want to see in the world. M. K. Gandhi
Ciphertext: JEASTeqZjS7oMoMk4bDPRmpSbSI7ue0JuL85KQXIT/NCaJMWsok3WNzkVyr5H/y6d3RsqzCH71LKmqggbUCtyG9ZJ8kUj+HzhPRquCot1A0tUA=:
Recovering key: 1000000000101320
Decrypting: JEASTeqZjS7oMoMk4bDPRmpSbSI7ue0JuL85KQXIT/NCaJMWsok3WNzkVyr5H/y6d3RsqzCH71LKmqggbUCtyG9ZJ8kUj+HzhPRquCot1A0tUA=:
Plaintext: You must be the change you want to see in the world. M. K. Gandhi
--- SUCCESS: Original and recovered PLAINTEXTS are identical! ---

C:\Users\anoor>
```

## 4.6—Decryption (GCM Mode)

To test decryption, call *sakagclient* with the *D* option and use the ciphertext from your encryption transaction (from the transaction similar to the [Encryption \(CBC Mode\)](#) on your computer—it is sure to be different from the ciphertext shown in this document) as well as the token from your encryption test as parameters. Make sure you copy the ciphertext *exactly* at it shows in the output of the encryption command.

```
java -cp sakagclient.jar GCMMain https://demo4.strongkey.com 1 all  
Abcd1234! D  
vQm9+o60Fjv4kNuHH7KfJsExIcH0z3BvI1YJ06PpFIlc9nxNgESBJjJLg3M1CBgLGU8Wlq/pr  
CKR0eYa0oWRm1tZS560bUYMEE/XdR2ia4WsNQ== 1000000000101319
```

You should see a result similar to the following:



```
Command Prompt  
C:\Users\anoor>java -cp sakagclient.jar GCMMain https://demo.strongauth.com 1 all Abcd1234! D vQm9+o60Fjv4kNuHH7KfJsExIcH0z3BvI1YJ06PpFIlc9nxNgESBJjJLg3M1CBgLGU8Wlq/prCKR0eYa0oWRm1tZS560bUYMEE/XdR2ia4WsNQ== 1000000000101319  
Calling EncryptionService at https://demo.strongauth.com...  
Recovering key: 1000000000101319  
Decrypting: vQm9+o60Fjv4kNuHH7KfJsExIcH0z3BvI1YJ06PpFIlc9nxNgESBJjJLg3M1CBgLGU8Wlq/prCKR0eYa0oWRm1tZS560bUYMEE/XdR2ia4WsNQ==  
Plaintext: You must be the change you want to see in the world. M. K. Gandhi  
C:\Users\anoor>
```

In this manner, the *sakagclient* can be used to test various KAM web services.

# 5—KMS Client



To test KA web services, the appliance comes with a Java-based client application to test the web service operations of the *KeyManagementService (KMS)*—a module that provides the ability to store and manage ANSI X9.24-1:2009 Symmetric Keys. The Java client—`kmsclient.jar`—can be used to call web service operations on StrongKey's Demo Client on the internet—[demo4.strongkey.com](https://demo4.strongkey.com)—or on a private Tellaro cluster in your network.

The KA Demo Client uses software identical to that of appliances delivered to customers. This allows your application development teams to verify code without having to wait for a KA implementation internally. Once verified against the KA Demo Client, the code will work against an internal KA implementation without changing a single line of code. The only change would be the URL of the host called and the parameters passed to the application at runtime. We recommend parameterizing the URL of the appliances to be called; this will make it easier to point applications to KA in your production environment.

The `kmsclient.jar` application is a Java application that calls KA web services; it simulates what applications must do to call the KMS on the Tellaro. While your client application will certainly be different based on the business functions it implements, the programming language it uses, and the application model it uses, ultimately it *must* perform the same web service functions as the test client application bundled with KA. Because the client application focuses only on testing the implementation of the KA web service, it is ideal for ensuring the correctness of your implementation.

The currently supported operations of the `kmsclient` are:

1. Generate a *Base Derivation Key (BDK)* and produce its key components.
2. Generate an RSA asymmetric key pair, encrypt and token the private key, and return the public key and token.
3. Generate an Initial Key (sometimes also referred to as *Initial PIN Entry Key* or *IPEK*) from a BDK for a card reader device.
4. Load a BDK's key components into the *Card Cryptographic Service (CCS)* module.
5. Load a BDK into the CCS Module from its previously submitted key components.
6. Store an ANSI Key in the appliance's secure storage.
7. Replace a currently stored ANSI Key in the appliance with new ANSI Key.
8. Delete a currently stored ANSI Key from the appliance.
9. Update the status of a currently stored ANSI Key in the appliance.
10. List supported card reader device manufacturers by ID.

The client application distinguishes between these operations to allow sites to test different levels of authorization for the web services:

1. *Key Management Operator (KMO)* users who are authorized to load key components.
2. *Key Management Custodian (KMC)* users who are authorized to generate and load BDKs and IPEKs.
3. *Key Management Administrators (KMA)* who are authorized to store, replace, delete, and update ANSI Keys.

## 5.1—Installing the Demo Client Application

StrongKey distributes the Demo Client as a .ZIP file. After unzipping the *democlients* distribution, the *kmsclient* executable and source will be in a folder called *kmsclient*.

The *kmsclient.jar* file contains executable code. The *kmsclient-src.zip* file contains the source for this executable as a project folder. You can build the executable .JAR file from scratch if you wish, or you can modify the client to integrate into your own application.

To use the *kmsclient.jar* program, open a shell or command prompt window and navigate to the directory location where *kmsclient.jar* is installed. Once you've changed to that directory, you may execute the .JAR file without specifying its location. For the rest of this document, we will assume that *kmsclient.jar* is in your current directory.

## 5.2—Displaying Help Options

Type the following command to verify that the client application is executable. It also shows a helpful prompt of the operations it can perform and the parameters it expects for each operation.

```
java -jar kmsclient.jar
```

You will see the following output.

```
Usage: java -jar kmsclient.jar https://<host:encport> <did> <username>
<password> DLK [bankid] <keytoken>
  java -jar kmsclient.jar https://<host:encport> <did> <username> <password>
GAK <key-name> <BDK|LTMK|MAC|TMK|TPK|GPK|Other> <RSA> <1024|2048|4096|8192>
<bankid> [terminalid] [terminaltype] [notes] <Hex|Base64>
  java -jar kmsclient.jar https://<host:encport> <did> <username> <password>
GBK <manufacturer>
  java -jar kmsclient.jar https://<host:encport> <did> <username> <password>
GIK <manufacturer> <device-serial-number> <return-type> [public-key-token]
  java -jar kmsclient.jar https://<host:encport> <did> <username> <password>
LBK <key-name> <hex-key-check-value> <manufacturer>
  java -jar kmsclient.jar https://<host:encport> <did> <username> <password>
LKC <AES|TDES> <128|192|256> <hex-key component> [kcv] <K-Value> <N-Value>
<key-name>
  java -jar kmsclient.jar https://<host:encport> <did> <username> <password>
RDK <retiringtoken> [parent-token] <key-name> <BDK|LTMK|MAC|TMK|TPK|Other>
<AES|TDES> <128|192|256> <bankid> [terminalid] [terminaltype] [hex-symkey]
<kcv> [notes]
  java -jar kmsclient.jar https://<host:encport> <did> <username> <password>
SDK [parent-token] <key-name> <BDK|LTMK|MAC|TMK|TPK|Other> <AES|TDES> <128|
192|256> <bankid> [terminalid] [terminaltype] [hex-symkey] <kcv> [notes]
  java -jar kmsclient.jar https://<host:encport> <did> <username> <password>
UPK [bankid] <keytoken> [newstatus] [newnotes]
  java -jar kmsclient.jar MFR
```



An explanation of the parameters to the Demo Client application is as follows:

Parameter	Explanation
<code>https://&lt;host:port&gt;</code>	The URL of the KA where the CCS web service operations will be submitted. For testing this client program on StrongAuth's DEMO machine, the URL is <a href="https://demo4.strongkey.com">https://demo4.strongkey.com</a> .
<code>&lt;did&gt;</code>	The unique Encryption Domain identifier. This was provided separately by Strongkey as a unique domain for testing.
<code>&lt;username&gt;</code>	The username within the encryption domain with authorization to call the web services. These usernames were provided separately by StrongKey for testing.
<code>&lt;password&gt;</code>	The password of the username within the encryption domain authorized to call the web service. These passwords were provided separately by StrongKey for testing.

The client supports the following operations

DLK	Delete ANSI Key	Delete an ANSI Key from KA secure storage.
GAK	Generate Asymmetric Key	Generate an RSA asymmetric key pair, encrypt and token the private key, and return the public key and token.
GBK	Generate BDK	Generate a BDK in KA and export it as three key components with their corresponding <i>Key Check Values (KCV)</i> .
GIK	Generate Initial Key	Generate an Initial Key on the KA derived from a BDK and a <i>Device Serial Number (DSN)</i> of the card reader device and export it as three key components with their corresponding KCVs.
LKC	Load Key Component	Load a BDK's key component with its KCV into the CCS Module (to be later assembled into the BDK by the LBK option).
LBK	Load BDK	Assemble a BDK from previously loaded key components, verify its KCV with the supplied KCV from the command line, and load the verified BDK into the CCS Module.
RDK	Replace ANSI Key	Replace a currently stored ANSI Key in the KA with a new ANSI Key.
SDK	Store ANSI Key	Store a BDK or LTMK in the KA from Key Components, or store a TMK or TPK encrypted by a key stored in KA.
UDK	Update ANSI Key	Update the status of an ANSI Key currently stored in the KA.
MFR	List Manufacturer	List supported card reader manufacturers by ID.
	<code>&lt;128 192 256&gt;</code>	Available key sizes accepted by the appliance.
	<code>&lt;AES TDES&gt;</code>	Either the AES or Triple DES algorithm.
	<code>&lt;BDK LTMK MAC TMK TPK GPK Other&gt;</code>	The type of key to be stored in the KA.
	<code>&lt;bankid&gt;</code>	A numeric identifier with which bank this key is associated.
	<code>&lt;ciphertext&gt;</code>	The ciphertext value to be decrypted through DUKPT.
	<code>&lt;device-serial-number&gt;</code>	The DSN for the device. This must be either an 11- or 15-byte string of hex-encoded data.



Parameter	Explanation
<hex-symkey>	The encrypted key which is to be loaded in KA. Can be null if the word "null" is specified.
<hex-key component>	A key component value submitted as a hex-encoded string.
<hex-key-check-value>	A KCV as a hex-encoded string.
<K-Value>	The numerical identifier for a key component in a set of key components.
<key-name>	A string identifying multiple key components as belonging to the same set. The value can be any 64-character long string (without spaces and special characters). It is merely used to identify the key components during a "Load BDK" operation.
<keytoken>	A token that references an ANSI key to operate on.
<ksn>	The Key Serial Number for this DUKPT transaction. Must be supplied as 20 hex-encoded bytes (10 bytes when decoded from Hex).
<manufacturer>	The Manufacturer ID for the product that will be creating encrypted swipes.
<N-Value>	The total number of key components in a set of key components.
<newnotes>	The new notes associated with an ANSI key.
<newstatus>	The new status for an ANSI key. Valid values for <i>newstatus</i> are <i>Active</i> , <i>Inactive</i> , <i>Suspended</i> , and <i>Retired</i>
<notes>	Any extra information to be sent when storing a key. Can be <i>NULL</i> if the word "null" is specified.
<parent-token>	The token in which the key encrypting key is stored. Can be <i>NULL</i> if the word "null" is specified
<plaintext>	The plaintext value to be encrypted through DUKPT.
<public-key-token>	Specifies a token in KA that holds a previously escrowed public key which can be used to encrypt keys for export from the appliance. This parameter is optional.
<return-type>	Determines the manner in which an Initial Key is returned to the calling application. The only currently supported value is to return the Initial Key as <i>KeyComponents</i> . Future implementations will also support returning the Initial Key, encrypted under a previously escrowed public key on KA/
<terminalid>	An optional numerical identifier for which terminal this ANSI key is associated with.
<terminaltype>	An optional string identifier for the type of terminal this ANSI key belongs to.

## 5.3—List Manufacturer

Some web service operations in the KMS servlet utilize a *ManufacturerID* to associate with an ANSI Key. If you plan on storing a BDK using the load BDK web service, you must identify the *ManufacturerID* to be used while making web service calls to that web service. This can be done with the *List Manufacturer* function. The IDs displayed are internal values used by the CCS to uniquely distinguish different card readers supported by KA:

```
java -jar kmsclient.jar MFR

Manufacturer ManufacturerID:
-----
IDTech 0
UIC 1
MagTek 2
Infinite 3
Dejavoo 4
PAX 5
```

To generate a new BDK, type in the following command. In this example the URL parameter is <https://demo4.strongkey.com>, the encryption domain ID is the numeral **1**, the *username* is **kmcuser** and the *password* is **Abcd1234!**. The last parameter indicates the card reader manufacturer this BDK will support (in this case, **Dejavoo**):

```
java -jar kmsclient.jar https://demo4.strongkey.com 1 kmcuser Abcd1234! GBK 4

Calling KeyManagementService at https://demo4.strongkey.com...

KeyComponent 1 of 3 [KCV1]
EE0F44EEE62E95DB4EE5A7D4A588F7ED [870B01]

KeyComponent 2 of 3 [KCV2]
FA886DDB069533F0BBF3D738EA23A25E [9F318A]

KeyComponent 3 of 3 [KCV3]
C9B8AC6A205A920D6829A3EC3A57CD3E [5C6A48]

BDK Key Check Value [0DD5F2]
```

The return value is an array of key components with their corresponding *Key Check Values* (KCVs). The generated BDK is split into three parts—KeyComponent 1 of 3, KeyComponent 2 of 3, and KeyComponent 3 of 3. Each component has a corresponding KCV; the BDK itself has its own KCV.

If you see an error message that resembles the following, it implies an outdated version of the JDK. *JDK 8 Update 112 or later* is required to make a web service connection to StrongKey's Demo Client when using Java.

```
Exception in thread "main" javax.xml.ws.WebServiceException: Failed to
access the WSDL at:
https://demo4.strongkey.com/kmsADM/KeyManagementService?wsdl. It failed
with:
    sun.security.validator.ValidatorException: PKIX path building failed:
    sun.security.provider.certpath.SunCertPathBuilderException: unable to
    find valid certification path to requested target.
    at
    com.sun.xml.internal.ws.wsdl.parser.RuntimeWSDLParser.tryWithMex(RuntimeW
    SDLParser.java:151)
    at
    com.sun.xml.internal.ws.wsdl.parser.RuntimeWSDLParser.parse(RuntimeWSDLPa
    rser.java:133)
    at
    com.sun.xml.internal.ws.client.WSServiceDelegate.parseWSDL(WSServiceDeleg
    ate.java:254)
    at
    com.sun.xml.internal.ws.client.WSServiceDelegate.<init>(WSServiceDelegat
    e.java:217)
    at
    com.sun.xml.internal.ws.client.WSServiceDelegate.<init>(WSServiceDelegat
    e.java:165)
    at
    com.sun.xml.internal.ws.spi.ProviderImpl.createServiceDelegate(ProviderIm
    pl.java:93)
    at javax.xml.ws.Service.<init>(Service.java:56)
    at
    com.strongauth.strongkeylite.web.EncryptionService.<init>(EncryptionServi
    ce.java:79)
    at strongkeyliteClient.Main.main(Main.java:109)
```

## 5.4—Load Key Component

Before any Initial Keys are derived, a BDK's three key components must be loaded into the KMS Module so the BDK may be assembled internally from the components. Using the values generated by the GBK option in the previous step, the first key component will be EEOF44EEE62E95DB4EE5A7D4A588F7ED with a Key Check Value of 870B01. This is Key Component 1 of 3. To identify this set of key components as belonging to the same BDK, we assign the value of *dejavooBDK* to identify this set. It could have been any string such as "MySecureBDK," or "DarkSideOfTheMoonBDK," etc. This time we use the *kmouser* to make the call.

```
java -jar kmsclient.jar https://demo4.strongkey.com 1 kmouser
Abcd1234! LKC AES 128 EE0F44EEE62E95DB4EE5A7D4A588F7ED 870B01 1 3
dejavooBDK
Calling KeyManagementService at https://demo4.strongkey.com...
{"did":"1","K":1,"N":3,"KCV":"870B01"}
```

The return value is a JSON object affirming the successfully loaded key component. This process is repeated for the remaining key components of the BDK, as shown below. Please note that the identification string (*dejavooBDK*) for the key components must be identical for all key components.

```
java -jar kmsclient.jar https://demo4.strongkey.com 1 kmouser Abcd1234!
LKC AES 128 FA886DDB069533F0BBF3D738EA23A25E 9F318A 2 3 dejavooBDK
Calling KeyManagementService at https://demo4.strongkey.com...
{"did":"1","K":2,"N":3,"KCV":"9F318A"}
java -jar kmsclient.jar https://demo4.strongkey.com 1 kmouser Abcd1234!
LKC AES 128 C9B8AC6A205A920D6829A3EC3A57CD3E 5C6A48 3 3 dejavooBDK
Calling KeyManagementService at https://demo4.strongkey.com...
{"did":"1","K":3,"N":3,"KCV":"5C6A48"}
```

## 5.5—Load BDK

To finish loading the BDK into KMS, we have to call the *Load BDK* web service. We need to specify the key component Identifier used when loading the key components—in this example's case it is *dejavooBDK*. Additionally, the KCV for the BDK itself must be provided to verify it is imported correctly from its components:

```
java -jar kmsclient.jar https://demo4.strongkey.com 1 kmcuser Abcd1234!
LBK dejavooBDK 0DD5F2 4
Calling KeyManagementService at https://demo4.strongkey.com...
{"did":"1","KCV":"0DD5F2"}
```

The return value is a JSON object affirming that BDK was successfully loaded.

## 5.6—Generate Initial Key

Now that the BDK is loaded into the KMS Module, an Initial Key can be derived from the BDK for injecting into a card reader device. You must specify the same *ManufacturerID* used when loading the BDK, and the DSN for the card reader device. The DSN must be either an 11- or 15-byte hex-encoded string. **The 21 bits for the device counter must not be included in the DSN.** If 11 bytes are sent, the DSN will be left padded with 1s (*FFFF*) internally by the CCS. A return type of *KeyComponents* is specified to indicate that the Initial Key should be returned as key components:

```
java -jar kmsclient.jar https://demo4.strongkey.com 1 kmcuser Abcd1234!
GIK 4 987654321FE KeyComponents
Calling KeyManagementService at https://demo4.strongkey.com...
KeyComponent 1 of 3 [KCV1]
3FA355C21252A639D4840CA814D19E48 [F5F3D9]
KeyComponent 2 of 3 [KCV2]
FCC12D7D7C0472708378F3AEFAA87E94 [A32F1B]
KeyComponent 2 of 3 [KCV2]
D6C6E7AE0EF00283E9D3E1BEB94D39D8 [A47A5D]
InitialKey Key Check Value [A9A783]
```

The return value is an array of key components with their corresponding KCVs. These components can now be used to inject into the card reader with the specified serial number. It is assumed that the card reader manufacturer has firmware built in to assemble the three key components into the device's Initial Key and verify the KCVs of each component and the assembled Initial Key using the standard algorithms specified in the ANSI specification.

## 5.7—Store ANSI Key (BDK or LTMK)

The *storeAnsiX9241Key* (SDK) operation is used for persistent, secure storage of ANSI X-924.1 keys in the KA. Keys must be stored in this manner before many of the CardCryptoService web services can be called. Before we can load a LTMK or a BDK, we must load all the Key Components of the key into the KA:

```
java -jar kmsclient.jar https://demo4.strongkey.com:8181 1 kmouser
Abcd1234! LKC TDES 128 E08B7275A339EA1A73B2B79155296767 658485 1 3 MyLTMK
{"did":"64","K":1,"N":3,"KCV":"658485"}
java -jar kmsclient.jar https://demo4.strongkey.com:8181 1 kmouser
Abcd1234! LKC TDES 128 42DC94AD5298B8F3AFAED04655F3B8E0 43FA8F 2 3 MyLTMK
{"did":"64","K":2,"N":3,"KCV":"43FA8F"}
java -jar kmsclient.jar https://demo4.strongkey.com:8181 1 kmouser
Abcd1234! LKC TDES 128 A1F0F8FBF75F588C466E7462E28537BD 6767FD 3 3 MyLTMK
{"did":"64","K":3,"N":3,"KCV":"6767FD"}
```

Now that the Key Components are loaded, call the *Store ANSI Key* web service. For this call, use *kmauser*:

```
java -jar kmsclient.jar https://demo4.strongkey.com:8181 1 kmauser
Abcd1234! SDK null MyLTMK LTMK TDES 128 1 123 null DC9119 "Loading LTMK"
Token received: {"did":"1","token":"100000000000000002"}
```

The web service returns a token for the stored key. This token must be recorded so it can be provided to future *Store ANSI Key* web service calls as a parent token.

## 5.8—Store ANSI Key (TMK or TPK)

Using the same *Store ANSI Key* web service as before, store keys that have been encrypted with keys already stored on the KA. This time, specify a <parent - token> that identifies the key-encrypting key and the <hex-symkey> which is the encrypted hex bytes of the key:

```
java -jar kmsclient.jar https://demo4.strongkey.com:8181 1 kmauser
Abcd1234! SDK 100000000000000002 MyTMK TMK TDES 128 1 123
9AE3E3407AE0C3DFA72D07CC71B8BC57 DC9119 "Loading TMK"
Token received: {"did":"1","token":"100000000000000003"}
```

As before, the web service returns the token which identifies this key in KA. Using this token, store a TPK under the TMK:

```
java -jar kmsclient.jar https://demo4.strongkey.com:8181 1 kmauser
Abcd1234! SDK 100000000000000003 MyTPK TPK TDES 128 1 123
982D868704702DF9070922F1DFB260A8 6776FF "Loading TPK"
Token received: {"did":"1","token":"100000000000000004"}
```

## 5.9—Replace ANSI Key (TMK or TPK)

The *Replace ANSI Key* web service looks very similar to the *Store ANSI Key* web service. Use it to store keys encrypted with keys that are stored on KA, while additionally retiring a key that already on the appliance. In this fashion, KA replaces one TPK for another TPK:

```
java -jar kmsclient.jar https://demo4.strongkey.com:8181 1 kmauser
Abcd1234! RDK 100000000000000004 100000000000000003 MyTMK TMK TDES 128 1 123
C65D9B58A575245A4B3EF06C4410BBCA2E24EEB85AEF49AE 079FC5 "Loading TMK"
Token received: {"did":"1","token":"100000000000000005"}
```

As before, the web service returns the token which identifies this key in KA.

## 5.10—Update ANSI Key

The *Update ANSI Key* web service updates the status or notes of a key stored on the appliance.

```
java -jar kmsclient.jar https://demo4.strongkey.com:8181 1 kmauser
Abcd1234! UPK 1 100000000000000005 Retired "Key Retired on May 25th 2017"
Success: {"DID":"1", "BankID":"1", "KeyToken":"100000000000000005",
"NewStatus":"Retired", "NewNotes":"Key Retired on May 25th 2017"}
```

## 5.11—Delete ANSI Key

The *Delete ANSI Key* web service can be used to delete a key stored on the appliance.

```
java -jar kmsclient.jar https://demo4.strongkey.com:8181 1 kmauser
Abcd1234! DLK 1 100000000000000005
Success: {"DID":"1", "BankID":"1", "KeyToken":"100000000000000005"}
```

## 5.12—Generate Asymmetric Key

To generate a new RSA asymmetric key, type the following command:

```
java -jar kmsclient.jar https://demo4.strongkey.com 1 kmcuser Abcd1234!
GAK myRSAKey LTMK RSA 4096 1 123 null myNotes Hex
Calling KeyManagementService at https://demo4.strongkey.com...
{"DID":1,"SRID":1528827528553,"KeyType":"RSA","KeyFormat":"RSAPublicKey",
"KeySize":"2048","PublicKey":"30820122300d06092a864886f70d010101050003820
10f003082010a0282010100a62ad0e81f57ca1c32390f442b3505b351f1a1a76346159a84
17846ce76e426573aad4b27094843898987b19241b29eba348b4db5c843a81e4d6ef72240
d815ca9a907f81d0211c66dd307bad804c556c15be86804c7c084f760ae132691158527d2
671769f1b0fc8ef76d0d0fef0becd98e1a63751e96b694b044fd89907f6c7007d56edd7d4
9bca67cf44540a919ebb9de7119515ec70eb12b688adb1fee8e216860c7bd7d99cd7a3d
b6edae21e7ffa3d77bb4c10726bc7c851fce12b864b4ab7b10e75d225b8de3cba44aaf262
97392f56f96c43ade09e4a3f578b57947c9d6ef56fe5df244383e112d98a35001f91e9bd1
6094f6e8e9a942c4a073cd110203010001","PublicKeyEncoding":"Hex","Nonce":"Vz
7GMoNPNTtpZ3pK0fZ/j+7P4U46WAtkqowwLHjx1Q=", "SignedNonce":"212f9db4322ee1
e90d0b0d311ccf080cc4d69a3861a2d13255ebe64af2309ac7b6fe9f3263956ec90c9333d
8288a3a92ce90fee9bb886b1c6de89e29870794c98ccdde07ec94715e99b9340c828a9fc0
f2e748b3504c6aa5f6670c148c2b81e20a45f2cc1439c821d328645c562a43370e17c0e35
895c4886599bcb1cef2a97f8d87a29beaec2e4048177eb49e369d841297a8bab4ee9cc9ff
1c91a2bec197672d68d027fb51e4c546cce06dbb59a2ca869f765f161ca50fa53e0744121
486b3463481f4099e5cf3154300e2947873c3c34250e0a3d5e0d7a8a62a1ec2c7d9bba15a
8ea740fba753f90c7782cc98d8b86932c5b6f6aed1314e9e7743f35b4dea", "SignedNonc
eEncoding":"Hex", "SignatureDigest":"SHA256", "Token":"10000000000002557"}
```

By storing the private key on the appliance and returning the public key, this provides another method to store *American National Standards Institute (ANSI)* keys by encrypting the ANSI key with the public key and supplying the corresponding private key token as the parent token.



# 6—CCS Client



To test KA web services, the appliance comes with a Java-based client application that can be used to test the web service operations of the *Card Crypto Service* (CCS)—a module that processes encrypted credit card transactions using the *Derived Unique Key Per Transaction (DUKPT)* algorithm specified in the *ANSI X9.24-1:2009 Symmetric Key Management* standard. The Java client—`ccsclient.jar`—can be used to call web service operations on StrongKey's Demo Client on the internet—[demo4.strongkey.com](https://demo4.strongkey.com)—or on your own Tellaro cluster within your network.

The KA Demo Client uses software identical to that of the appliances delivered to customers. This allows application development teams to verify code without having to wait for a KA implementation internally. Once verified against the KA Demo Client, the code will work against an internal KA implementation without changing a single line of code. The only change would be the host URL to be called and the parameters to be passed to the application at runtime. We recommend parameterizing the URL of the appliances to be called; this will make it easier to point applications to KA in a production environment.

The `ccsclient.jar` application is a simple Java application that calls web services on KA; it simulates what applications must do to call the CCS on the Tellaro. While your client application will certainly be different based on the business functions it implements, the programming language it uses, and the application model it uses, ultimately it *must* perform the same web service functions as the test client application bundled in KA. Because the client application focuses only on testing the implementation of the KA web service, it is ideal for ensuring the correctness of your implementation.

The currently supported operations of the Demo Client are as follows:

1. Generate a *Base Derivation Key (BDK)* and produce its key components.
2. Generate an Initial Key (sometimes also referred to as *Initial PIN Entry Key* or *IPEK*) from a BDK for a card reader device.
3. Load a BDK's keycomponents into the CCS Module.
4. Load a BDK into the CCS Module from its previously submitted key components.
5. Process an encrypted card swipe by decrypting it using the DUKPT algorithm, re-encrypting the plaintext cardholder data using a data encryption key in KA and tokenizing it.
6. Generate a test encrypted card swipe using a specific Initial Key so the encrypted card swipe can be submitted to KA for processing.
7. List supported card reader device manufacturers by ID.

The client application distinguishes between these operations to allow sites to test different levels of authorization for the web services:

1. Users who are authorized to only encrypt.
2. Users who are authorized to only decrypt.
3. Users who are authorized to encrypt *and* decrypt.
4. Users who are authorized to perform all operations.
5. Users who are *NOT* authorized to perform any operation.



## 6.1—Installing the Demo Client Application

StrongKey distributes the demo application as a .ZIP file. After unzipping the *democlients* distribution, the *ccsclient* executable and source will be contained within a folder called *ccsclient*.

The `ccsclient.jar` file contains executable code. The `ccsclient-src.zip` file contains the source for this executable as a project folder. You can build the executable .JAR file from scratch if you wish, or you can modify the client to integrate into your own application.

To use the `ccsclient.jar` program, open a shell or command prompt window and navigate to the directory location where `ccsclient.jar` is installed. Once you've changed to that directory, you may execute the .JAR file without specifying its location. For the rest of this document, we will assume that `ccsclient.jar` is in your current directory.

## 6.2—Prerequisites

The steps in this chapter rely on ANSI keys being loaded on KA prior to executing *ccsclient* commands. See *Chapter 4—KMS Client* if you do not have ANSI keys loaded yet.

## 6.3—Displaying Help Options

Type the following command to verify the client application is executable. It also shows a helpful prompt of the operations it can perform and the parameters for each.

The explanation of the parameters to the Demo Client application is as follows:

Parameter		Explanation
<code>https://&lt;host:port&gt;</code>		The URL of the Tellaro where the CCS web service operations will be submitted. For testing this client program on StrongKey's Demo Client, the URL is: <a href="https://demo4.strongkey.com/">https://demo4.strongkey.com/</a> .
<code>&lt;did&gt;</code>		The unique Encryption Domain identifier. This was provided separately by StrongKey as a unique domain for testing.
<code>&lt;username&gt;</code>		The username in the encryption domain with authorization to call the web services. These usernames were provided separately by StrongKey for testing.
<code>&lt;password&gt;</code>		The password of the username within the encryption domain with authorization to call the web service. These passwords were provided separately by StrongKey for testing.
The client supports the following operations:		
GCD	Get Card Capture Data	Process an encrypted card swipe by decrypting it using the DUKPT algorithm, re-encrypting the plaintext cardholder data, tokenizing it, and returning plaintext card swipe data with the token in place of the encrypted cardholder data.
SWP	Generate Swipe Data	Simulate an encrypted card swipe by creating it with supplied key components of an Initial Key and test cardholder data.

Parameter		Explanation
RPB	Re-encrypt Pin Block	Decrypt a PIN Block sent to the appliance using a BDK stored on the appliance, and re-encrypt that PIN Block using a TPK stored on the appliance.
DENC	DUKPT Encrypt	Encrypt a plaintext with a key stored in KA using the DUKPT algorithm.
DDEC	DUKPT Decrypt	Decrypt a ciphertext with a key stored in KA using the DUKPT algorithm.
DMAC	DUKPT MAC	Create a MAC of a plaintext value using a key derived from a BDK stored on the appliance.
MFR	List Manufacturer	List supported card reader manufacturers by ID.
	<mfr>	The <i>ManufacturerID</i> for the product that will be creating encrypted swipes. Can be <i>NULL</i> if the word “null” is specified.
	<AES TDES>	Either the AES or Triple DES algorithm.
	<bdktoken>	The token in which the BDK is stored. Can be <i>NULL</i> if the word “null” is specified.
	<ccdata>	A “raw” card swipe as it would come out of a DUKPT-enabled card reader. The format printed on the screen is the format expected by the CCS for the “Get Card Capture Data” web service option.
	<ciphertext>	The ciphertext value to be decrypted through DUKPT.
	<DEK PIN MAC> or <MAC_REQUEST  MAC_RESPONSE>	The type of key to use in this DUKPT operation.
	<device-serial-number> or <dsn>	The <i>Device Serial Number (DSN)</i> for the device. This must be either an 11- or 15-byte string of hex-encoded data.
	<hex-encrypted-PIN- block>	The encrypted key which is to be loaded in KA. Can be <i>NULL</i> if the word “null” is specified.
	<hex-KSN>	The <i>Key Serial Number (KSN)</i> for this DUKPT transaction. Must be supplied as 20 hex-encoded bytes (10 bytes when decoded from hex).
	<plaintext>	The plaintext value to be encrypted or MAC'd through DUKPT.
	<test-card-number>	A 16-digit number to simulate a credit card; it can be any value and does not need to be real cardholder data.
		The token in which the ANSI Key is stored. Can be <i>NULL</i> if the word “null” is specified.
	<tpktoken>	The token in which the TPK is stored.

You will see the following output.

```
Usage: java -jar ccscclient.jar https://<host:encport> <did> <username>
<password> DENC [token] [mfr] <ksn> <plaintext> <AES|TDES> <DEK|PIN|MAC>
java -jar ccscclient.jar https://<host:encport> <did> <username>
<password> DDEC [token] [mfr] <ksn> <ciphertext> <AES|TDES> <DEK|PIN|MAC>
java -jar ccscclient.jar https://<host:encport> <did> <username>
<password> DMAC [token] [mfr] <ksn> <ciphertext> <MAC_REQUEST|
MAC_RESPONSE>
```

```

java -jar ccscclient.jar https://<host:encport> <did> <username>
<password> GCD <ccdata> <dsn> [manufacturer]
java -jar ccscclient.jar https://<host:encport> <did> <username>
<password> RPB <bdktoken> <tpktoken> <hex-KSN> <hex-encrypted-PIN-block>
<AES|TDES>
java -jar ccscclient.jar MFR
java -jar ccscclient.jar SWP <manufacturer> <device-serial-number> <key-
component1> <key-component2> <key-component3> <test-card-number>

```

## 6.4—List Manufacturer

Before the `ccscclient.jar` can be used to make web service calls, you must identify the *ManufacturerID* to be used while making web service calls to KA. This can be done with the *List Manufacturer* function. The IDs displayed are internal values used by the CCS to uniquely distinguish different card readers supported by KA:

```

java -jar ccscclient.jar MFR
Manufacturer ManufacturerID:
-----
IDTech 0
UIC 1
MagTek 2
Infinite 3
Dejavoo 4
PAX 5

```

## 6.5—Generate Swipe Data

While the Initial Key's components can be injected into a device reader from the previous steps, `ccscclient.jar` also has the ability to quickly test if the BDK and Initial Keys generated are working correctly. To enable this, the tool offers an option to generate a test card swipe. This command does not make a web service call to KA, and therefore does not need the web service URL or credentials. It does, however, need the *ManufacturerID* of the card reader device for which the BDK and Initial Key were generated, the DSN of the card reader device for which the Initial Key was generated, the three key components of the Initial Key, and a test 16-digit number to simulate a credit card number (we use `1111222233334444` in this example):

```

java -jar ccscclient.jar SWP 4 987654321FE
3FA355C21252A639D4840CA814D19E48 FCC12D7D7C0472708378F3AEFAA87E94
D6C6E7AE0EF00283E9D3E1BEB94D39D8 1111222233334444
Generated Swipe Data:
%B1111000000004444^TESTSWIPE/STRONGAUTH
^08043210000000725000000?|;1111000000004444=080432100000007250?|
987654321FE|
66ACA81502DCF1978DBA4A824C0350D4AA3C1DC428B5107C3179505A460A13225576DF0EE
452C67E2A83647D1173ED16AF1BF1099EF0BE018F139BA0317AA22E7E5CDFFD27ECFEC705

```

```
0A294C7B4CBE59 |
99EDF0FB7249400803F3CC79B28ACB5DD63C25985777315F1EEC44D08ACD4163A321B24D9
390F7C1A45A27CF8E55FC9C || FFFF987654321FE00001 ||
```

The client generates the transaction's derived key from the Initial Key, creates simulated Track1 and Track2 data, encrypts it, and formats the output as a real card reader device from the manufacturer might. The output presented on the screen is the precise output required for the last step of this test.

`ccsclient.jar` currently supports only the *Dejavoo* and *PAX* card reader swipe formats.

## 6.6—Get Card Capture Data

With the simulated card swipe data, the Get Card Capture Data web service can be called to process the DUKPT transaction. The web service requires the entire swipe blob as input. It is critical to wrap the swipe data in single quotes (') to prevent any special characters from being interpreted by the Linux or Windows shell. The DSN and the Manufacturer ID of the card reader are also necessary parameters for the web service.

```
java -jar ccsclient.jar https://demo4.strongkey.com 1 all Abcd1234! GCD
'%B1111000000004444^TESTSWIPE/STRONGAUTH
^08043210000000725000000?|;1111000000004444=080432100000007250?|
987654321FE|
66ACA81502DCF1978DBA4A824C0350D4AA3C1DC428B5107C3179505A460A13225576DF0EE
452C67E2A83647D1173ED16AF1BF1099EF0BE018F139BA0317AA22E7E5CDFFD27ECFEC705
0A294C7B4CBE59|
99EDF0FB7249400803F3CC79B28ACB5DD63C25985777315F1EEC44D08ACD4163A321B24D9
390F7C1A45A27CF8E55FC9C||FFFF987654321FE00001||' 987654321FE 4
Calling CardCryptoService at https://demo4.strongkey.com...
{"DID":"1","SRID":"1481245973096","Token":"2000000000000336","ExpiryDate"
:"0804","ExpiryMonth":"04","ExpiryYear":"08","MaskedPAN":"111100000000444
4","Digest":null,"Valid":true,"Exists":true,"AssociationID":"1","IssuerID
":"111122","CardholderName":"STRONGAUTH
TESTSWIPE","Firstname":"STRONGAUTH","Lastname":"TESTSWIPE","Notes":null}
```

Upon successfully processing the transaction, the KA returns a JSON with parsed values from the card swipe. By default, the appliance does not return the decrypted card number with the JSON. Instead, the KA re-encrypts and tokenizes the credit card number and returns a Token—2000000000000336, in this example. To view the actual credit card number, `sakaclient.jar` is necessary to decrypt this token (see the chapter on *sakaclient* on how to use it for testing).

## 6.7—DUKPT Encrypt

KA can be used to encrypt text using the DUKPT algorithm to derive a key. This web service requires that the encrypting key must have already been stored in KA and can be referenced using either a token or a *manufacturerID*. Additionally, a KSN of 10 bytes (20 while hex-encoded), encryption algorithm, and derived key type must be provided for this transaction. The plaintext to be encrypted must be passed to the web service in a hex-encoded format:

```
java -jar ccsclient.jar https://demo4.strongkey.com:8181 1 all Abcd1234!
DENC null 4 62994900410005c00014
3B353432343138303432363330373235393D303231323130313331373031303635383F33
TDES DEK
SUCCESS:
{"DID": "1", "SRID": "1493672135917", "BDKToken": null, "MFR": "0", "KSN": "629949
00410005c00014", "Ciphertext": "B1EDF7B94A5F345FDDC35EBED2FFE7A111520A6C82C
CFB0E747D2E38D7D7DCAF55890C9FD899B0A32898F8D4679B1444"}
```

## 6.8—DUKPT Decrypt

Just as KA can be used for DUKPT encryption, it can also decrypt using DUKPT. The same parameters are required as with DUKPT Encrypt, but instead of passing a plaintext, hex-encoded ciphertext is sent to be decrypted:

```
java -jar ccsclient.jar https://demo4.strongkey.com:8181 1 all Abcd1234!
DDEC null 4 62994900410005c00014
B1EDF7B94A5F345FDDC35EBED2FFE7A111520A6C82CCFB0E747D2E38D7D7DCAF55890C9FD
899B0A32898F8D4679B1444 TDES DEK
SUCCESS:
{"DID": "1", "SRID": "1493671673791", "BDKToken": null, "MFR": "0", "KSN": "629949
00410005c00014", "Plaintext": "3B353432343138303432363330373235393D30323132
3130313331373031303635383F33"}
```

KA can be used to generate a MAC over a plaintext using DUKPT-derived MAC keys. The plaintext must be hex-encoded for transport through the web service:

```
java -jar ccsclient.jar https://demo4.strongkey.com:8181 1 all Abcd1234!
DMAC null 4 FFFF9876543210E00001 3430313233343536373839303944393837
MAC_REQUEST
SUCCESS:
{"DID": "1", "SRID": "1493665559600", "BDKToken": "100000000000000003", "MFR": nul
l, "KSN": "FFF9876543210E00001", "MAC": "9CCC78173FC4FB64"}
```

# 7—SEDOS



While KA is capable of encrypting/decrypting structured data elements—such as credit card numbers, social security and bank account numbers, or *JavaScript Object Notation (JSON)* data structures, there are use cases when a company might have massive amounts of data that must be protected at rest; the data might be structured or unstructured, but the quantity of data is far too voluminous to justify encrypting it within an application.

Examples might include the following:

- Centralized log files of hundreds or thousands of servers that must be encrypted at rest, but decrypted when used by *security incident event management (SIEM)* tools
- Vast quantities of medical/health data used in research, which must be protected in accordance with the *Health Information Portability and Accountability Act (HIPAA)* security and privacy rules
- Archived databases with transaction data at e-commerce/gaming/streaming media sites that might must be protected for the privacy of their customers
- Other use cases too numerous to mention

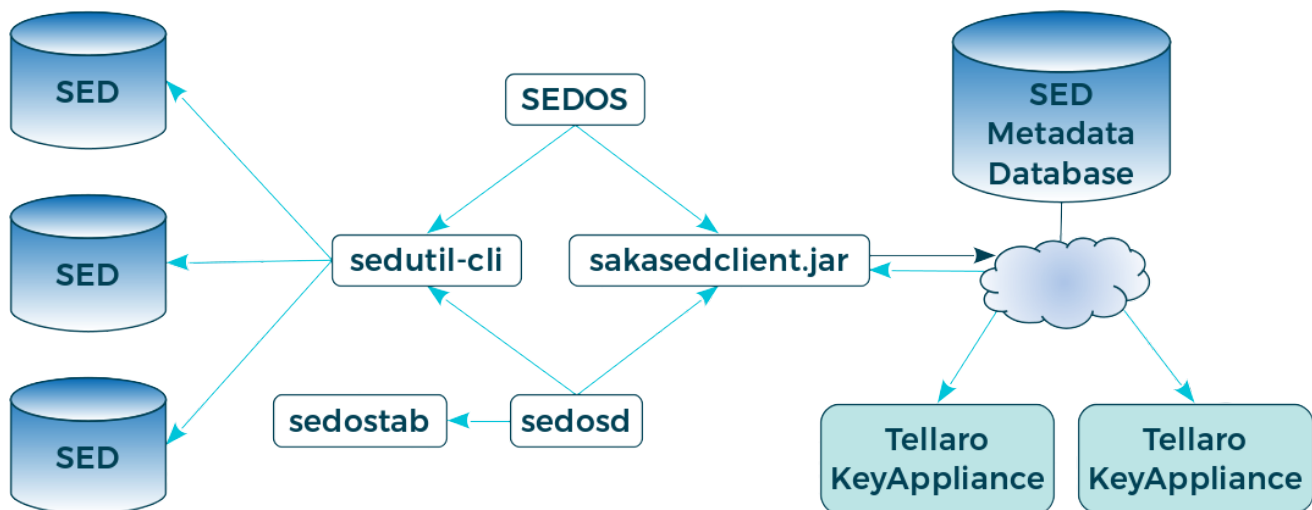
In these situations, a business unit might use *Self-Encrypting Drives (SED)* to store such massive quantities of data, using the high-speed encryption capabilities in the firmware of the SED to protect data.

However, SEDs conforming to the *Trusted Computing Group's (TCG) OPAL* standard, use a secret authentication key to protect access to the *media encryption key (MEK)*. The authentication key must be protected, as it enables the SED to decrypt all data blocks of the drive. While managing and protecting a single—or a few—SEDs is manually feasible, the management of large numbers of SEDs across an enterprise presents complex administrative problems.

To address this administrative problem of securing the authentication keys of a large number of SEDs within a computing environment, StrongKey has created the *Self-Encrypting Drive Orchestrator Script (SEDOS)*—a client tool of the KA. SEDOS orchestrates the use of multiple tools and resources to protect the authentication keys of SEDs. These tools consist of the following:

- `sedutil-cli`, an open-source command line interface tool responsible for interacting with the SED. It is responsible for functions such as: i) initializing the SED; ii) Injecting the authentication key into the SED so it can use the MEK to encrypt/decrypt data blocks
- `sakasedclient.jar`, an open-source command line Java client responsible for communicating with the *SED Metadata Database (SEDMDB)* and KA
- SEDMDB, a MariaDB database which holds information about every SED registered and known to SEDOS
- KA, the secure vault where authentication keys—generated for the SEDs (by SEDOS)—are escrowed for safekeeping so they may be recovered by system tools during a server boot process (to learn more about KA, please review *1—Introduction*)
- `sedosd`, `sedostab` and `/root/pass`, configuration files necessary for this capability
- `SEDOS.sh` (also depicted as SEDOS in this document), the shell script responsible for orchestrating all of the above

A high-level picture of the architecture of SEDOS is shown here:



SEDOS is responsible for the following operations:

- ▶ Initializing a new SED by calling on `sakasedclient.jar` generating a new authentication key, escrowing it on the KA, adding information about the SED to SEDMDB, and calling the `sedutil-cli` to “inject” the authentication key into the SED to initialize it
- ▶ Modifying information about an SED in the SEDMDB, such as its current status, configuration, etc.
- ▶ Changing the authentication key of an SED by calling on `sakasedclient.jar` and `sedutil-cli` to coordinate changing the authentication key and updating the SAK and SEDMDB in the process
- ▶ Removing an authentication key from the SEDMDB and KA, thus ensuring that encrypted data can never be decrypted again
- ▶ Displaying information about SEDs on a given system

`sedosd` is a shell script used during the boot process by the `rc` system on Linux. It also calls on `sakasedclient.jar` and `sedutil-cli` to automatically retrieve the SED's authentication key from the KA and “inject” it into the SED to authorize it for decryption. `sedosd`, subsequently, uses `sedostab` to determine the mount point for the decrypted SED, then mounts the drive/file systems.

Both `sakasedclient.jar` and `sedosd` use `/root/pass` to authenticate themselves against KA to retrieve an SED's authentication key. The `/root/pass` file's contents must be maintained with high levels of security, given that it provides access to decrypted authentication keys from KA. If a site can use other tools/mechanisms to make the KA credential and its passwords to SEDOS or `sedosd` without storing it in `/root/pass`, sites should do so.



## 7.1—The SEDOS Client Software Distribution

StrongKey distributes the demo application as a .ZIP file. After unzipping the *democlients* distribution, the `sakasedclient` executable and source will be contained within a folder called `sakasedclient`.

The `sakasedclient.jar` file contains pre-built executable code. The `sakasedclient-src.zip` file contains the source for this executable as a project folder.

Similarly, the `sedutil-cli` is an ELF 64-bit LSB executable, x86-64, version 1 (GNU/Linux), dynamically linked (uses shared libs) for GNU/Linux 2.6.32; the `sedutil-src.zip` is the source tree snapshot downloaded from GitHub and checked into StrongKey's SubVersion repository for configuration control.

You can choose to build the executable files from scratch if you wish. Both source trees are configured for use with NetBeans IDE 8.x.

## 7.2—Installing the SEDMDB Schema

Before using SEDOS, a MariaDB relational database with the SEDMDB schema must be available so `sakasedclient.jar` can communicate with it. The client software distribution requires the database to have the following:

1. The host running the database must be called `sakadb.dc`. To test/use this distribution, add an entry to `/etc/hosts` and add `sakadb.dc` as an alias to `localhost`.
2. MariaDB must be listening on **Port 3306**—this is the default port.
3. The MariaDB instance must have a database called **sakased**.
4. The *sakased* database must be readable and writable by a database credential with the name of **strongauth** with a password of **AbracaDabra**.
5. The *sakased* database must have a table with the name of **SED\_METADATA**. This table is created by a *structured query language (SQL)* file in the distribution with the name of `sed_metadata.sql`. A text file—`create.txt`—may be “sourced” when connected to MariaDB's MySQL client as the *strongauth* user; it will automatically create the required table. As reference, the table is shown below.

```
create table SED_METADATA (  
  sedid int unsigned not null auto_increment,  
  sedmfr varchar(32) not null,  
  sedsno varchar(32) not null,  
  model varchar(32),  
  capacity int unsigned,  
  fqdn varchar(256),  
  sakadid smallint unsigned,  
  sakatoken varchar(64),  
  oldsakatoken varchar(64),  
  status enum('Active', 'Inactive', 'Other')
```



```

not null,
enroll_date datetime,
rotate_date datetime,
notes varchar(512),
primary key (sedid),
unique index (sedmfr, sedsno),
unique index (sakadid, sakatoken),
index (model),
index (fqdn)
)
engine=innodb;

```

6. Once the database is configured, confirm you can connect to MariaDB via the MySQL tool with the *strongauth* username and *AbracaDabra* password while connecting to the *sakadb.dc* host and *sakased* database. If you connect and see the following schema when you execute *desc sed\_metadata*, you know everything is setup correctly.

Field	Type	Null	Key	Default	Extra
sedid	int(10) unsigned	NO	PRI	NULL	auto_increment
sedmfr	varchar(32)	NO	MUL	NULL	
sedsno	varchar(32)	NO		NULL	
model	varchar(32)	YES	MUL	NULL	
capacity	int(10) unsigned	YES		NULL	
fqdn	varchar(256)	YES	MUL	NULL	
sakadid	smallint(5) unsigned	YES	MUL	NULL	
sakatoken	varchar(64)	YES		NULL	
oldsakatoken	varchar(64)	YES		NULL	
status	enum('Active', 'Inactive', 'Other')	NO		NULL	
enroll_date	datetime	YES		NULL	
rotate_date	datetime	YES		NULL	
notes	varchar(512)	YES		NULL	

## 7.3—Installing the SEDOS Software Distribution

The `install-sedos.sh` script in the distribution installs the SEDOS software on a system with Samsung SEDs, and configures them for use. Specifically, the script performs these operations:

- Installs scripts, libraries, and configuration files that make up the SEDOS distribution on the Linux-based system
- Discovers SEDs on the system and registers them in the SEDMDB, generates unique authentication keys for each SED, escrows the authentication keys within the KA, and initializes the SEDs
- Creates an initialization script—`sedosd`—to enable the recovery of authentication keys from the KA as part of the standard Linux boot process
- Creates a `sedostab` text file to enable `sedosd` to determine where to mount each unlocked SED so the decrypted SED is accessible to applications

The sequence of steps to install the SEDOS distribution manually are as follows. If you plan on installing the distribution through an automated software distribution/installation scheme, the steps can be scripted:

1. Login as `root` to the Linux system with the SEDs.
2. **Create** `/usr/local/software` if it does not exist. **Unzip the SEDOS distribution** into that director; it should extract the distribution into `/usr/local/software/sedos`. Type the following commands:

```
shell> mkdir /usr/local/software
shell> tar xvzf <sedos-distribution-file> -C /usr/local/software
```

3. **Change directory** to `/usr/local/software/sakasedos`.

```
shell> cd /usr/local/software/sakasedos
```

4. Using a text editor, **modify** the `sakaconfig` file. This file contains key value pairs specifying information about KA. If using StrongKey's Demo Client, this is supplied; if using a private KA, use the appropriate parameters.

The `sakaconfig` file has four (4) parameters. **Modify the values** to reflect your testing or production environment and save the file. Make sure the file is owned by `root` and has Read and Write privileges only for the `root` user. In a production environment, this file contains sensitive data that can decrypt authentication keys from KA.

The *sakaconfig* parameters are shown in the table here:

<b>SAKA_FQDN</b>	<p>The <i>fully qualified domain name (FQDN)</i> of the KA where encrypted authentication keys will be escrowed. If using StrongKey's Demo Client, this is usually one or more of the following:</p> <ul style="list-style-type: none"><li>&gt; <a href="https://demo4.strongkey.com">https://demo4.strongkey.com</a></li><li>&gt; <a href="https://demo2.strongauth.com">https://demo2.strongauth.com</a></li><li>&gt; <a href="https://demo3.strongauth.com">https://demo3.strongauth.com</a></li></ul> <p>If using a private KA, this must be that Tellaro's FQDN.</p>
<b>SAKA_DID</b>	<p>The unique Encryption Domain identifier on the KA.</p> <p>If using a private KA, this is the identifier of an encryption configured on your own appliances to store the encrypted authentication keys.</p>
<b>SAKA_USERNAME</b>	<p>Encryption domain username with authorization to call the web service.</p> <p>If you are using your own KA, this is the credential's username created on the private appliances with authorization to call KA web services.</p>
<b>SAKA_PASSWORD</b>	<p>The password of the username with authorization to call the web services.</p> <p>If you are using a private KA, this is the password for the credential with authorization to call web services on the private KA.</p>

5. The `install-sedos.sh` script within this directory is responsible for the installation and requires the following parameters:

<b>saka-config</b>	<p>The location where the install script moves the <i>sakaconfig</i> file modified in <i>Step 4</i>, above. Specify just the directory name of the destination; make sure to use a directory with access restricted only to the <i>root</i> user. The <i>root</i> user's home directory (<i>/root</i>) is a good location.</p>
<b>path-to-java</b>	<p>The JDK location on the target system. <i>Only the Oracle JRE is currently supported.</i> Using a different JDK may result in unpredictable behavior.</p>
<b>-m</b>	<p>Specifies if the drives should be mounted after they have been initialized and formatted. Without this option the drives must be manually mounted after the install script completes.</p>
<b>Mount point [...]</b>	<p>The location each SED is mounted after it is unlocked. If there are multiple drives on the system, multiple mount points can be provided—one per SED.</p> <p>The number of mount points provided <i>must</i> match the number of SEDs on the system, and must be specified in ascending device order.</p> <p>For example, if a server has two SEDs—<i>/dev/sda</i> and <i>/dev/sdb</i>—the two mount points could be <i>/adisk</i> and <i>/bdisk</i>. In this case, <i>/dev/sda</i> will be mounted on <i>/adisk</i> and <i>/dev/sdb</i> on <i>/bdisk</i>. Alternatively, if you chose to provide the mount points as <i>/database/001</i> and <i>/database/002</i>, <i>/dev/sda</i> will be mounted on <i>/database/001</i> and <i>/dev/sdb</i> on <i>/database/002</i>.</p>

An example follows of the `install-sedos.sh` command to move the *sakaconfig* file to */root*, initialize two SEDs, and mount them on */adisk* and */bdisk*:

```
shell> ./install-sedos.sh /root /usr/local/strongauth/jdk8/bin/java -m /adisk /bdisk
```

6. During the installation process the install script copies the `sakasedclient-configuration.properties` file to `/usr/local/strongauth/sakased/etc`. This file is used to specify a syslog server where the `sakasedclient.jar` file will write logs to. This file can be modified with any text editor.

```
shell> vi /usr/local/strongauth/sakased/etc/sakasedclient-configuration.properties
```

Modify the property below with the FQDN of the syslog server:

```
sakasedclient.cfg.property.syslog.host=<syslog-server-fqdn>
```

This file can also be left as is, but must exist in `/usr/local/strongauth/sakased/etc` for the `sakasedclient.jar` file to run correctly.

7. To verify everything is working, the target system with the SEDs must be powered down and subsequently powered up. The SEDs should be automatically unlocked by the `init` script and mounted on `/adisk` and `/bdisk`.

## 7.4—Using the SEDOS Shell Script Manually

While the SEDOS can handle most of the work of initializing and mounting decrypted SEDs automatically, occasionally there is a need to use the tools to perform SED-related tasks manually. The `SEDOS.sh` script in the distribution makes this possible. Specifically, the menu-driven script:

- Allows you to enroll an individual SED into the SEDMDB, generate its authentication key and escrow it on the KA
- Unlocks a specific SED by recovering its authentication key from the KA and “injecting” it into the SED
- Changes an SED's authentication key and updates the SEDMDB and KA automatically
- Deletes an SED from the SEDMDB and its authentication key from the KA, ensuring it cannot be decrypted again; the SED must be reinitialized for the SED to be re-usable
- Displays information about a specific or all SEDs on the system

Before running `SEDOS.sh` you must copy `sakasedclient-configuration.properties` from the distribution to `/usr/local/strongauth/sakased/etc`. This file contains the configuration for the `sakasedclient.jar` to write out messages to a syslog server. For more information on configuring this file, please refer to [1.3—Installing the SEDOS Software Distribution, Step 6](#).

The `SEDOS.sh` script depends on the `sakaconfig` file created during the installation of the SEDOS distribution (as described in the previous section). If the file does not exist, it must be created manually and placed in a secure location on the file system—the `/root` directory is recommended since it is accessible only to the `root` user on a Linux system.

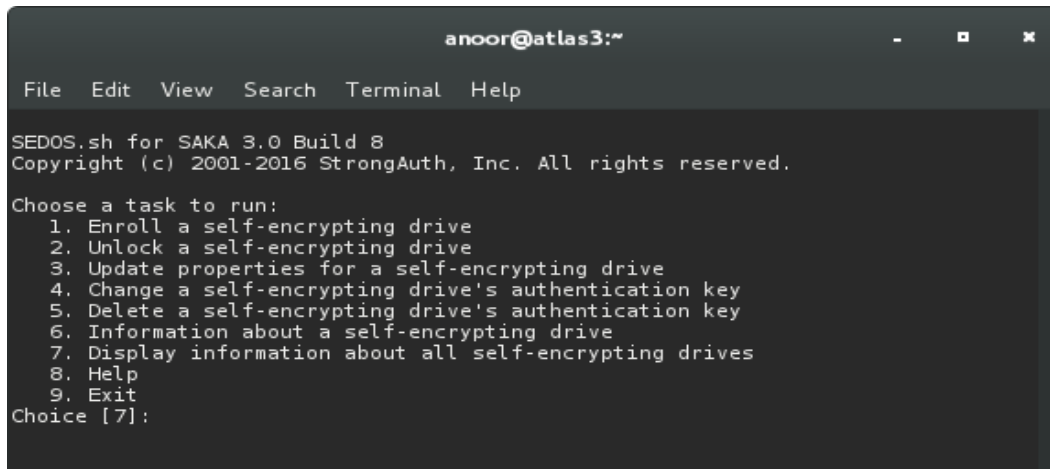
The key value pairs of the `sakaconfig` file are described in the previous section of this chapter; a sample file is available in the SEDOS distribution—it can be modified to reflect the parameters of the KA to be accessed, then placed in its final location.

The `SEDOS.sh` script assumes the `sakaconfig` file is located in `/root` by default; if this is not the case, edit the `SEDOS.sh` script to modify its location. `SEDOS.sh` additionally depends on the `/etc/sedostab` file—normally created if the SEDOS distribution was

installed using the `install-sedos.sh` script from the distribution. If this does not exist, create it manually. The mount point can be any location on the Linux system. The format of this file is as follows, where the serial number of the SED is separated from its mount point by spaces or tabs:

Serial Number of the SED	Mount Point for the SED
SMG1238FTZ8567W	databaseprimary/01
S251NSAG437593D	database/logs/01


Upon executing the `SEDOS.sh` script, a text-based menu displays:



```
anoor@atlas3:~  
File Edit View Search Terminal Help  
SEDOS.sh for SAKA 3.0 Build 8  
Copyright (c) 2001-2016 StrongAuth, Inc. All rights reserved.  
Choose a task to run:  
1. Enroll a self-encrypting drive  
2. Unlock a self-encrypting drive  
3. Update properties for a self-encrypting drive  
4. Change a self-encrypting drive's authentication key  
5. Delete a self-encrypting drive's authentication key  
6. Information about a self-encrypting drive  
7. Display information about all self-encrypting drives  
8. Help  
9. Exit  
Choice [7]:
```

The menu items present the following options:

- Enroll an SED: This option uses `sedutil-cli` to query information about the SED, enrolls it in the SEDMDB (if it does not already exist), generates an authentication key for the SED, escrows the authentication key securely in KA, and initializes the SED using `sedutil-cli`. Once the disk is initialized, it will be in an unlocked state and a file system may be created on it.
- Unlock an SED: This option uses the SEDMDB to look up information about the requested drive. If the SED is in the SEDMDB, its authentication key is recovered from KA and “injected” into the SED using `sedutil-cli`. If the SED is already mounted, choosing this option will generate errors.
- Update properties for an SED: This option changes metadata attributes about a specific SED in the SEDMDB. Attributes that are modifiable are: *manufacturer*, *serial number*, *model*, *capacity*, *FQDN*, *status*, and *notes*.
- Change an SED's authentication key: This option generates a new authentication key for the chosen SED, and updates the SEDMDB and KA to reflect this information.
- Delete an SED's authentication key: This option deletes the SED from the SEDMDB and its corresponding authentication key from KA.

 **NOTE:** Deleting an SED from the SEDMDB or an authentication key from KA will render affected the SED inoperable, as it will not be able to decrypt anything on the SED anymore. The SED will have to be reset to factory settings and reinitialized with a new authentication key to be usable once again. All previously encrypted data on the SED will be permanently lost.

- Information about an SED: This option displays information about a specific SED as reflected in the SEDMDB. This information is only as accurate as that provided to the SEDMDB during the SED's initialization. Should this information not be updated as it changes, it can become desynchronized with the true state of the SED.
- Display information about all SEDs: This option scans the system for *Trusted Computing Group (TCG)* OPAL-compliant drives and displays the following information about them: *device, manufacturer, model, capacity, model number, and serial number*. This information is provided using `sedutil-cli` and reflects what is visible at the operating system layer. Use this option to learn about the manufacturer and serial number of the drive to execute other tasks with `SEDOS.sh`.

## 8—skceclient



In order to test the Tellaro *CryptoEngine (CE)* web services, the appliance comes with a Java-based client application to test the CE Module web service operations. The Java client—called `skceclient.jar`—can be used to call web service operations on a Demo Client on the internet—[demo4.strongkey.com](https://demo4.strongkey.com)—or on a private KA cluster within your network.

The KA Demo Client uses software identical to that of appliances delivered to customers. This allows application development teams to verify code without having to wait for an internal KA implementation. Once verified against the KA Demo Client, the identical code will work against a private KA implementation without changing a single line of code. The only difference would be the URL of the host being called and the parameters passed at runtime. (StrongKey recommends parameterizing the URL of the appliances being called; this will make it easier to point applications to KA during deployment to production).

Apart from the demo Java client, *client URL (cURL)* can also be used to test REST web services exposed by the appliance. This document focuses on the *StrongKey Encryption Engine (EE)* REST services.

1. Encrypt files of any type or size.
2. Decrypt previously encrypted files.
3. Encrypt files of any type or size and store them in the cloud (AWS, AZURE, or EUCALYPTUS).
4. Download a previously encrypted file from the cloud and decrypt it.
5. Ping the CE module.

All the web services accept the following parameters:

<b>svcinfo</b>	Contains parameters required to connect and authenticate the request
<b>fileinfo</b>	Contains metadata about the file
<b>encinfo</b>	Contains metadata about the encryption mechanism
<b>authzinfo</b>	Contains information related to the authorization for a file
<b>storageinfo</b>	Contains information about cloud storage
<b>filedata</b>	InputStream containing the file to be encrypted/decrypted

## 8.1—Parameters

The next sections describe the parameters in more detail. Not all parameters are mandatory for all web services; Look at individual web service sections for detailed information.

### 8.1.1—svcinfol

Parameter	Type	Description
<b>did</b>	String	The domainID to which the client is connecting
<b>svcusername</b>	String	Username for the service credential
<b>svcpassword</b>	String	Password for the service credential username

### 8.1.2—fileinfo

Parameter	Type	Description
<b>filename</b>	String	Name of the file to encrypt/decrypt

### 8.1.3—Encinfo

Parameter	Type	Description
<b>algorithm</b>	String	Algorithm used to encrypt the file
<b>keysize</b>	Int	Size of the key to encrypt the file
<b>uniquekey</b>	Boolean	Indicates whether to use a new key per transaction or to use the EE configured key policy

### 8.1.4—Authinfo

Parameter	Type	Description
<b>username</b>	String	Username of the account trying to encrypt/decrypt a file
<b>userdn</b>	String	Optional userDN
<b>authgroups</b>	String	Hyphen-separated (-) list of groups to determine the authorization during decryption; only users belonging to the specified groups will be able to decrypt
<b>requiredauthorization</b>	Int	(Optional) Indicates the number of FIDO authorizations required to decrypt a file



## 8.1.5—storageinfo

Parameter	Type	Description
cloudtype	String	Type of cloud used: Eucalyptus, AWS, or AZURE
cloudname	String	Name of the cloud instance to be used
cloudcontainer	String	Name of the cloud bucket where the encrypted file will be stored
accesskey	String	Cloud access key
secretkey	String	Cloud secret key
cloudcredentialid	String	KA Token to retrieve the cloud credentials

To test this against the KA Demo Client, download the *Certificate Authority (CA)* file from <https://letsencrypt.org/certs/letsencryptauthorityx3.pem.txt> and use it to call web services using cURL. To test this against your own instance of EE, replace all the values in the demo with values corresponding to your environment.

## 8.2—Encrypt

To encrypt a test file, type the following command. It makes a POST to encrypt the `abc.txt` document and save it locally as `abc.txt.zenc`.

```
$ demo:~/topaz4> java -cp skceclient.jar EncryptionEngine
https://demo.strongkey.com 1 service-cc-ce Abcd1234! E
/usr/local/strongauth/skce/etc/abc.txt -lu encryptdecrypt -lg
    "cn=EncryptionAuthorized,did=1,ou=groups,ou=v2,ou=SKCE, \
    ou=StrongAuth,ou=Applications,dc=strongauth,dc=com"
-wp SOAP
skceclient.jar 4.1 (Build 163)
Copyright (c) 2001-2019 StrongKey. All rights reserved.
Calling Encrypt at https://demo.strongauth.com ...
SHA256 message digest received   :
f39884240904a896dc9b5dcd54fa4084ddeef0ec3a172d21565aca9e4e00905b
SHA256 message digest calculated :
f39884240904a896dc9b5dcd54fa4084ddeef0ec3a172d21565aca9e4e00905b
SHA sum verified! Encryption successful
Done!
```

## 8.2.1—Decrypt

To decrypt the encrypted test file, type the following command. It makes a POST to decrypt the `abc.txt.zenc` document and save it locally as `abc.txt`. The *Decrypt* call does not require *encnfo*.

```
$ demo:~/topaz4> java -cp skceclient.jar EncryptionEngine
https://demo.strongkey.com 1 service-cc-ce Abcd1234! D abc.txt.zenc -lu
encryptdecrypt -wp SOAP
skceclient.jar 4.1 (Build 163)
Copyright (c) 2001-2019 StrongKey. All rights reserved.
Calling Decrypt at https://demo.strongkey.com ...
SHA256 message digest received :
883a1d4f11c8d5080df750209391c005677c53bd1821eb0c4c49ed8603dc72dd
SHA256 message digest calculated :
883a1d4f11c8d5080df750209391c005677c53bd1821eb0c4c49ed8603dc72dd
SHA sum verified! Decryption successful
Done!
```

## 8.2.2—Encrypt to Cloud

To test how *encrypt to cloud* can encrypt a file and save it to a bucket in the cloud, execute the following command. It encrypts `abc.txt` and saves it to a bucket in the configured cloud. The output will display the hash of the encrypted file along with the storage location:

```
$ demo:~/topaz4> java -cp skceclient.jar EncryptionEngine
https://demo.strongkey.com 1 service-cc-ce Abcd1234! CE
/usr/local/strongauth/skce/etc/abc.txt -lu encryptdecrypt -lg
    "cn=EncryptionAuthorized,did=1,ou=groups,ou=v2,ou=SKCE, \
    ou=StrongAuth,ou=Applications,dc=strongauth,dc=com"
-f cloud.properties -wp SOAP
skceclient.jar 4.1 (Build 163)
Copyright (c) 2001-2019 StrongKey. All rights reserved.
Calling Encrypt to Cloud at https://demo.strongkey.com ...
Response: Uploaded abc.txt.zenc to Eucalyptus. Instance name: mywalrus;
Bucket: testskce
SHA256 message digest :
47816b9eab91549ac29eb0f8cee49d6d4a395bc3d89321495f9f082922d120fd
Done!
```

## 8.2.3—Decrypt from Cloud

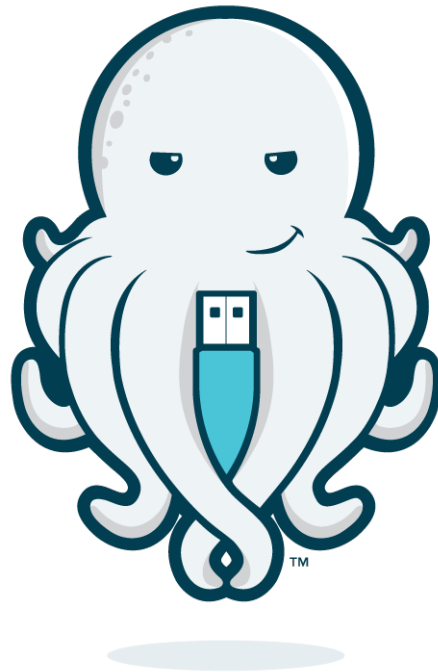
To test how *decrypt from cloud* first downloads the encrypted file and then decrypts it, execute the following command. It downloads `abc.txt.zenc` from the cloud and decrypts it. Filedata is not specified, as the file is downloaded from the cloud and the response will be saved as `abc.txt`.

```
$ demo:~/topaz4> java -cp skceclient.jar EncryptionEngine
https://demo.strongkey.com 1 service-cc-ce Abcd1234!
    CD abc.txt.zenc -lu encryptdecrypt - cloud.properties -wp SOAP
skceclient.jar 4.1 (Build 163)
Copyright (c) 2001-2019 StrongKey. All rights reserved.
Calling Decrypt from Cloud at https://demo.strongkey.com ...
SHA256 message digest received   :
883a1d4f11c8d5080df750209391c005677c53bd1821eb0c4c49ed8603dc72dd
SHA256 message digest calculated :
883a1d4f11c8d5080df750209391c005677c53bd1821eb0c4c49ed8603dc72dd
SHA sum verified!
Done!
```

## 8.2.4—Ping

The following example shows how to run the ping web service which can be used as a health check for the CE module. The output will display the status of the CE.

```
$ demo:~/topaz4> java -cp skceclient.jar EncryptionEngine
https://demo.strongkey.com 1 skceping Abcd1234! P -wp SOAP
skceclient.jar 4.1 (Build 163)
Copyright (c) 2001-2019 StrongKey. All rights reserved.
Calling Ping at https://demo.strongkey.com ...
Response received  :
<SKCE:Component>
  <SKCE:Property>
    <SKCE:PropertyName>Name</SKCE:PropertyName>
    <SKCE:PropertyValue>SKCE</SKCE:PropertyValue>
  </SKCE:Property>
  <SKCE:Property>
    <SKCE:PropertyName>Version</SKCE:PropertyName>
    <SKCE:PropertyValue>SKCE Version 2.0 (Build
163)</SKCE:PropertyValue>
  </SKCE:Property>
  <SKCE:Property>
    <SKCE:PropertyName>Authentication</SKCE:PropertyName>
    <SKCE:PropertyValue>Successful</SKCE:PropertyValue>
  </SKCE:Property>
  <SKCE:Property>
    <SKCE:PropertyName>Encryption Status</SKCE:PropertyName>
    <SKCE:PropertyValue>Successfully encrypted
/usr/local/strongauth/skce/etc/abc.txt </SKCE:PropertyValue>
  </SKCE:Property>
  <SKCE:Property>
    <SKCE:PropertyName>Decryption Status</SKCE:PropertyName>
    <SKCE:PropertyValue>Successfully decrypted encryption output
</SKCE:PropertyValue>
  </SKCE:Property>
</SKCE:Component>
Done!
```



# **S T R O N G K E Y**

20045 Stevens Creek Boulevard Suite 2A  
Cupertino, CA 95014  
USA