Quantum secured wallets – PHASE I – using QRNG to generate entropy

Summary

In this communication we describe the first phase of our work to build quantum sequred wallets for blockchains. This phase assumes the use of Quantum Random Number Generator for the generation of entropy for the wallets function of private key generation. In the work reported here we have used our QRNGBase service providing Entropy-as-a-Service via REST based, SLL protected API, and a popular open-source Hierarchical Deterministic Wallets. Have build a new code into the wallets' code that uses QRNG to generate entropy.

Basic info about wallets

A cryptocurrency wallet is a solution that stores public and private keys for blockchain based cryptocurrency transactions. A wallet can be implemented as an independent physical device, an application for a desktop/laptop computer or mobile device or even as a physical medium capable to store strings of data.

Over time the wallets functionality of the primary key storage, was expanded to cover functionalities of encrypting and signing information sent to the blockchain. The signing is critically important for execution of smart contracts on the blockchain.

Securing cryptocurrency wallets

There are various levels on which wallets can be protected. While in the course of the development of our product and services we will be proposing the ultimate protection of wallets capable of resisting the threats of quantum computers, in the beginning we address the security of the entropy (randomness) generation, critically important for the security of the private/public key pairs. Almost all existing software wallets us pseudo-random number generators (PRNG) to generate the keys. This is far from secure, for both the quality reasons (poor entropy) and for the possibility of backdoors in the generators. The case of Dual_EC_DRBG which allegedly had a trapdoor. This was a very serious problem. If one knows the trapdoor, he could predict the output of the random-number generator after collecting just 32 bytes of its output. In reality, one would only need to monitor one TLS internet encryption connection in order to crack the security of that protocol.

While such attacks on popular wallets are now known, the mere possibility of them should be avoided.

The solution we propose is to use QRNG to generate absolutely random and unpredictable sequence of random numbers. Until the QRNG devices are sufficiently miniaturized and widely available, they can be used in the Entropy-as-a-Service mode.

Protecting existing wallets with QRNGBase service

We have built code that calls our QRNGBase into two popular open-source wallets: PyWallet[1] oraz HDWallet[2]. In both cases the implementation was successful, but as PyWallet is much harder to setup and to work with in different environments, so we focused on the later one and report only that work.

Essentially we have added the following code:

```
def generate_qrng_entropy(strength: int = 128):
    def generate_byte_types(number_type, number_amount, number_length):
        # getting the api from quantumblockchains.io
        api_key = '47416dad-5ea0-463b-b2db-37edd4f77277' # constant
        api_provider = "qbck" # constant
        api_target = 'block' # constant
        api_url_prefix = 'https://qrng.qbck.io/'+api_key+'/'+api_provider+'/'+api_target+'/'
        # number_type // options: 'bigint', 'hex', 'bin', 'base64', 'base56'
        # number_amount // the amount of random numbers generated
       # number_length // the size of the random number in bytes (for bigint, hex and bin) or
       // characters (for base64 and base56)
        api_url = api_url_prefix+number_type+'?size='+str(number_amount)+'&length='+str(number_length)
        response = httpx.get(api_url, verify=False) # without verify=False property it does
                                                    # not work
        dictionary_response = json.loads(response.text) # by default the imported data is a
        string, so we need to covert it
        generated_number_list = dictionary_response['data']['result']
        # testing and printing results
        print('api_url: '+api_url)
        print('status_code: '+str(response.status_code))
        response_time = dictionary_response['data']['ExecuteTime']
        print('execution_time: '+response_time)
        return generated_number_list
    return generate_byte_types('hex', 1, strength//8)[0]
```

to the hdwallet source code, essentially modifying the standard function:

```
def generate_entropy(strength: int = 128) -> str:
    """
    Generate entropy hex string.

    :param strength: Entropy strength, default to 128.
    :type strength: int

    :returns: str -- Entropy hex string.


    if strength not in [128, 160, 192, 224, 256]:
        raise ValueError(
            "Strength should be one of the following "
            "[128, 160, 192, 224, 256], but it is not (%d)."
            % strength
        )
    return hexlify(os.urandom(strength // 8)).decode()
```

The results

The QRNG modified hdwallet worked correctly working in the same way as the unmodified version.

[1] https://pypi.org/project/pywallet/
[2] https://hdwallet.readthedocs.io/en/v2.1.1/

The generated cryptographic materials we properly generated. The test output is:

```
('cryptocurrency', 'Bitcoin')
('symbol', 'BTC')
('network', 'mainnet')
('strength', 256)
('entropy', 'f189b4827ec2000efa949c5bb57a6c69df0815810d17fce60dc2f77de1917cd0')
('mnemonic', 'various ethics calm work cactus alter tumble near forum profit only
squirrel vacuum approve aerobic pepper woman corn idle upon task silly viable
autumn')
('language', 'english')
('passphrase', None)
('seed',
'ca7c0882db60c3bb971778f632351d4da51ffaa4bb201357b37226d93e2a5831968feac3a417341428
d2be37e395c407acf17c93298acff446d043cdf771ee66')
('root_xprivate_key',
'xprv9s21ZrQH143K2SQccPu6NU4j6m1W5cEAVRtxhFyqkJd4riG95sDpQMPjZJUpthbJvHmwuJmStH5Dqd
vzs4DxMUA6JJwawzuuHPyPEzrDpyu')
('root_xpublic_key',
'xpub661MyMwAqRbcEvV5iRS6jc1TenqzV4x1repZVePTJeA3jWbHdQY4x9iDQZbMLJMihMV9sZWC5ydAwm
5tWcj9eU3Y7uCNmcUv2K9HVGPWBC7')
('xprivate_key',
'xprv9s21ZrQH143K2SQccPu6NU4j6m1W5cEAVRtxhFyqkJd4riG95sDpQMPjZJUpthbJvHmwuJmStH5Dqd
vzs4DxMUA6JJwawzuuHPyPEzrDpyu')
('xpublic_key',
'xpub661MyMwAqRbcEvV5iRS6jc1TenqzV4x1repZVePTJeA3jWbHdQY4x9iDQZbMLJMihMV9sZWC5ydAwm
5tWcj9eU3Y7uCNmcUv2K9HVGPWBC7')
('uncompressed',
'87173ba3cec7fa048a544d662087f8bb783516f669ac6f26d352bee5b42cdf152838f9c87f12d63033
7c1713b4e2777d8d1cfa58d32c7970bd14b3a76fc5d6a4')
('compressed',
'0287173ba3cec7fa048a544d662087f8bb783516f669ac6f26d352bee5b42cdf15')
('chain_code', '262fbc39b93faa28663c37369e8530311f8a5cbe23608dfb6f590a1b7871b4dc')
('private_key', '7c4926e7e04aba778b97fe65c4213ab8e6a5a125535ba7c67c12dca5f98efb18')
('public_key',
'0287173ba3cec7fa048a544d662087f8bb783516f669ac6f26d352bee5b42cdf15')
('wif', 'L1PJi6t3SK1sLPzjYQJUgugxhXUf2uqwYgpzGCbRUKfcmdf7qNaY')
('finger_print', '5249fb9a')
('semantic', 'p2pkh')
('path', None)
('hash', '5249fb9a6c973a1e6463f36f00034c3500c94c96')
('addresses', {'p2pkh': '18W739XAcVKuVgh4UTsCCofSJJKNgSugJa', 'p2sh':
'3GSJETAU8LWHGzJQV2Vjjti9BbHkxviT81', 'p2wpkh':
'bc1q2fylhxnvjuapuerr7dhsqq6vx5qvjnyk07vn7a', 'p2wpkh_in_p2sh':
'38msxqKSFhaLavc23jAZRNMYWqjf2T5mme', 'p2wsh':
'bc1q0mzgpjs6zefzzzy8g2thnp02f9f9ud0s5ll8fjfr0wg6ncs9m6hsvvgcgf', 'p2wsh_in_p2sh':
'3QfPwjHwXWZXRKzwwSDRu9dhRnYB2pbMxY'})
```

The conclusion

We have demonstrated the modification of hdwallet leading to the use of Quantum Random Number Generators for augmented entropy generation and unpredictability which ultimately makes the resulting wallet much safer.

The work reported here will be used as a first step of further and deeper modification of cryptocurrency wallets.