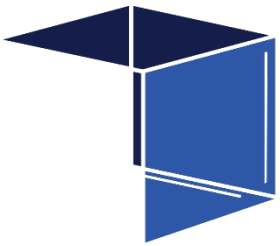
Quantum Dice – Thales Luna HSM QNRG-Backed HSM Integration Guide



QUANTUM
DICE

V 1.0.1
11-1-2024

INTRODUCTION

Random number generation is an essential task in any cryptographic stack and is a core element of the security of any cybersecurity system. It is used to create the various keys, certificates, nonces, tokens, and others that enable the various cryptographic functions required by the system. As such, the reliability and the security of the random number generation is a key aspect of cybersecurity.

The generation of randomness has been an open topic of development in computer science for a long time due to its complex nature both from a theoretical mathematical point of view, and, especially, from a practical implementation point of view. The modern approach is based on two separate systems:

- **Pseudo-Random Number Generators (PRNGs):**
  Also known as Deterministic Random Bit Generators (DRBGs), these are algorithms that are designed to produce a sequence of unpredictable digital random numbers, with known and desirable statistical properties, starting from an initial random input (typically called a random seed). A wide range of PRNGs have been developed over the past decades which rely on different mathematical tools such as congruential generators and hash functions.

- **True Random Number Generators (TRNGs):**
  These are generators which rely on some real-world physical process to produce entropy (a mathematical quantity used to quantify and measure randomness). Many different systems and devices can be used as TRNGs, but they fall into the two broad categories of TRNGs, ones which are based on classical hardware noise (henceforth referred to as Hardware Random Number Generators (HRNGs)) and ones which use quantum mechanical processes to produce entropy (henceforth referred to as Quantum Random Number Generators (QRNGs)). Depending on how a TRNG is constructed, it can have wildly different features and capabilities for everything from its speed of generation to the reliability and security of its output.

The complex nature of how to work with the different types of PRNGs and TRNGs, and how to evaluate their impact on the security of a system and how they affect its attack surface, means that great care needs to be taken when designing any cryptographic system. In general, most modern cybersecurity hardware relies on a combination of a TRNG which is used to seed a PRNG which in turn produces all the cryptographic primitives that are needed by the hardware.

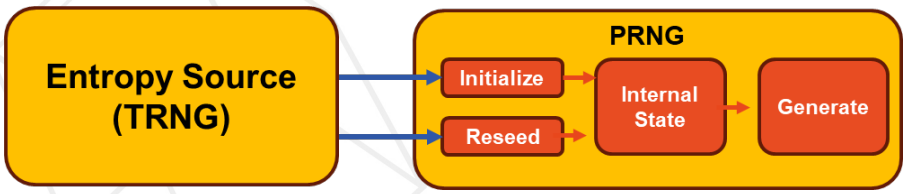


Figure 1 Simplified model of a TRNG used as an entropy source to seed a PRNG

As such, many standards organisations have produced and maintain standards for random number generation in cryptographic applications. The standards outline the minimal features

and capabilities required for random number generators that can be used in cybersecurity use cases. Some of the most well-known standards are the NIST (USA) SP800-90 family of special publications (SP800-90A. SP800-90B and SP800-90C) and the BSI (Germany) DRG.4.

The Thales Luna Hardware Security Module (HSM) 7 line has introduced improved Random Number Generation capabilities where it has an internal physical entropy source which is seeding an AES-256 CTR DRBG where the latter has a performance that is significantly improved from previous versions. Moreover, the Luna HSM 7 line supports the injection of externally generated entropy using the standard PKCS#11 API. This feature allows users to integrate entropy sources that would provide additional security features such as the Quantum Dice DISC™-enabled QRNGs.

Quantum Dice's QRNG solutions are all powered by their patented source-device independent self-certification (DISC™) protocol [1]. This allows them to leverage the unique properties of quantum photonics to continuously measure and certify the amount of quantum entropy that is being produced each sample.

This integration guide outlines how to integrate the two solutions to have a QRNG-backed HSM solution which can leverage both the market-leading performance of the Thales Luna HSM 7 and all its cryptographic capabilities, as well as the unique features allowed by Quantum Dice's DISC™-powered Quantum Entropy-as-a-Service (QEaaS).

REQUIREMENTS

To be able to follow this guide, you will need the following:

1. Access to a machine with Universal Client for Luna HSM installed and configured to access a Luna HSM or Luna Cloud HSM partition with crypto officer credentials
2. Access to the Quantum Dice QEaaS [contact qeaas@quantum-dice.com for QEaaS access] or a QD QRNG Hardware device
3. Quantum Dice QEaaS SDK (this guide has been tested up to SDK v1.0.1) or Hardware SDK
4. Access to PKCS#11 library (usually provided with the Luna HSM system)
5. Python PKCS#11 Wrapper (the documentation for this wrapper can be found here: https://python-pkcs11.readthedocs.io/en/latest/)
6. Quantum Dice Background Application

SYSTEM ARCHITECTURE

Whether you prefer to rely on physical hardware systems, or use the QD QEaaS, the integration with the Thales Luna HSM is straightforward and can be done with your cryptographic application using a few simple tools and with a couple of easy-to-implement steps.

Below, you will find couple of simple diagrams to explain how you can use QD's QEaaS or your on-premises QRNG hardware systems to seed your Thales Luna HSM with minimal setup or adjustments to your workflow.

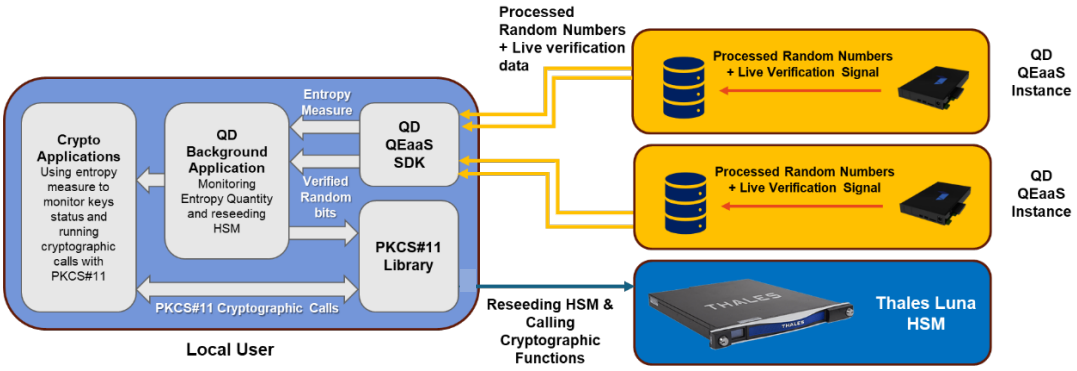


Figure 2: Diagram for integrating QD's Quantum Entropy-as-a-Service solution with the Luna HSM

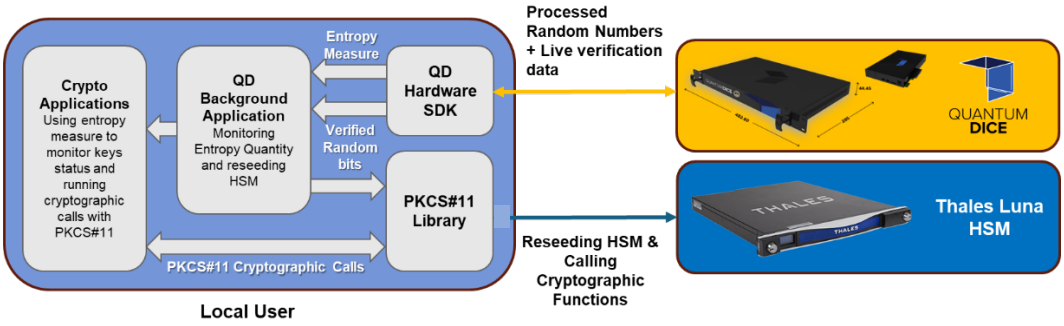


Figure 3: Diagram for integrating QD's Hardware QRNG solutions with the Luna HSM

The next section of the guide focuses on the QEaaS Python SDK but is easily transferable to the QEaaS C++ SDK or any of QD's hardware SDKs.

PYTHON SDK

Quantum Dice's Hardware or QEaaS Python SDK provides functionality to generate certified randomness as well as give access to the certification signal which gives the user access to the continuous measurement of the quantum entropy that is used to produce the randomness in the samples they are receiving [2]. This section of the guide focuses on a guide to implement your own reseeding mechanism using the Python SDK.

```python
import pkcs11
import os
import qdremrand

# Initialise our PKCS#11 library
lib = pkcs11.lib("PKCS11_LIB")
token = lib.get_token(token_label='HSM_IDENTIFIER')

data = b'INPUT DATA'

# Initialize connection QD QEaaS
qrn = qdremrand.QDRemRand("HOST_ADDRESS", "PORT_NUMBER", "SERVER_CERTIFICATE", "CLIENT_CERTIFICATE", "CLIENT_KEY")

# Open a session on our token
with token.open(rw=True, user_pin='USER_PIN') as session:
    # Generate 64 bytes of Certified Quantum Entropy
    entropy = qrn.randbytes(64)
    # Seed to HSM using PKCS11 API
    session.seed_random(entropy)
    # Generate an AES key in this session
    key = session.generate_key(pkcs11.KeyType.AES, 256)
    # Get an initialisation vector
    iv = session.generate_random(128)  # AES blocks are fixed at 128 bits
    # Encrypt our data
    crypttext = key.encrypt(data, mechanism_param=iv)
    print(f"{crypttext}")

```

Figure 4: Code Snippet of using the Python PKCS11 Wrapper and the QD QEaaS Python SDK to inject entropy into a Luna HSM

The PKCS#11 Library offers a number of functionalities to interface with HSM systems, it's generally used in C++ but there is also a simple Python wrapper for it. The main function of interest for our purposes is C_SEED_RANDOM which allows one to seed an HSM with an external source.

As can be seen in the code snipped above which shows an example of how to use the Quantum Dice QEaaS SDK to generate a random sequence, seed a Luna HSM using the PKCS#11 library and then use the rest of the library's cryptographic functionalities to run a basic encryption task. QD's SDK also allows you to extract the verification information provided thanks to the DISC protocol to allow you to check the quality and quantity of quantum entropy that is being produced before seeding your keys with it. This gives you an unprecedented level of assurance and monitoring on the performance of the seeding of your encryption keys.

For your convenience, you can also use the QD-developed background application which continuously reseeds the entropy of your HSM while provide an easily accessible audit trail of the performance of the QRNG and the quantity of quantum entropy used to seed the HSM. Your cryptographic application can then check with the QD background application to confirm that the desired quantum entropy has been correctly injected. You can then add additional logic to terminate the connection to the HSM and/or the application if the entropy seeding fails or the quantum entropy measurements coming from DISC are insufficient for your use case.

NOTES

[1] Please refer to Quantum Dice's DISC™ explainer document for more details about the protocol, how it works and its benefits

[2] Please refer to the QD SDK documentation for more details about the functionality allowed in both Python and C++