CLEAR CRYPTOSYSTEM

String Encryption SDK – Java Edition

Abstract

Start securing content to quantum-safe levels of encryption in less than two minutes.
Easy to follow SDK guide and steps to get you encrypting Strings fast.

Quantum Knight
info@quantumknight.io

Table of Contents

Section 1 – String (text-based) Encryption

Overview

This document provides detailed instructions for using the "**String Interface**" of the CLEAR Cryptosystem ("CLEAR"). The String interface is a convenience method built on top of the root binary interface and is designed to make encrypting textual data more convenient. Additional convenience (for Strings) is achieved via the incorporation of UTF-16 encoding of cleartext binary input and Base64 encoding of binary ciphertext output. UTF-16 Unicode character encoding widens all characters to two [2] bytes to support multiple languages and advanced character-sets. In this way, data encrypted using the String interface will result in a cyphertext that is larger (in byte size) than the original. If looking to achieve encryption that is the same size as the original, please utilize the CLEAR streaming interface whereby you may directly encrypt byte arrays.

```
// INITIALIZE "STRING" INTERFACE
CLEARString cs = CLEAR.clearString();       // FACTORY PATTERN – GET A REFERENCE TO CLEAR API (STRING ENCRYPTION / DECRYPTION INTERFACE)

// ENCRYPT A STRING – SYSTEM GENERATED KEY
CLEARResult cr = cs.encrypt(testData, CLEAR.STRENGTH_2048_BIT, SingleJobEncryptionSysGenKeyMode.SINGLE_JOB_SYSTEM_GEN_KEY);
```

Figure 1 - Example of CLEAR String Encryption

As illustrated by the example above, CLEAR String encryption requires only **two lines of code in all cases**. In the following sections we will explore the various input parameters, response outputs, and unique

modes of operation available with CLEAR String encryption. String encryption is intended for use within online applications, messaging systems, and anywhere that unique and strong protection of textual data or tokenization is required. String Encryption provides various methods for applying security to text in a highly performant manner.

System Generated Encryption Keys (SGK)

There are only relatively few publicly available Symmetric encryption ciphers in the world today. The ones that can be consumed in Java are typically already included within the Java Runtime Environment itself, made available for use by developers as part of the Java Cryptography Extension (JCE) interface. One may also find encryption ciphers online at "the legion of the bouncy castle", via www.bouncycastle.org.[1]

A common thread amongst the open-source symmetric ciphers is the way in which they are instantiated programmatically. In many cases, user-level encryptions begin when an operator generates some form of password or passphrase (and optional salt) that will ultimately become their encryption key. The initialization sequence for most symmetric ciphers (at least in the JCE) is one that requires an initialization vector (IV), a salt, and a secret key. These implementations vary subtly by cipher; however, they typically involve the programmatic generation of entropy of 256 bits or less.

As it happens, mishandling of the initialization-phase entropy (as mentioned above) is one of the larger contributing factors to weakness in encryption implementations. For example, it wouldn't be AES-256 itself that provides a weakness (algorithmically); rather, it is more frequently poor user implementation or hard-coding of a salt or IV that introduces weakness into security systems. Key storage, secure vaulting, and proper key-handling in symmetric encryption is extremely important.

The CLEAR interface is designed for ease of use and minimal required coding. The generation of entropy (randomness) in Java is frequently accomplished using the **SecureRandom** class for what is deemed to be cryptographically secure random number generation. Secure Random can produce random integers as well as binary byte-array sequences. In the *default configuration*, the Java implementation of CLEAR also utilizes Java Secure Random for its System Generated Encryption Keys ("SGK").

SGK do not *require* a user-defined key or password to function. CLEAR will automatically generate strong entropy with the aforementioned "cryptographically secure random number generator" (CSRNG) as the starting place for all encryptions. Like other symmetric algorithms, CLEAR SGK are internally constructed in way that incorporates an initialization vector, a salt, and a secret key vector. Each of these precursory artifacts are made directly from a series of 64-bit long integers, produced directly by CSRNG as their source of entropy. For additional detail on the inner-workings and mechanics of CLEAR encryption, please see our white paper.

Perhaps you're now asking yourself the question, "How can RNG produced by CSRNG being considered 'Quantum Strength' in the CLEAR Cryptosystem?" If that is the case, we're going to get to the answer to that important question on the very next page.

Professional cryptanalysis and academic peer review applied to CLEAR over multiple rounds and multiple years have conclusively come back with the same consistent results.

"***The length of the key material ('key-space') is mathematically equivalent to the bit-strength of the encipherment.***"

In other words, 512 bits of key material means definitive production of a 512bit encipherment.

Java Secure Random is the default CSRNG

Grover's Algorithm[2] was developed in 1996 by Lov Grover as a mechanism for intended for quadratic performance boost of searching through an unsorted database. Ostensibly, an unintended consequence of Grover's Algorithm was its ability to reduce the time complexity of brute force attacks on round-reduced block ciphers.

With the utmost respect for the Advanced Encryption Standard (AES) and 256bit security that was drafted in the late 1990's and ratified by NIST in 2001; we are now looking at the need for new solutions that can contend with computational factors that did not exist when AES was created.

When CLEAR produces 512 bits of CSRNG as the "fuel" (and starting place) for its encryption operation; the net result is encryption that can be measured at 512 bits of security protection. As we together enter the next generation of cyber security challenges, we are facing the power of qubit computers as well as the advancement of AI algorithms that capable of round-reducing encryption ciphers.

Plug-N-Play Random Number Generator

OK – so the fate of your security rests in the hands of the Java Secure Random number generator? Really? No, definitely not. Certainly, the easiest way to use CLEAR is to run it with as few parameters as possible and let the default onboard CSRNG do the work of producing your initial entropy, the "fuel" for encryption. However, this alone does not allow for the ultimate in flexibility.

We understand that enabling plug-and-play RNG is the best way to build trust and give you full control over the *input* into your encryption jobs. An essential feature of CLEAR is the ability to plug-in your own entropy source (random number generator). The "*… if you don't like ours, use your own.*" ethos is core to the way we see CLEAR being used to guarantee absolute data privacy and true cybersecurity.

CLEAR has been constructed with one "built-in" plugin for external RNG, as well as an open-SDK interface for adding RNG input from external sources. The built-in plugin allows CLEAR to source entropy from a reputable online and open-source provider called www.random.org[3].

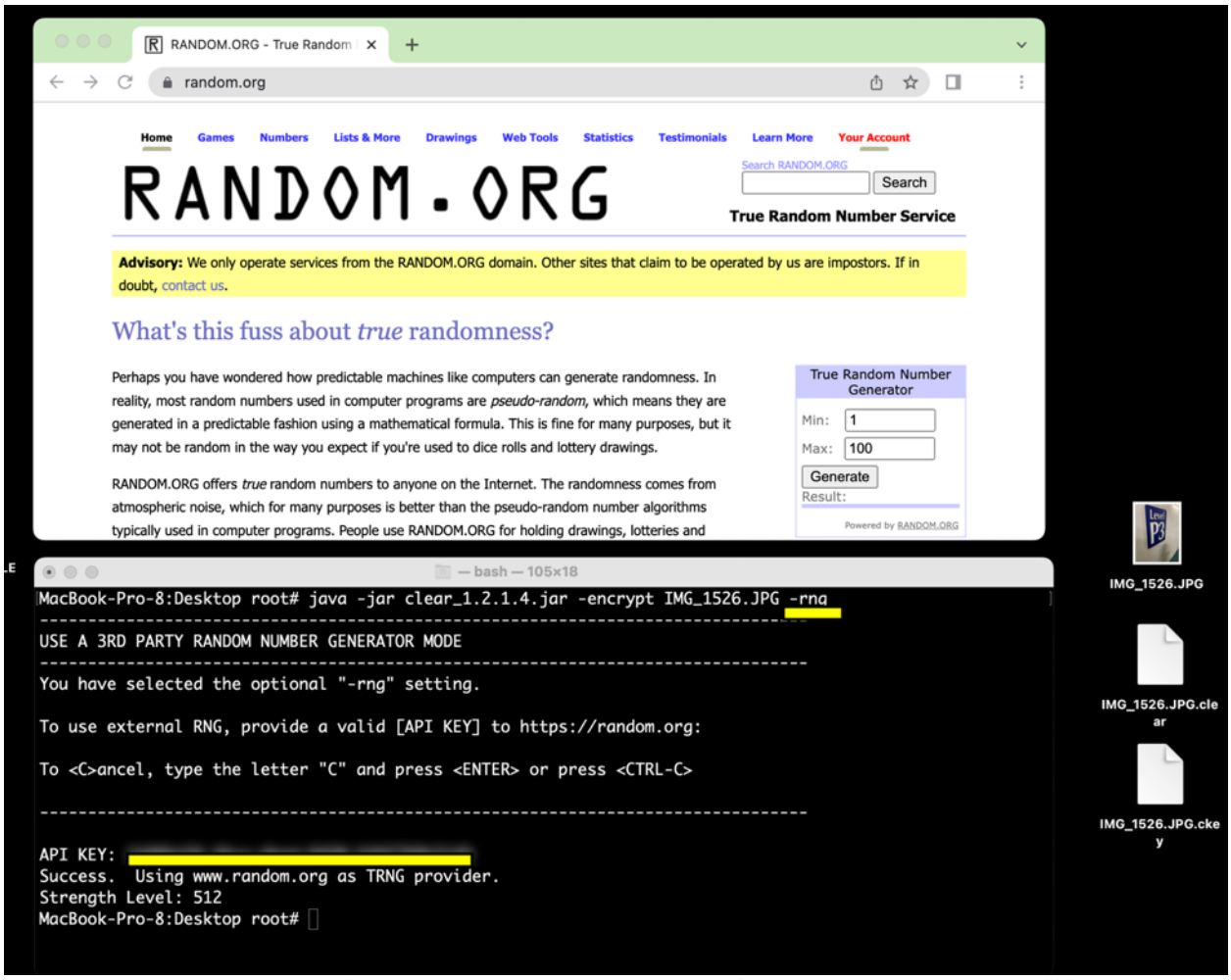


Figure 2 – Example of invoking Random.ORG pluggable RNG from CLEAR Command-Line Interface

The example shown in Figure-2 illustrates the use of the CLEAR "**-rng**" flag (at the CLI) as well as the input of a user's own "API Key" to replace Java CSRNG with online RNG from www.random.org as the source for encryption entropy.

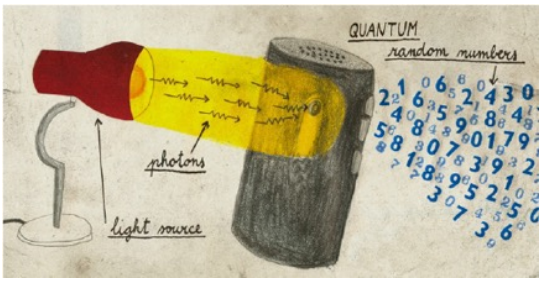


Figure 3 – Generating random numbers from light.

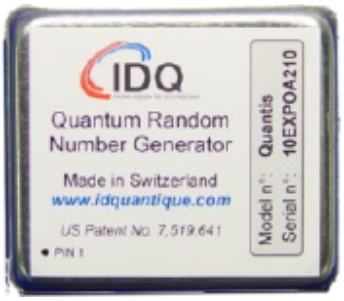


Figure 4 – Hardware-based RNG

CLEAR is also designed to work with external hardware-based random number generators. Often referred to as "quantum random number generators" or "QRNG", products such as ID-Quantique[4] may be directly integrated into CLEAR via localized USB connection or on-a-chip (motherboard) type connection.

Programmatically, changing the random number generator in CLEAR can be done with one simple line of code. As shown below, the developer SDK allows a picklist of external RNG types:

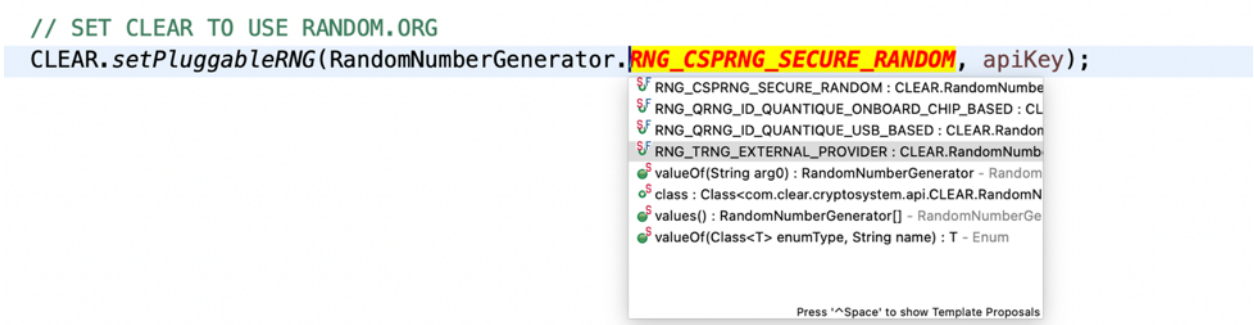


Figure 5 - Programmatic Selection of External RNG

Enough of a preamble …

LET'S START ENCRYPTING THINGS!

Section 2 – Using the CLEAR String SDK

String Encryption – System Generated Keys (SGK) – (Basic Mode)

The very best starting point to learn CLEAR (in our opinion) is the basic String encryption with System Generated Key ("SGK"). This section breaks down each aspect of the operation.

Let's Encrypt:

CLEAR Encryption SDK is made to be SIMPLE to use. You can encrypt data in two [2] lines of code.

```
// LINE-1:   INITIALIZE "STRING" INTERFACE
CLEARString cs = CLEAR.clearString();          // FACTORY PATTERN - GET A REFERENCE TO CLEAR API

// LINE-2:   ENCRYPT A STRING - USE SYSTEM GENERATED KEY (SGK)
CLEARResult cr = cs.encrypt(_CLEARTEXT_TEST_DATA, CLEAR.STRENGTH_2048_BIT, SingleJobEncryptionSysGenKeyMode.SINGLE_JOB_SYSTEM_GEN_KEY);

System.out.println(cr.getCipherTextString());   // THIS IS THE ENCRYPTED RESULT!
```

Let's Decrypt:

CLEAR Decryption SDK is made to be SIMPLE to use. You can decrypt data in two [2] lines of code.

```
// PRE-REQUISITE ITEM
String myKeyMaterial = previousJob.getKeyMaterial();  // KEY MATERIAL PRODUCED BY ORIGINAL ENCRYPTION

// LINE-1:   INITIALIZE "STRING" INTERFACE
CLEARString cs = CLEAR.clearString();                 // FACTORY PATTERN - GET A REFERENCE TO CLEAR API

// LINE-2:   ENCRYPT A STRING - USE SYSTEM GENERATED KEY (SGK)
CLEARResult cr = cs.decrypt(_CIPHERTEXT_DATA, myKeyMaterial, SingleJobDecryptionMode.SINGLE_JOB_KEY);

System.out.println(cr.getClearTextString());          // THIS IS THE DECRYPTED RESULT!
```


Now, let's dive a little deeper. From here we will break these operations down into each of their individual components. The following illustration highlights the various features and options included in each of the interfaces and objects used in the encryption and decryption operations mentioned on the previous page.

PART 1 – INITIALIZATION:

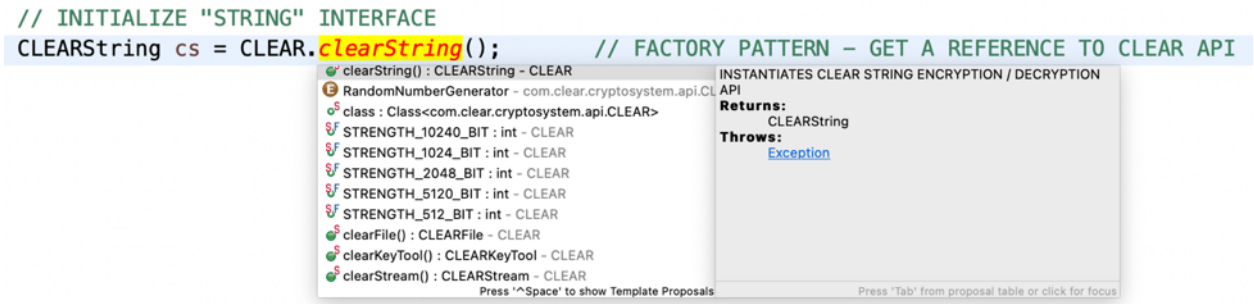


Figure 6 - Initialize the CLEAR String Interface

CLEAR naturally exposes its available features via "picklist" or drop-down by way of automatic code completion in the Integrate Development Environment ("IDE"). As shown in Figure-7 above, CLEAR can be initialized into String handling mode of operations by picking the CLEARString ("CS") Interface from the CLEAR application SDK. As mentioned in Section 1, the CS Interface is designed to facilitate text handling with accommodation for UTF-16 and Unicode special characters.

PART 2 – EXAMINING THE INTERNALS OF CLEAR-STRING INTERFACE

Once we have a reference (handle) to the CS interface, we can use code-completion in the IDE to inspect the available features and functions as a picklist or drop-down selection. To do this in the Eclipse IDE, type "cs.encrypt" and press ctrl-space.

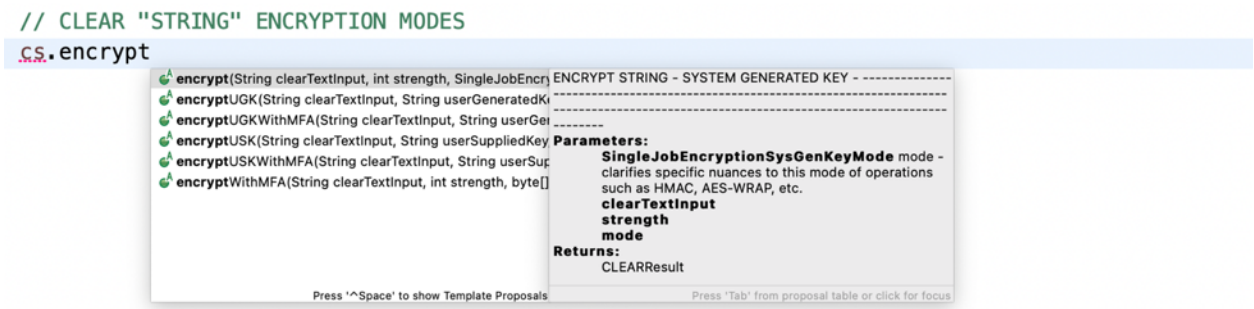


Figure 7 - Inspecting the functions available in CLEAR String SDK

As shown in Figure-8 above, there are six [6] possible encryption functions within the CS Interface. The System Generated Key (SGK) mode is the default usage of CLEAR and thus referenced only as "encrypt", shown as the highlighted row at the top of the picklist.

Encrypting with SGK mode presents a function with three [3] possible input parameters. JavaDoc for the SGK mode is articulated as follows:

```
/**
 * ENCRYPT STRING - SYSTEM GENERATED KEY -
 * ---------------------------------------------------------------
 * @param clearTextInputBinary - cleartext of variable length
 * @param strength - user specified CLEAR Encryption Strength mode
 * @param mode - clarifies specific nuances to this mode of operation such as HMAC, AES-WRAP, etc.
 * @return CLEARResult
*/
public CLEARResult encrypt( String clearTextInput,
                            int strength,
                            SingleJobEncryptionSysGenKeyMode mode);
```

INPUT: Parameters / Arguments:

- **clearTextInput** – this is any string that you wish to encrypt. It is assumed to be in "cleartext".
- **strength** – this is the bit strength you wish to use for encryption.
  - Options are: 512, 1024, 2048, 5120, 10240.
- **mode** – This is an enum with an additional picklist of "sub-features" and post-processing options.

OUTPUT: Return Type:

- **CLEARResult** – this is an object wrapper that contains the results of the encryption operation.

PART 3 – ENCRYPTION "MODE" INPUT PARAMETER

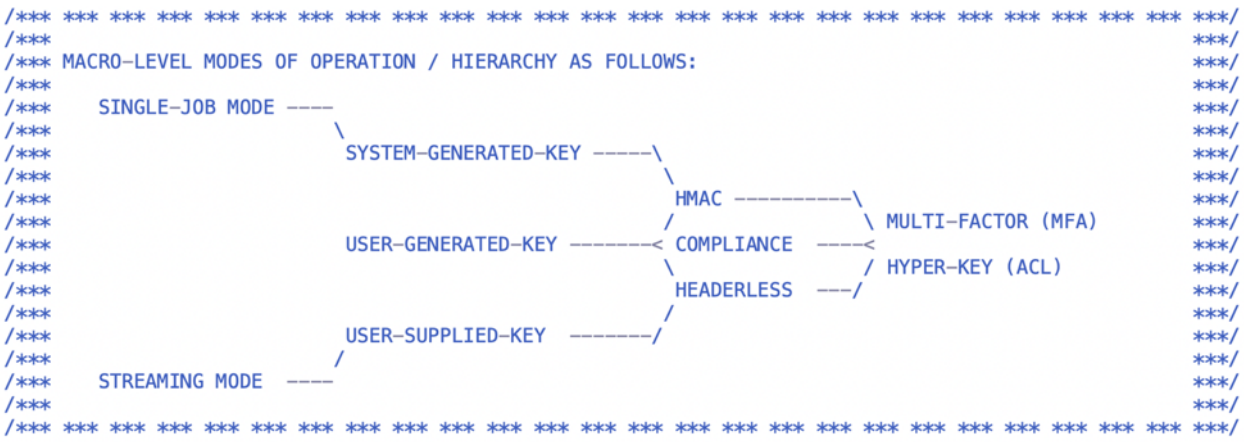


Figure 8 - Overview - CLEAR Modes of Operation

As illustrated in Figure-9, CLEAR encryption can be executed across a variety of modalities according to a hierarchical set of "modes of operation". All CLEARString Interface modes of operation fall under the category of "Single Job Mode". Single Job means "one contiguous encryption operation". This is different to "Streaming" modes of operation that allow for multiple "continuous" encryption (on a single key), such as the encryption of live data packets. More on that in the Streaming Encryption documentation.

System Generated Keys (SGK) – (Basic Mode) simply implies that sub-functions (to the right of key-type in Figure-9) are not utilized.

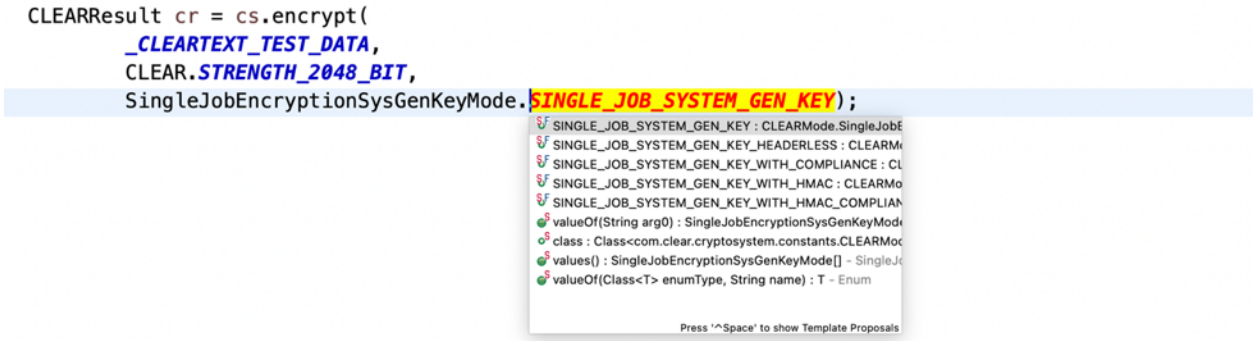


Figure 9 - Selecting Mode of Operation - Encryption

The CLEARResult object encapsulates important information returned by the encryption operation. As shown below in Figure-11, the basic SGK mode for String Encryption will include a "**cipherTextString**" as well as "**keyMaterial**".

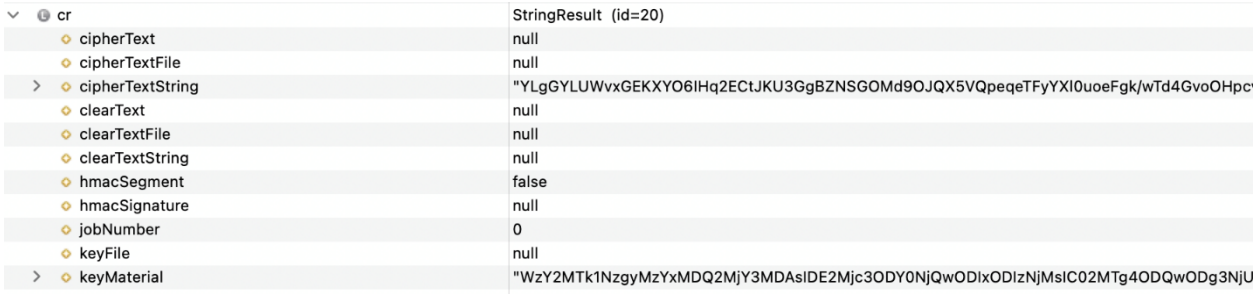


Figure 10 - Live inspection of the CLEAR Result Object after Encryption

The "**keyMaterial**" element contains the actual CLEAR Hyper-Key compliant format material that must be saved / stored for subsequent use with decryption operations. Without this key material, the ciphertext cannot be decrypted again in the future.

Different uses of CLEAR Encryption will return ciphertext differently. All CLEARString Interface encryption operations will return ciphertext within the element "cipherTextString". Contrast this with Streaming or local File-based encryption modes that will place their respective results in "cipherText" or "cipherTextFile", respectively. More detail on those modes of operation may be found in the documentation on Streaming and File based encryption. Unique to CLEARString encryption is that the net-result (cipherTextString) is automatically base64 encoded for convenient use directly as a String.

PART 4 – DECRYPTING THE STRING

In this example, we pass the ciphertext produced by the encryption operation directly into the input parameter of the decrypt function. Further, we also pass the key material generated in the first stop directly into the second parameter / argument for decryption.

```
// DECRYPT A STRING – WITH THE SYSTEM-GENERATED KEYSTRING PRODUCED BY ENCRYPTION
cr = cs.decrypt(cr.getCipherTextString(),
                cr.getKeyMaterial(),
                SingleJobDecryptionMode.SINGLE_JOB_KEY);
```

Figure 11 - Selecting Mode of Operation - Decryption

As illustrated in Figure-12 above, the decryption functions have fewer selections than encryption. Decryption of String data still follows the "Single Job Mode" concept; however, we only need to take care to specify whether the decryption is "Headerless" or in "Compliance Mode". For basic string decryption, simply select "Single Job Key". More detail and information about Headerless and Compliance modes of operation is covered in their respective sections in this document.

---

cr	StringResult (id=20)
cipherText	null
cipherTextFile	null
cipherTextString	null
clearText	null
clearTextFile	null
clearTextString	"The quick brown fox jumps over the lazy dog!! Some UTF-16 Asian based 中国人 characters
hmacSegment	false
hmacSignature	null
jobNumber	0
keyFile	null
keyMaterial	null

Figure 12 - Live inspection of the CLEAR Result Object after Decryption

As can be observed in Figure-13 above, the result of the reversal of ciphertext (decryption) will be a String value in the "**clearTextString**" field of CLEARResult object. The Quantum Knight team feels it is important to mention that decryption operations will always function in 100% of circumstances, irrespective of whether a license-key is found.

If you do not have a valid license-key for CLEAR, then the usual results of an encryption operation (clearTextString and keyMaterial) will be returned as null. From a developer's perspective, please take care to check for Null Pointer Exception (NPE) in your code when implementing encryption that is license dependent.

String Encryption – System Generated Keys (SGK) – (Various Strength Levels)

A feature of CLEAR is the ability to generate encryption at different key-length (encryption strength) levels. CLEAR offers the following encryption modes:

- 512bit Encryption Mode - Post-Quantum Commercial Strength
- 1024bit Encryption Mode - Military Communications Strength
- 2048bit Encryption Mode - TOP-SECRET – Signal Intelligence Communications
- 5120bit Encryption Mode - Satellite / Space / Submarine Communications
- 10240bit Encryption Mode - Long-Term Archival Strength (20 year +)

```
private static int[] _STRENGTHS   =  {  CLEAR.STRENGTH_512_BIT,
                                        CLEAR.STRENGTH_1024_BIT,
                                        CLEAR.STRENGTH_2048_BIT,
                                        CLEAR.STRENGTH_5120_BIT,
                                        CLEAR.STRENGTH_10240_BIT

};
```

The strength parameter may be free-form input as an integer argument. We recommend using the onboard CLEAR constants for picking strength settings. Entering an unsupported integer will result in the default encryption strength of 512bit.

String Encryption – System Generated Keys (SGK) – (with HMAC)

CLEAR incorporates Authenticated Encryption with Associated Data (“AEAD”) with Hash based Message Authenticate Code ("HMAC") as a mechanism that supports a form of signed encryption that is resistant to malicious injection and tampering. AEAD is a variant of Authenticated Encryption (“AE”) that allows a recipient to check the integrity of both the encrypted and unencrypted information in a message. AEAD binds associated data (AD) to the ciphertext and to the context where it is supposed to appear so that attempts to "cut-and-paste" a valid ciphertext into a different context are detected and rejected.

During decryption operations, signed AEAD / HMAC authentication will compare the HMAC found near the prefix of the ciphertext with a freshly generated HMAC that is produced by decryption. If the Message Authentication Codes (“MAC”) do not match exactly, then the signature is determined to be invalid and thus implies a possible tampering of the ciphertext.

Encrypting in HMAC Mode

Encrypting a String in HMAC with AEAD is almost identical to encrypting in standard (basic) mode. To use HMAC during encryption, select “**SINGLE_JOB_SYSTEM_GEN_KEY_WITH_HMAC**” for the mode selection.

```
// ENCRYPT A STRING – SYSTEM GENERATED KEY
CLEARResult cr = cs.encrypt(_CLEARTEXT_TEST_DATA,
                            CLEAR.STRENGTH_2048_BIT,
                            SingleJobEncryptionSysGenKeyMode.SINGLE_JOB_SYSTEM_GEN_KEY_WITH_HMAC);
```

Figure 13 - Encrypt a String SGK in HMAC Mode

Important note: Encrypting in HMAC mode requires more computational overhead than non-HMAC based encryptions. HMAC Mode will add **72 bytes of additional data** to the ciphertext output.

Decrypting in HMAC Mode

Decrypting a String that was encrypted in HMAC mode is entirely identical to the standard decryption operation pictured above in Figure-12. CLEAR automatically detects HMAC in the ciphertext header and will attempt to validate HMAC authentication whenever it exists.

Important note: Failed HMAC authentication (during decryption) will result in a failure to decrypt with and error message and null response output.

String Encryption – System Generated Keys (SGK) – (FIPS 140-2 Compliance Mode)

FIPS 140-2 (Federal Information Processing Standard 140-2) is a U.S. government standard established by the National Institute of Standards and Technology (NIST) to ensure the security and strength of cryptographic modules used in hardware and software products.

The standard provides a rigorous set of requirements and guidelines for cryptographic modules and their underlying algorithms to protect sensitive information. FIPS 140-2 validation is the process of testing and certifying a cryptographic module against the FIPS 140-2 standard.

The validation is carried out by NIST-accredited Cryptographic Module Testing (CMT) laboratories. Upon successful validation, the tested product is added to the NIST's Cryptographic Module Validation Program (CMVP) list, indicating that the product is compliant with FIPS 140-2 requirements.

FIPS 140-2 module validation for CLEAR Cryptosystem can be found here:

https://csrc.nist.gov/projects/cryptographic-module-validation-program/certificate/4482

Key-wrapping is a technique used to protect cryptographic keys by encrypting them with a strong key encryption algorithm. AES-256 (Advanced Encryption Standard with 256-bit keys) is one of the approved algorithms for key-wrapping in FIPS 140-2. Using key-wrapping can help extend compliance to non-validated cryptographic modules by encapsulating sensitive key material within the secure boundaries of a FIPS 140-2 validated module.

By following this approach, the sensitive key material remains protected by the FIPS 140-2 validated module, and the non-validated module only handles encrypted (wrapped) keys, thereby extending a level of FIPS 140-2 compliance to the non-validated module. However, it is crucial to note that this strategy does not make the non-validated module fully FIPS 140-2 compliant.

CLEAR is packaged with the Crypto-Compliance CCJ JAR as an optional-use accessory to support total FIPS 140-2 compliance as a validated module. When added to the classpath of CLEAR and executed in compliance mode, AES encryption is injected into the inner core of the CLEAR encipherment process. Irrespective of the use of the CCJ JAR, CLEAR applies AES-256 key-wrapping as a post-processing step whenever executed in compliance mode.

Encrypting in Compliance Mode

Encrypting a String in Compliance mode is almost identical to encrypting in standard (basic) mode. To use Compliance mode during encryption, select "**SINGLE_JOB_SYSTEM_GEN_KEY_WITH_COMPLIANCE**" for the mode selection.


```
// ENCRYPT A STRING – SYSTEM GENERATED KEY
CLEARResult cr = cs.encrypt(_CLEARTEXT_TEST_DATA,
                            CLEAR.STRENGTH_2048_BIT,
                            SingleJobEncryptionSysGenKeyMode.SINGLE_JOB_SYSTEM_GEN_KEY_WITH_COMPLIANCE);
```

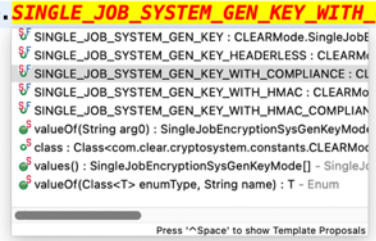


Figure 14 - Encrypt a String SGK in FIPS 140-2 Compliance Mode

Important note: Encrypting with CLEAR in compliance mode will result in the generation of a key format that is different to non-compliance modes. In CLEAR File-based encryption, the difference is most easily recognizable as being saved under a different file extension. E.g. ".ckey" versus ".ccom" versus ".ccomf". Please see the CLEAR-File SDK documentation for more detail. In this case of CLEAR-String SDK, however; compliance-mode key output is largely indistinguishable from non-compliance mode output.

cr	StringResult (id=27)
cipherText	null
cipherTextFile	null
cipherTextString	"L7+BgYlQy0kCSy7LrHcmd1aCkT01b1Sxt3s4YqO6/AXr9wjw1AgWCxpskoblDvmBZxd1
clearText	null
clearTextFile	null
clearTextString	null
hmacSegment	false
hmacSignature	null
jobNumber	0
keyFile	null
keyMaterial	"IBgAAHvTMsPf20P7vWuHQetecZQQI7PXkIQc4MNwX3qvhcsyGRRBQU9vdRpLcXU

Figure 15 - CLEAR Result for a Compliance-Mode Operation (without CCJ JAR)

cr	StringResult (id=30)
cipherText	null
cipherTextFile	null
cipherTextString	"6k+IMICY4WzfIzSw44PRWD7KRjURRFG2Pt7cglz4MchkUznQI9mCMDtxuvlj0/MIGe
clearText	null
clearTextFile	null
clearTextString	null
hmacSegment	false
hmacSignature	null
jobNumber	0
keyFile	null
keyMaterial	"EBgAAM7CQ79b61CubJLVMPPAeMMB/8UCpstA9EpEpk/FlsgnTF1jH3Z6exYtCkBe

Figure 16 - CLEAR Result for a Compliance-Mode Operation (with CCJ JAR)

As can be observed in Figure-16 and Figure-17, respectively, the results from compliance operations cannot be directly observed. For this reason, it is very important to take care and store keys in a location and such a way that their type is remembered for future use. Keys made in compliance modes can only be decrypted using decryption operations being run (similarly) in compliance mode.

Decrypting in Compliance Mode

```
// DECRYPT A STRING – WITH THE SYSTEM-GENERATED KEYSTRING PRODUCED BY ENCRYPTION
cr = cs.decrypt(cr.getCipherTextString(),
                cr.getKeyMaterial(),
                SingleJobDecryptionMode.SINGLE_JOB_KEY_WITH_COMPLIANCE);
```

Figure 17 - Decrypt a String SGK in FIPS 140-2 Compliance Mode

Decrypting a String that was encrypted in CLEAR Compliance mode requires decryption in compliance mode. Further, if CCJ JAR was used during the encryption process, it must be used in the decryption process as well. Ciphertext encrypted with CCJ may not be decrypted without CCJ, and vice versa.

String Encryption – System Generated Keys (SGK) – (Headerless Mode)

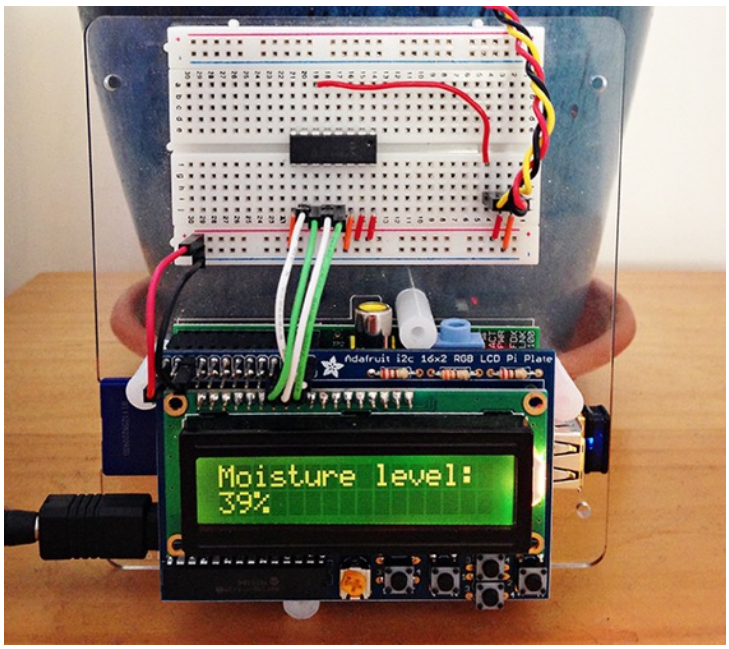


Figure 18 - An IOT Sensor with small-sized data packets

The CLEAR default mode of operation will produce a ciphertext that is only eight [8] bytes larger than the original data. When running in HMAC mode, the total additional encumbrance to ciphertext is eighty [80] bytes.

Adding 8 bytes of data to something might not seem like much. If you're encrypting a 3GB file, would you really notice an additional 8 bytes?

Well, if your data comes from a small IOT sensor and is only sending packets of 8 bytes; e.g.; "39%" (as pictured), then adding 8 more bytes could be a really impactful matter. This is why we have "**CLEAR Headerless Mode.**"

Encrypting data in CLEAR Headerless Mode adds exactly zero [0] additional bytes of overhead to ciphertext, making it ideal for use with sensors and IOT / IIOT devices. As a caveat, Headerless Mode encryptions cannot be combined with HMAC / AEAD signed encryptions.

Encrypting in Headerless Mode

Encrypting a String in Headerless mode is almost identical to encrypting in standard (basic) mode. To use Headerless mode during encryption, select "**SINGLE_JOB_SYSTEM_GEN_KEY_HEADERLESS**" for the mode selection.

```
// ENCRYPT A STRING – SYSTEM GENERATED KEY
CLEARResult cr = cs.encrypt(_CLEARTEXT_TEST_DATA,
                            CLEAR.STRENGTH_2048_BIT,
                            SingleJobEncryptionSysGenKeyMode.SINGLE_JOB_SYSTEM_GEN_KEY_HEADERLESS);
```

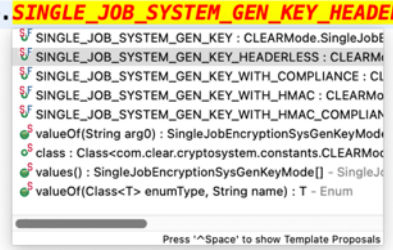


Figure 19 - Encrypt a String SGK in Headerless Mode

Encryption in Headerless mode is intended for environments (like IOT) where eight additional bytes of data encumbrance is problematic or material to a use-case. Even in large database environments where one might not necessarily view eight bytes as a material concern; strategies that use CLEAR to encrypt every VARCHAR in a database may elect to use Headerless mode when every character matters.

Important note: In cases (like IOT) where each bit of data is counted and padding is infeasible, the Quantum Knight team recommends using CLEAR-Streaming SDK (also in Headerless mode). CLEAR-String SDK will add space to ciphertext by way of UTF-16 / Unicode encoding of String data. The byte[] binary interface in CLEAR-Streaming SDK is more exacting.

Decrypting in Headerless Mode

```
// DECRYPT A STRING – WITH THE SYSTEM–GENERATED KEYSTRING PRODUCED BY ENCRYPTION
cr = cs.decrypt(cr.getCipherTextString(),
                cr.getKeyMaterial(),
                SingleJobDecryptionMode.SINGLE_JOB_KEY_HEADERLESS);
```

Figure 20- Decrypt a String SGK in Headerless Mode

Decrypting a String that was encrypted in CLEAR Headerless mode requires decryption in Headerless mode as well. Failure to match modalities in encryption and decryption will result in a failure to decrypt.

String Encryption – System Generated Keys (SGK) – (MFA Mode)

Perhaps one of the most powerful features of CLEAR is its ability to combine symmetric encryption keys with Multifactor Authentication (MFA) or biometric data. The CLEAR algorithm literally combines your own binary MFA input with cryptographic key material to create a key that can only be used for decryption when recombined with the original MFA. In one sense, this is a means for separating symmetric key material into a public / private key-pair.

Encrypting in MFA Mode

Encrypting a String in MFA mode is slightly different to encrypting in standard (basic) mode in that the method signature for input parameters changes. Specially, the SDK needs the ability to take MFA input data as a separate input parameter. MFA Mode can be combined with other CLEAR modes as illustrated a bit later in this document.

```
// ENCRYPT A STRING – SYSTEM GENERATED KEY + (MFA MODE)
CLEARResult cr = cs.encryptWithMFA(_CLEARTEXT_TEST_DATA,
                                   CLEAR.STRENGTH_2048_BIT,
                                   mfaMaterial,
                                   SingleJobEncryptionSysGenKeyMode.SINGLE_JOB_SYSTEM_GEN_KEY);
```

Figure 21 - Encrypt a String SGK in MFA Mode

In Figure-22 above, we are using the "overloaded" encrypt method that appends "**WithMFA**" to the method name (underlined in fuchsia). Further, we are adding the parameter "**mfaMaterial**", a binary byte[] to the method arguments after the Strength setting and prior to the Mode setting.

Important note: MFA Material can be any manner of binary input. Internally, the mfaMaterial needs to match the length of the encryption key material. If you submit data that is shorter than the key, CLEAR

will automatically multiply it and stretch to match. The minimum MFA data length is 1 byte of data and CLEAR does not modify or enhance MFA data. That being said, and a matter of best practices in cybersecurity, the Quantum Knight team recommends providing MFA data that is at least as long as the bit strength of the encryption operation. The maximum recommended length of MFA Material is 64MB. Although allowable by CLEAR as an input parameter, the use of very large MFA Material may be an unnecessary waste of system memory resources.

Decrypting in MFA Mode

```
// DECRYPT AGAIN + (MFA MODE)
cr = cs.decryptWithMFA(cr.getCipherTextString(),
                       cr.getKeyMaterial(),
                       mfaMaterial,
                       SingleJobDecryptionMode.SINGLE_JOB_KEY);
```

Figure 22 - Decrypt a String SGK in MFA Mode

Decrypting a String that was encrypted in CLEAR MFA mode requires decryption in MFA mode as well. Failure to match modalities in encryption and decryption will result in a failure to decrypt. As is the intention of MFA, failure to decrypt with the same mfaMaterial that was utilized during encryption will also result in encryption failure.

String Encryption – System Generated Keys (SGK) – (Mixed Modes)

CLEAR encryption modes may be used together in several different combinations. One example is combining HMAC mode with Compliance Mode.

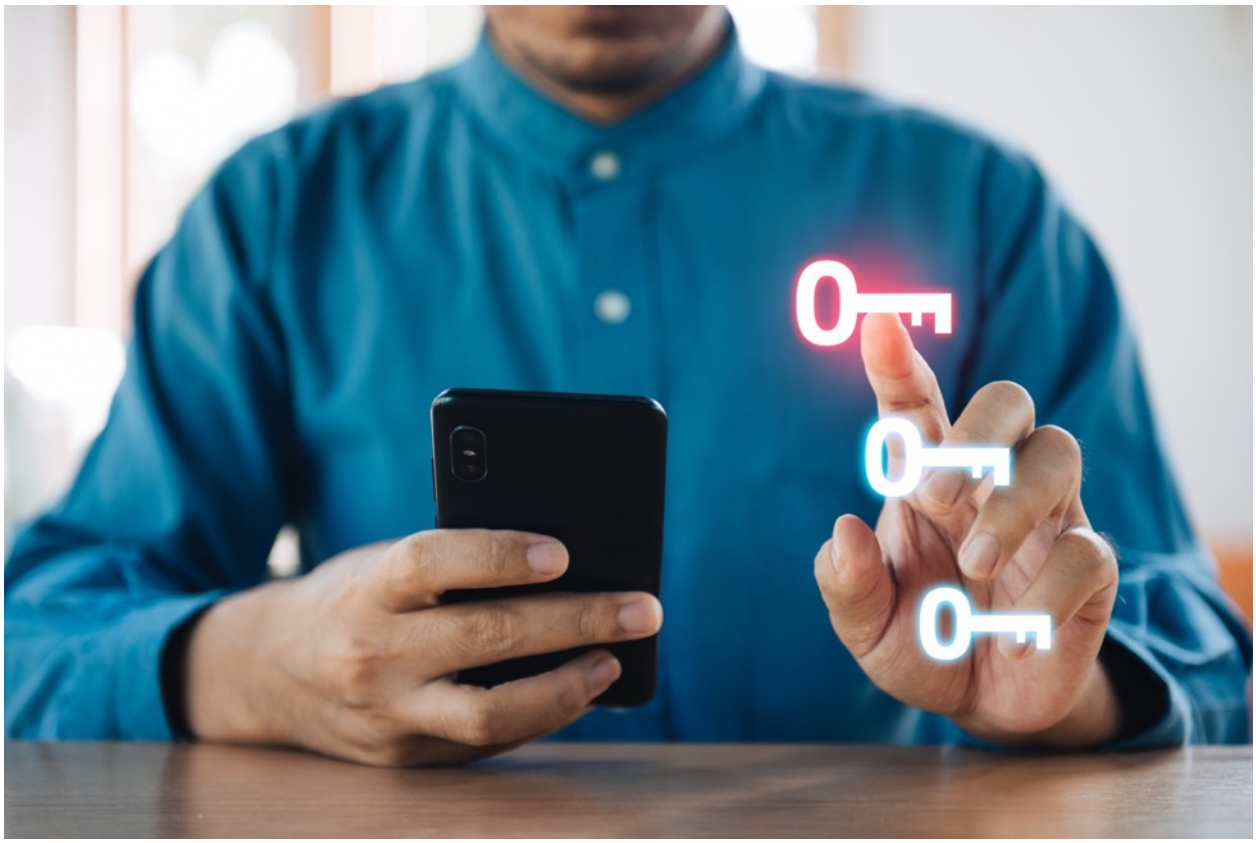
```
// ENCRYPT A STRING - SYSTEM GENERATED KEY
CLEARResult cr = cs.encrypt(_CLEARTEXT_TEST_DATA,
                            CLEAR.STRENGTH_2048_BIT,
                            SingleJobEncryptionSysGenKeyMode.SINGLE_JOB_SYSTEM_GEN_KEY_WITH_HMAC_COMPLIANCE);
```

Figure 23 - Encrypt a String SGK in combined (HMAC and Compliance) Modes

The following are valid combinations of modes for CLEAR String encryption with System Generated Key:

1. HMAC + Compliance
2. HMAC + Compliance + MFA
3. HMAC + MFA
4. Compliance + MFA
5. Headerless + MFA

Section 3 – User Generated Keys + CLEAR String SDK



User Generated Encryption Keys (UGK)

There are many real world use-cases whereby the generation of encryption keys via direct user-supplied entropy is required. A simpler definition for this would simply be the use of a "password" or "passphrase" to be incorporated into the generation of entropy. When a user generates a key with a password, CLEAR will produce a CLEAR Hyper-Key compliant format from the password.

Passwords are converted to any entropy format that is recognizable to CLEAR via a process of several transformative security steps as follows:

- Passwords are modified into a one-directional hash using SHA-512 hash algorithm. This step prevents passwords from being reversed back into their original state.

- Hashed password output is transformed into a form of integer-based entropy and "widened" via hamming-weight balancing and pseudo-random number transformation.

It is important to note that User Generated Keys ("UGK") will always produce the same output given the same input. That is to say that if we use the passphrase, "**my_password**"; the subsequent use of the same passphrase will produce an *identical key material*.

UGK will always produce encryption keys of 512bit strength. UGK cannot be used with CLEAR's higher strength settings. The team at Quantum Knight would like to convey that key-reuse across multiple encryption-runs (with a password) is not a best practice in cybersecurity. The CLEAR cryptosystem will not automatically prevent an operator from using the product in this way.

String Encryption – User Generated Keys (UGK) – (Basic Mode)

CLEAR String encryption produces ciphertext and key material as dual outputs in the System Generated Key (SGK) and User Generated Key (UGK) modes of operation. All SGK modes of operation mentioned in the previous section will automatically produce encryption keys based on the specified numeric strength. Alternatively, UGK modes of operation take a "**user password**" as an input parameter in lieu of a numeric strength designation.

```
// ENCRYPT A STRING – USER GENERATED KEY
CLEARResult cr = cs.encryptUGK(_CLEARTEXT_TEST_DATA,
                               "my_password",
                               SingleJobEncryptionUserGenKeyMode.SINGLE_JOB_USER_GEN_KEY);
```

Figure 24 - Encrypt a String UGK in Basic Mode

As shown in Figure-25, Encrypting a String in UGK requires "user password" in lieu of a Strength setting.

String Encryption – User Generated Keys (UGK) – (Combinations & Caveats)

Mechanically, UGK Encryption will take a user's password and blend it with pseudo-randomly generated numbers to create sufficient entropy to generate 512 bits of key space. Whilst the minimum amount of data required to generate a UGK is one [1] single character, the Quantum Knight team recommends providing MFA data that is at least as long as the bit strength of the encryption operation.

Further, CLEAR does not prevent large amounts of UGK data are allowable by CLEAR as an input parameter, however; the use of very large UGK Material may be an unnecessary waste of system memory resources.

What about Password Re-Use?

As a best practice in symmetric cryptography, the Quantum Knight team recommends against password re-use in encryption keys. As a form of "abstract" non-transactional key exchange, the CLEAR UGK interface could be used to produce CLEAR-compatible key material on two sides of an encryption sharing-schema if Alice and Bob were both aware of a password and create their own private keys via UGK modes.

What combinations are available for UGK?

In the CLEAR-String SDK, all the discrete combinations of modes from System Generated Key (SGK) interfaces, including HMAC mode, Compliance mode, and MFA mode, are all similarly available to the User Generated Key (UGK) SDK.

User Supplied Encryption Keys (USK)

Real-world symmetric cryptography often implies the need for a User Supplied Key (“USK”). Simply, this means that we can begin encryption (or decryption) with a CLEAR Hyper-Key compliant format that has been produced earlier, and at a different time than a specific encipherment operation.

Here’s an example:

As a CLEAR SDK developer, you are creating some code within a website or internet-facing API that is used to tokenize Strings whereby key-exchange or key-rotation is not feasible or unnecessary to the security of your use-case.

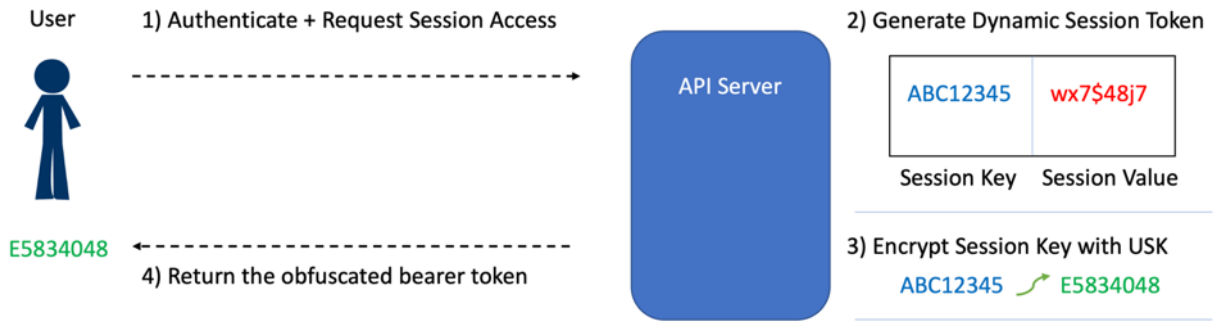
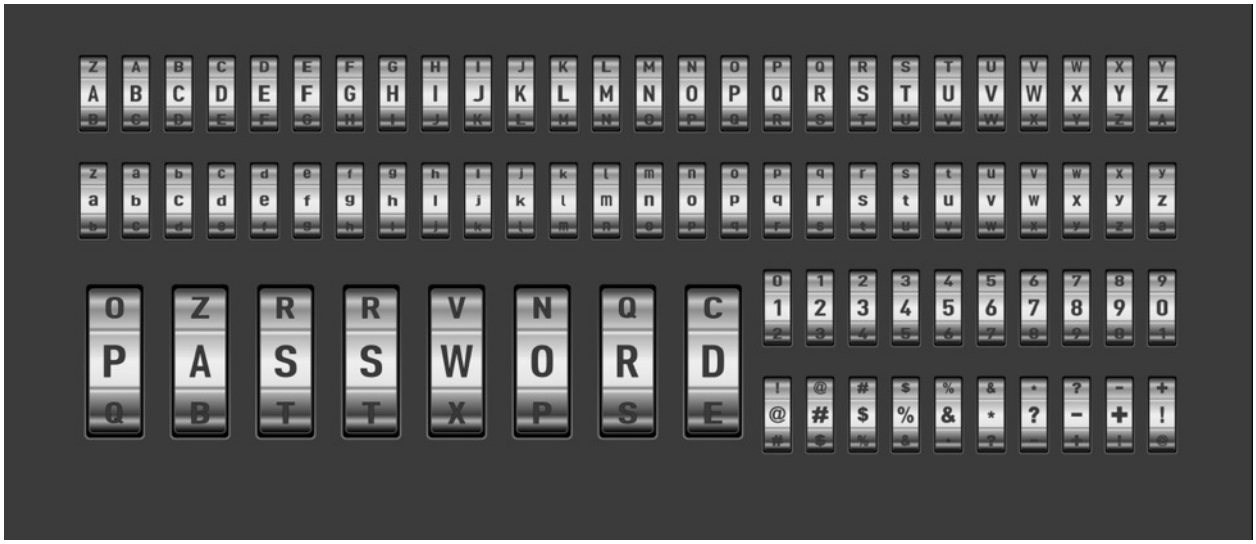


Figure 25 - Example of CLEAR String in USK Mode

As shown in the example above, there are scenarios whereby the use of a single encryption key may be used to “blind” dynamic data so that it can be shared externally without revealing the actual system state. The Quantum Knight team would like to convey that proper key management and handling is foundational to the security of systems and infrastructure.

The exhibit shown above in Figure-6 is only intended to illustrate a data transformation with USK in the simplest terms and is not being presented as a strategy or security recommendation. Tokenization of secure information in a well-managed environment is a best practice.

Key Tool



CLEAR symmetric encryption keys are generated in a proprietary format branded as Hyper-Key™. As described previously in this document, CLEAR keys can be generated as System Generated Key (SGK) or User Generated Key (UGK). In the case of SGK, the default onboard CSRNG can be used, or a plug-n-play random number generators (RNG) can be used to provide cryptographic key entropy.

There are use-cases whereby creating a CLEAR encryption key (without performing an encryption job) may be convenient, necessary, or specifically required to satisfy a solution. For this reason, CLEAR provides "Key Tool" as a convenient utility feature for generating stand-alone keys without performing an encryption operation. Keys can be stored, used later, or applied specifically to new encryption jobs in lieu of generating keys during the encryption process. Additional detail and suggested use of User Supplied Keys (USK) is provided in the USK sections below.

Creating a System Generated Key (SGK)?

```
// MAKE A SYSTEM GENERATED KEY (SGK)
CLEARKeyTool keyTool = CLEAR.clearKeyTool();
String usk = keyTool.genKey(strength);
```

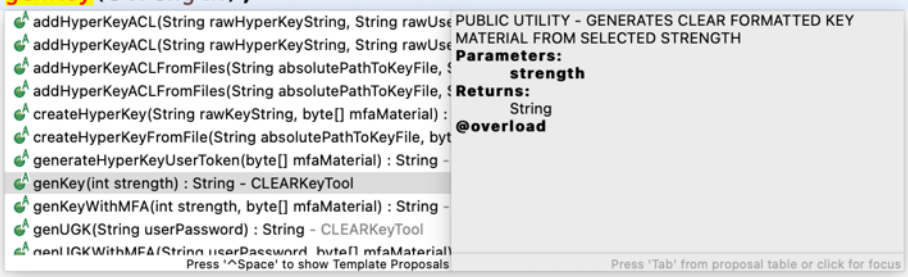


Figure 26 - Keytool - Generate a System Generated Key (SGK)

References

[1] The Legion of the Bouncy Castle –

[2] Grover's Algorithm –

[3] Random.ORG –

