The Qwyit Protocol

Reference Guide

Version 1.6 March 2019

Copyright Notice

Copyright © 2019 Qwyit LLC. All Rights Reserved.

Abstract

This paper provides a technical overview of the Qwyit Protocol; formerly known as RPM (Real Privacy Management).

Contents

Introduction ... 4
Approach ... 4
Qwyit Low-level Functions ... 4
  MOD16 and MOD16D ... 4
  One Way Cut (OWC) ... 5
  Position Digit Algebra Function (PDAF) ... 5
  Combine and Extract ... 5
Qwyit Assumptions ... 5
Qwyit Protocol Process ... 5
  Initial Authentication Token/Credential Distribution (Verified Setup – VSU) ... 6
  Per Message Token/Credential Distribution (Authentication Handshake – AH) ... 7
  Messaging - QwyitTalk ... 9
  Qwyit Stream Cipher ... 10
QwyitTalk and QDS Operation ... 11
Security and Claims ... 12
  Qwyit Design ... 12
  Qwyit Security Basis ... 12
  Reference Case Analysis ... 14
  Security Summary ... 14
  Expert Analysis ... 14
Qwyit Functions and Protocol – Reference Code and Test Vectors ... 15
  The Qwyit Reference Primitive Functions in VB6 ... 15
  ' FUNCTION: MOD16 Encrypt Function - String Return ... 16
  ' FUNCTION: MOD16 Decrypt Function - String Return ... 18
  ' FUNCTION: One-Way Cut Function - String Return ... 20
  ' FUNCTION: Position Digit Algebra Function (PDAF) ... 22
  ' FUNCTION: Byte to 4-bit to Byte Translation Function ... 27
  ' FUNCTION: Qwyit Combine Function ... 29
  ' FUNCTION: Qwyit Extract Function ... 31
  ' FUNCTION: Qwyit Stream Cipher - Data Security Function - (both Encrypt/Decrypt) ... 33
Appendix – Qwyit Functions in Pseudo-Code ... 36
  ' FUNCTION: MOD16 Encrypt Function - String Return ... 36
  ' FUNCTION: MOD16 Decrypt Function - String Return ... 37
  ' FUNCTION: One-Way Cut Function - String Return ... 38
  ' FUNCTION: Position Digit Algebra Function (PDAF) ... 39

FUNCTION: Combine *42*
FUNCTION: Extract *43*

The Qwyit Protocol

This document is the Qwyit™ [**kwahy**-it] Reference Guide for the Qwyit Protocol, which provides authentication (embedded) and data security (stream cipher). "Qwyit" refers to all parts of this version of the protocol, including the low-level functions. The complete protocol includes authentication key management through the Qwyit Directory Server system. Reference implementations with test vectors are available in several platforms. To obtain specific Qwyit APIs in any particular programming language, please contact Qwyit LLC, or go to www.qwyit.com. Qwyit was formerly known as RPM (Real Privacy Management).

Introduction

Qwyit provides authentication and data security at either the session level, or individual transaction level of any communications network. The Qwyit method uses a symmetric key exchange for authentication and data security. It is a one-pass embedded authentication method, based on underdetermined equation sets. Qwyit provides fast performance, small size, flexible trust, simple execution and improved security.

Approach

Qwyit provides low-level foundation functions based on two simple principles:

- Modular arithmetic
- Digit position manipulations

These two simple principles, applied in unique configurations, provide the necessary protection for exchanging numeric key information and encrypting communications. They also provide the defining features that differentiate Qwyit from the other systems: speed and size. Qwyit performs step transformations of numeric key values into underdetermined numeric key results by pointing into values using other values and extracting the results which are then combined in simple modular fashion to generate transmittable values. The recipient then uses these transmission values to perform the same pointing, extraction and combinations to arrive at the implied transfer values. The resulting values are both the authentication credentials and the data security keys used for encrypting message data. Qwyit then provides a simple stream cipher with key chaining for data security (encryption and decryption).

By building a system flow of the Qwyit low-level functions, one can quickly and simply insert security – authentication and data encryption – into any communications system. The following is the reference Qwyit key management architecture and stream cipher.

Qwyit Low-level Functions

(Please contact Qwyit for the definitive function calls and library configurations)

MOD16 and MOD16D

This pair of functions takes two input values and performs a modular add without carry of each digit position within the hexadecimal values.

One Way Cut (OWC)
This function uses modular add without carry to combine the adjacent digits (or other digit pairing combinations) to provide a one-way gate for irreversibly obtaining a new value from an old one.

Position Digit Algebra Function (PDAF)
This is a wide ranging, flexible function for taking one or two values and uniquely pointing a value into itself or one value into the other to obtain both position and value sub-results that are then modularly combined. The function is specifically capable of 'moving through' the inputs in a unique position/value manner such that a random unending result stream can be obtained (as measured by entity checking means). Using the PDAF provides a definitive, non-repudiated, one-way gate that can be easily configured within a function combination to provide wide-ranging underdetermined security with fast execution.

Combine and Extract
Using the same basic PDAF construct, this function pair creates a new key value 'alphabet' by selecting digit-position defined values and modularly adding them, and extracts a new key from that alphabet by selecting digit-position defined values from the add without carry result.

Combining these low-level functions in a variety of ways allows an almost infinite number of secure communications systems to be built, dependent on requirements and assumptions.

Qwyit Assumptions

The following are required and/or recommended for a Qwyit implementation:

1. Every participant must have the ability to operate the minimal Qwyit functionality. Each participant must have Qwyit resident on their device, be able to operate a served version of Qwyit or operate a received version of Qwyit within an application.
2. Ability to store the Qwyit credentials (keys). A system-wide risk-to-value assessment should be performed to determine the desired storage technique(s). For example, the credentials can be PIN or password hashed, kept locally or on a token (USB, RFID, etc.), etc.
3. A Qwyit Directory Server (QDS) trust model (singular, multiple) with either a centralized or de-centralized (or combination) Key Distribution Center (KDC – the key management portion of a QDS). Participants must have a Security Association with a QDS through a Verified Setup (VSU) and for every Qwyit messaging session (system defined – transaction level, session level, system level, etc.), they will perform an Authentication Handshake (AH), be granted shared message keys and perform messaging exchange using the Qwyit Cipher (Qwyit*Talk*[SM]) [See *QwyitTalk and QDS Operation* section]
4. Recommended: a best-of-breed random number generator (NIST approved); if not available locally, risk/reward entropy analysis should be performed and appropriate provision provided

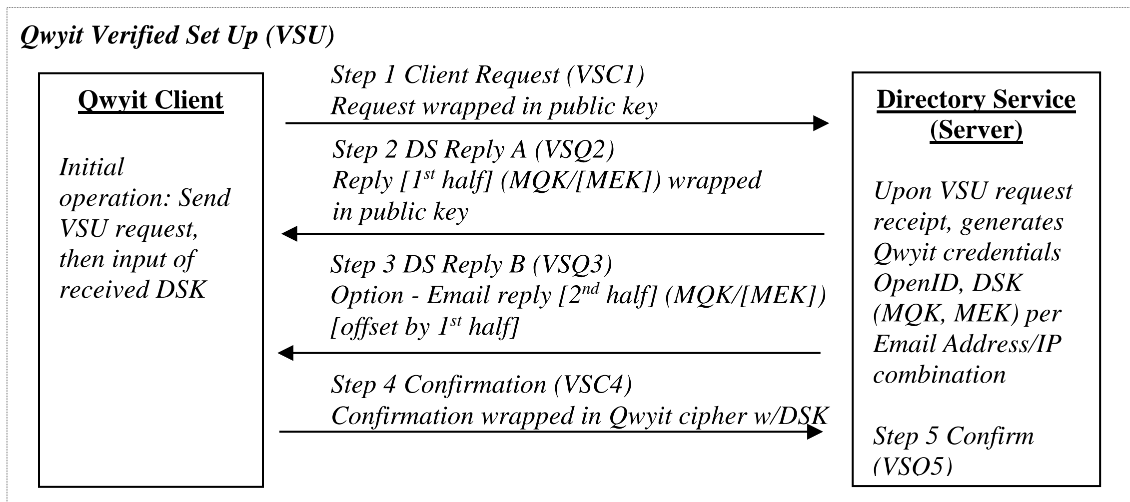
Qwyit Protocol Process

The following are the Qwyit Protocol processes:

Initial Authentication Token/Credential Distribution (Verified Setup – VSU)

- Initial Qwyit authentication token distribution is accomplished through a Verified Setup (VSU) with a Qwyit Directory Server (DS). Token is a 512-bit 2-part key, with a public identifier: OpenID [up to 64-bits], MQK [Master Qwyit Key, 256-bits], MEK [Master Exchange Key, 256-bits])

Architecture



NOTE: This is the *minimum recommended* electronic key distribution method: using 2 independent communication bands. Depending on the security requirements of the system, one may use more or less, although not recommended – TLS/SSL only uses a single band wrapped in a public key – Qwyit recommends at least two bands. One can imagine *QwyitPlus* (*Q+*) using 3 bands (adding SMS texting), or *QwyitMax* (*QMax*) using 4 bands (adding a phone call), or even *QwyitPlatinum* (*QP*) using 5 bands (adding a paper delivery (or two!)) all sending either MQK/MEK portions or simple PDAF or OWC offsets. Qwyit finds no value in a security discussion placing emphasis on the initial key distribution method – all arguments end at manual entry of whispered values; *we have not found a single communications system anywhere in the world with more than 2 participants that does not require some kind of initial authorization to join – not one – and key (identifying token of some kind) distribution upon joining is a requirement of every system – a simple risk/reward distribution analysis, then a chosen method, is de rigueur. Qwyit recommends a minimum of 2 bands, as shown above.*

Details

NOTE: Qwyit may use the TCP/UDP IANA reserved port for HTTPX, port number 4180 in a web architecture

Verified Setup (VSU)

- Client, initiate VSU on startup (and as required/desired) to Directory Service/Server (*Step 1, VSC1*)
  - Create Public/Private key pair as per shortest, fastest method (recommend 512-bit ECC)
    - Generate per VSU – do not store/re-use
  - Send VSU Request (VSC1) to DS
    - Send DS Flag code, optional public key, optional email address to DS
    - DS Flag code is a setting in Client Config – Set initially at Start Up to High (#0), the default.
      - DS Flag (DSF) codes where:
        - 1 (High) = key halves sent through email and through DS
          - Public key, email address required
        - 0 (Low) = DS Only – entire key sent through DS (no email) [*Not recommended]
          - Public Key required
      - Email address is a public POP address

NOTE: Based on above note re:security levels/bands, the DSF list may need extension [3 (Q+), 4 (QMax), 5 (QP), etc.]

- Directory Service/Server reply to client (*Step 2, VSQ2*)
  - The client has submitted a VSC1 using a public key for the reply
    - Generate OpenID (recommend 16 digit, 64-bit unique IDs), DSK for client
      - DSK is 512-bits, and includes two halves
        - Master Qwyit Key (MQK, first 64 hex digits, 256-bits)
        - Master Exchange Key (MEK, second 64 hex digits, 256-bits)
      - If DSF = 1
        - Reply (VSQ2) with OpenID, 1st half (MQK) encrypted by public key
        - Perform Step 3 to email address, sending OpenID and the 2nd half (MEK) offset (MOD16 encrypted) by 1st half (MQK)
      - If DSF = 0
        - Reply (VSQ2) with OpenID, whole DSK encrypted by public key

- Directory Service/Server reply to client (*Step 3, VSQ3*)
  - If DSF =0, this step is not performed
  - The client has submitted a VSC1 using an email address for the reply
    - Generate OpenID, DSK for client (IF not already done in Step 2)
      - If DSF = 1
        - Reply in email with 2nd half (64-digits, 256-bits) MOD16 encrypted with 1st half (MQK) (this will be OpenID, MEK *encrypted* to email address

- Client decrypt of reply and confirmation (*Step 4, VSC4*)
  - If DSF = 1
    - Perform decrypt the VSQ2 using private key to reveal 1st half (MQK) of DSK
    - Open email message to reveal 2nd half (MEK *encrypted*) of DSK
    - Open application for key entry
      - Enter both halves, and OpenID, into applet fields (form for entry of OpenID, MQK, MEK, full DSK – when show form, only those applicable to the DSF method (either MQK, MEK or full DSK active)
      - Click button for "Plug In Key" (or some relevant, pertinent UI text)
        - Applet will take 2nd half (MEK *encrypted*) and perform a MOD16D using MQK to reveal correct 2nd half (correct MEK)
        - Concatenate 1st half and 2nd half of DSK making it whole
        - Insert into use (store DSK, OpenID in cookie, file, db – method?
  - If DSF = 0
    - Perform decrypt using private key to reveal whole DSK
    - Applet will automatically insert OpenID and decrypted DSK into use (store DSK, OpenID – method?)
  - Reply (VSC4) to DS with Confirmation message
    - Perform Qwyit cipher using DSK (MQK and MEK), generating message key (W)
    - Use message key to encrypt confirmation message
      - Message format: "[OpenID] VSU ready!" (no brackets, just OpenID value)
    - Send (VSC4) output (OpenID, OR, ciphertext) to DS

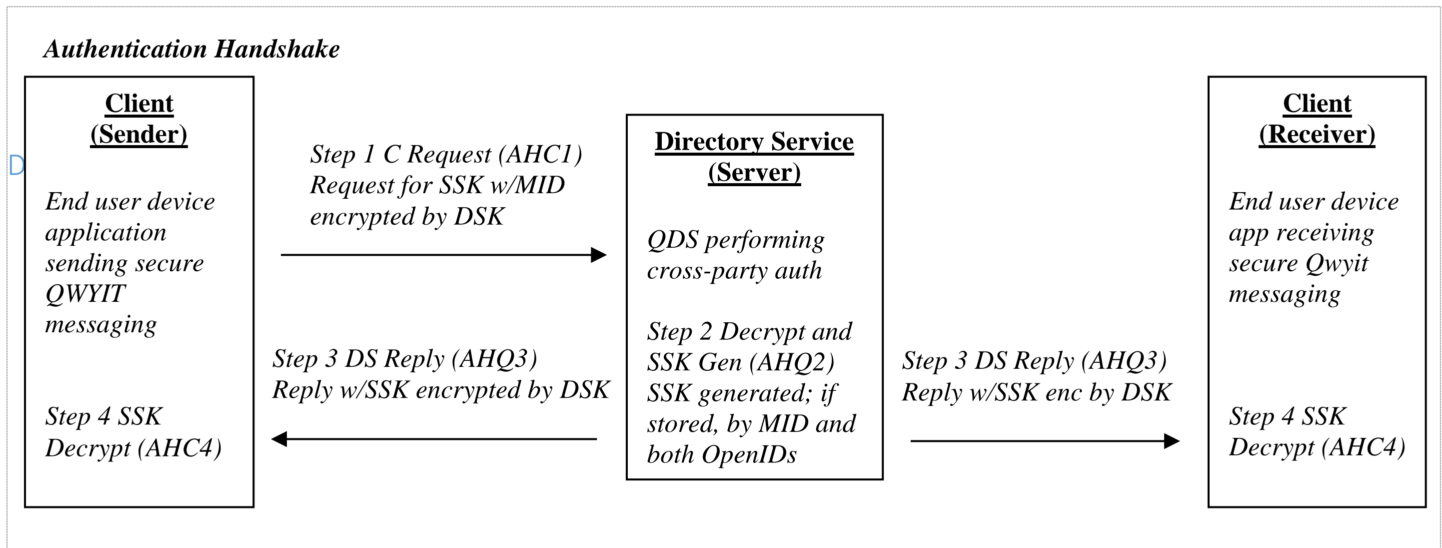
- DS decrypt of confirmation message (S*tep 5, VSQ5*)
  - For all DSF values (0,1)
    - Perform Qwyit cipher using the DSK (found by sent OpenID) and reveal the message key (W)
    - Use message to decrypt confirmation
    - If OpenID in message matches OpenID in header, confirmation
      - If not, error sent
      - If confirmed, store IP Address, OpenID, DSK, email address into DS KDC – method?

Per Message Token/Credential Distribution (Authentication Handshake – AH)

- For each participant-pair messaging session, the sender will initiate a private real-time Authentication Handshake (AH) with a shared Qwyit Directory Server in order to create a Qwyit VPN tunnel with the intended receiver (based on their OpenID). Token is a session-based 512-

bit Session Start Key (SSK). Participant-pair will generate their own Session Master Key (SMK) from the SSK in 2-parts: SQK [Session Qwyit Key, 64 hex digits, 256-bits], SEK [Session Exchange Key, 64 hex digits, 256-bits])

Architecture



Details

NOTE: The AH SSK is unique to each messaging pair of participants; after delivery, it is destroyed. Should there be a requirement to retain SSKs, the DS will need to be set to do so (by MID). Since the SSK is *not* the actual SMK used to encrypt the session/message, if the requirement arises to understand private conversations, after requiring the DS to store the SSK, the entire message stream could be captured and read (requirement to outside agency).

Authentication Handshake (AH)

- Client Sender sends Authorization Request to Directory Service/Server (*Step 1, AHC1*)
  - Create MessageID (MID, 64 hex digits, 256-bits) using (P)RNG
  - Send Authorization Request (AUTR) to DS in order to receive an SSK for communication w/intended Client
    - Send AHC1:SenderOpenID, ReceiverOpenID, MID encrypted in Qwyit Cipher using DSK to DS

- Directory Service/Server generates SSK (*Step 2, AHQ2*)
  - Decrypt AHC1 from Client using respective OpenID and DSK in Qwyit Cipher
    - Check that Client who is requesting is the same as encrypted request
  - Create Session Start Key (SSK, 128 hex digits, 512-bits)
  - If to be stored, by MID and SenderOpenID, ReceiverOpenID

- Directory Service/Server reply to Clients (*Step 3, AHQ3*)
  - Send SSK Accept (AHQ3) to Sender and Receiver Clients
    - Send SA:SenderOpenID, ReceiverOpenID, SSK encrypted in Qwyit Cipher using DSK to Clients
    - The DSSA is sent to both the sending and receiving Clients, and DS is authentic if using the correct DSK

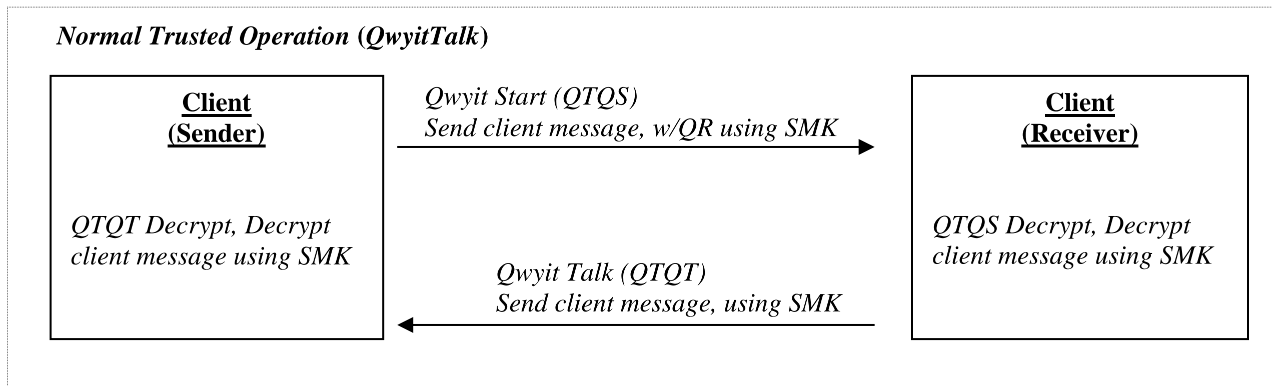
- Clients decrypt SSK (*Step 4, AHC4*)
  - Both Sender and Receiver Clients decrypt AHQ3 from DS using DSK in Qwyit Cipher
    - Receiver Client may not 'answer'; Sender will initiate contact with Qwyit Start message in Normal Operation

Messaging - QwyitTalk

- Normal Trusted Operation, each participant using the SMK in secure Qwyit Cipher exchanges – this is Qwyit*Talk*SM

(The OpenID does not require any specific storage requirements; SMK (SQK and SEK) must be handled and stored securely, although only for this particular session.)

Architecture



Details

Normal Trusted Operation

- Sending Client sends Qwyit Start to Client Receiver (*Qwyit Start, QTQS*)
  - Sending Client generates a Qwyit Return (QR, 128 hex digits, 512-bits)
  - Using the AH SSK from the DS for this communication:
    - Perform MOD16(SSK,QR); result is the Session Master Key (SMK, in 2 halves, SQK and SEK)
    - Create new message content (IMSG, Initial Message, opening communication)
    - Perform Normal Trusted Operation using Qwyit Cipher on IMSG with SMK (SQK and SEK)
  - Send Qwyit Start (QTQS) message
    - Send QTQS:OpenID, QR, OR, Ciphertext of IMSG

- Receiving Client decrypts QTQS (*Qwyit Talk, QTQT*)
  - Receiver Client, using the AH SSK received from the DS for this communication
    - Determines SMK from the QS message by performing the same MOD16(SSK,QR)
  - Commences Normal Trusted Operation message Receive decryption using SMK in Qwyit Cipher on OR, Ciphertext

- Clients perform Normal Trusted Operation messaging w/Qywit Cipher using SMK (*QwyitTalk, QTQT Encrypt/Decrypt*)
  - Send Qwyit Talk (QTQT) w/SMK (SQK, SEK) in Qwyit Cipher on message contents in secure communications
    - Send QTQT:OpenID, OR, Ciphertext of message contents
  - QTQT Decrypt w/SMK (SQK, SEK) in Qwyit Cipher to reveal secure message contents
    - Receive and decrypt QTQT in Qwyit Cipher using OR, to reveal Plaintext message contents

Qwyit Stream Cipher and Key Exchange/Update

Encrypt

Initial System Setup
Credentials issued *QK and EK*, along with an ***OpenID***

Initial Message Setup
Generate a new random salt (Open Return ***OR***)

Primitives

There are three ***Qwyit*** fundamental operations:

Combine – this is a select of the position digits and mod add the two inputs to create a number alphabet

Extract – this is a simple select out of the alphabet

Add w/o carry (MOD16) – this is single-position modulus arithmetic of two input numbers

Key Generation

① Perform a MOD16 or MOD16D w/ ***O R*** and *EK* resulting in seed Result
R = MOD16D[*EK*, ***OR***]

② Perform a Combine using *QK* and *R* resulting in alphabet *A*
A = Combine[*QK*,*R*]

③ Perform an Extract of *W* from alphabet *A*
W = Extract[*QK*,*A*]

⑤ **Next Block**
Use *W* in MOD16 w/ *R* resulting in *NextR*
R = MOD16(*W*,*R*)

④ **Block Encrypt**
Use *W* in MOD16 w/PT, ***CT*** = MOD16(*PT*, *W*)

Or Decrypt
Use *W* in MOD16D w/CT, ***PT*** = MOD16D(*CT*, *W*)

⑥ **Send Message**
Send ***OR*** and ***CT*** to the intended recipient using ***OpenID***

Key Exchange
Secure periodic redistribution of *QK*, *EK*

Decrypt

Send

- Generate random salt Open Return [OR 256-bits])
  (This step does not require an 'unbreakable' or 'secure' (P)RNG process since the value is public; but it should meet the base requirements for uniqueness and randomness since it is the QwyitTalk™ basis)

1. Perform MOD16 using EK and OR, resulting in R [256-bits]
2. Perform COMBINE using QK and R, resulting in alphabet A [256-bits]
3. Perform EXRACT using QK and A, resulting in message key W [256-bits]
4. Perform MOD16 with W and up to 256-bits of the Plaintext (PT), resulting in Ciphertext (CT)
   a. For performance, with key and PT identical lengths, this may be an XOR of W and PT resulting in CT
5. Perform MOD16 using W and R, resulting in the NextR [256-bits]
   - Perform 1 (using NextR as OR), 2, 3, 4 and 5 iteratively until the end of Plaintext (256-bits at a time), concatenating total C (There is no need to pad the plaintext)
     - Optionally (continual/random/regular), update EK (and/or QK) in MOD16's with last R or W
       Any 'update series' (with flag values, error checking and confirmations) is system-specific.
6. Send OpenID, OR and C to Recipient

Receive

1. Perform MOD16 using EK and OR, resulting in R [256-bits]
2. Perform COMBINE using QK and R, resulting in alphabet A [256-bits]
3. Perform EXRACT using QK and A, resulting in message key W [256-bits]
4. Perform MOD16D using 256-bits of Ciphertext (C) and W, resulting in Plaintext (P) [or CT XOR W = PT]
5. Perform MOD16 using W and R, resulting in the NextR [256-bits]
    - Perform 1 (using NextR as OR), 2, 3, 4 and 5 iteratively until the end of Ciphertext (256-bits at a time), concatenating total P
    - Optionally (continual/random/regular), update EK (and/or QK) in MOD16's with last R or W

QwyitTalk and QDS Operation

Qwyit works in a Federated Trust model, in both a Public and/or Private trusted network system(s). Qwyit LLC will operate a *Common Qwyit Directory Service* (*CQDS*) havin two distinct functions:

- Operation as a single, global individual QDS and associated KDC for any network access (e.g., A publicly accessible private QDS)
- Operation as the repository of trusted Private global interconnectivity, showing public record of any and all Private QDS's desiring interconnected, federated trust (.e.g., a Public KDC of Private QDS's that have performed VSU's and desire global trusted interconnectivity)

In establishing a Qwyit secure network, any operator can perform any or all of the following:

1. Embed Qwyit*Talk* in their products and operate their own Private QDS
2. Embed Qwyit*Talk* and operate their own Private QDS that is VSU connected to the CQDS
3. Embed Qwyit*Talk* and connect to the CQDS
4. Build Qwyit*Talk* and connect to another operator's QDS (Private and/or Public)

Scenario 1 establishes a private Qwyit network that cannot connect to 'the outside world', but very well may be interconnected with other operator-trusted networks (corporate, etc.).

Scenario's 2 and 3 will both have global trusted Qwyit connectivity; the advantage of Scenario 2 being that their Private QDS may be connected to additional/inter- and/or intra-networks that the operator desires to *not* be globally connected (enterprise, mobile, etc.) – the operator can easily manage this trust through their QDS whereas in Scenario 3, Qwyit LLC would not perform such 'housekeeping' (Trust in the Qwyit LLC CQDS is across all members).

Scenario 4 might be preferred for industry-specific QDS operation (utilities, etc.) where such QDS's can be both Private (customer relation to their electric company) and/or Industry-Public (specific geographic region of inter-connected utilities such as the Northwest, etc.) where the operators collectively run their PubQDS (a Public QDS operating for their domain and member's Private QDS as the CQDS does globally).

The Qwyit protocol processes will enable any and all Federated Trust connectivity of CQDS/QDS.

Security and Claims

There are several layers of study for a protocol as flexible as Qwyit, from cryptographic through business process application, where each has their own terminology, characteristics, etc. Qwyit LLC will leave it to the analyst to investigate any particular assumption(s) and property(ies) set. The general slogan for Qwyit is that it performs mutual, continuous authentication and data security. This defining statement means:

1. Mutual: both participants share the same secret, and exchange information mathematically based on it in both directions
2. Continuous: every Qwyit configuration presents unique information in every exchange
3. Authentication: Qwyit exchanges establish both parties as genuine
4. Data Security: Qwyit provides irreversible, one-way function protection of upper level credentials; and creates unique keys used by either a best-of-breed cipher (such as AES) or the Qwyit primatives-based stream cipher, on every communication.

Qwyit Design

Qwyit is the result of research into new methods for exchanging information securely. As there is a great deal of both theoretical and practical knowledge that purports to 'solve' most of the cryptographic needs of secure data exchange, Qwyit LLC has concentrated on making real-world, real-time enhancements. Qwyit has several features that result in distinction in the marketplace that are a direct result of its purposeful design; Qwyit provides authentication and data security based on the required real-time characteristics:

- Limited system components
- User-friendly participation
- Simple installation and operation
- Fast communications performance
- Long lifecycle
- High ROI
- Flexible application and functionality
- Small code space
- High Security (protection versus risk)

Qwyit meets these criteria, and does so by configuring uniquely designed cryptographic and engineering primitives.

Qwyit Security Basis

Qwyit creates consistent (having at least one solution), underdetermined systems of equations, and maintains them throughout the continued use of the algorithm. Underdetermined systems are unsolvable (secure); the type created by Qwyit is such that there is

- a large space of incorrect answers
- a *possible* small set of working answers that will produce an outcome that is correct only once (configuration and key size dependent – not all configurations and key values will produce a set) and
- a single correct answer

With recommended symmetric key sizes of greater than at least 256-bits, Qwyit provides strong protection against brute-force attacks. Analysis of any particular Qwyit implementation should be performed against the expected attack scenarios for that particular implementation.

Initial Key Distribution

The Qwyit Directory Service configuration of the reference Qwyit protocol provided in this paper relies on secure initial distribution of authentic tokens, and then the protocol and stream cipher maintain that authenticity and deliver strong data security to the communications through direct mathematic association. Should any other initial distribution mechanisms or processes be deemed necessary to provide 'better' starting security, then simple implementation of that requirement should be done – hand deliver, provide embedded in hardware, etc. There are many well-known techniques with established risk-reward profiles; Qwyit LLC recommends using them where necessary to compliment or replace the reference hard-math distribution outlined in this guide and as noted above.

Authenticaton

Qwyit authentication is provided through the mathematical generation (hence connection) between the original authentic token and any subsequent child encryption or update key. The impossibility of arriving at a child message key through any means other than direct underdetermined use of the original authentication token is the basis of authentic use; e.g., every 'inspection' (child key use) of ciphertext/plaintext encryption provides implied presentation of the original token. Unlike current authentication that relies on 'cryptographically secure' random number generation and subsequent 'encapsulating' of that child key with an authentic original token, Qwyit uses a *direct mathematic derivation* of the subsequent keys from the authentic original token; which is provided infrequently through cryptographically secure (P)RNGs where the ability to centrally secure their creation and use is exponentially simpler than distributed need/use of random child keys. Use of the original token is required and its security is maintained through one-way derivation and 'implied' arrival (it is never actually sent or exposed throughout any communications).

Data Security

Qwyit data security is reliant on the underlying cipher's properties for data integrity and security. Since Qwyit can pass its uniquely derived message key to any cipher, if any particular cipher is shown to be weak in any of the standard cryptanalytic attacks (or any other claimed property), it should be abandoned. The reference case Qwyit configuration using the Qwyit stream cipher does not provide data integrity, since this property isn't necessary in a real-world 'fully secure' network (e.g., it isn't a property of open systems, leaving only nuisance attacks (non-meaningful to the content), and it is the same with fully closed systems. The need vs. processing time, from a network perspective, is unwarranted – but hashing integrity can certainly be added if (needlessly) desired).

The Qwyit stream cipher provides strong data security based on Shannon Security, with child message key creation having the properties of OTP keys. Uniquely, the Qwyit stream cipher uses known random numbers as its seed, and will maintain the known keys' random properties throughout long-term use in chaining unknown derivatives; e.g., there is no need for seeds to be "cryptographically secure", only that the original authentication token base remains secure and unknown.

Reference Case Analysis

Qwyit provides a software reference implementation at the end of this document, as well as SDKs in multiple platforms, for analysis. The following are the simple assumptions for a secure Qwyit exchange:

- Initial Qwyit credentials securely distributed – the methodology for DS distribution presented in this document can be altered to meet any security threats – for analysis, initial distribution is assumed to be secure
- Secure switch (DS) and local storage of Qwyit credentials (e.g., nothing in any mechanism for the secure handling of Qwyit credentials is used to weaken an exchange)

The reference case is for investigation as to the validity of the underdetermined equation sets created by Qwyit exchanges – e.g., is the Qwyit protocol secure? Contact Qwyit LLC for help with analysis in any particular areas of study.

Security Summary

Qwyit is a one-pass secure message envelope protocol with a reference configuration that provides continuous mutual authentication and secure data communications. Its strength is based on underdetermined equation sets, and is designed for use in conjunction with other security processes and policies that protect the system components such that Qwyit's protocol assumptions remain valid. Within the context of its valid assumptions, Qwyit will deliver the properties stated in this paper.

Expert Analysis

Qwyit mathematics and security claims have been independently verified through many man-months of study since 2011 by Dr. Giovanni Di Crescenzo, a cryptographic expert and Senior Scientist, Applied Communication Sciences. His published work on RPM includes (http://csrc.nist.gov/groups/ST/lwc-workshop2015/presentations/session3-dicrescenzo.pdf) and his paper on RPM/Qwyit is currently under review by NIST for certification as a US National Standard for Lightweight Cryptography (http://csrc.nist.gov/groups/ST/lwc-workshop2015/papers/session3-dicrescenzo-paper.pdf).

"The core technology of the Qwyit System is a new method for generating a sequence of master keys, with derived session and child keys, for use in encryption and authentication. This core technology is based on sound principles of randomization, derived keys, and presenting the adversary under certain attacks with underdetermined equations."

– *By Dr. Alan T. Sherman, May 27, 2005, An Initial Assessment of the Qwyit Authentication and Key-Management System: Highlights*

"The realized security is sharing a pair of common-credentials, sharing a common-key, secrecy of messages, sender authentication, common-key authentication, message authentication, common-key renewal, renewal of a pair of common-credentials, etc. Such a security-function integrated simple communication system will be useful for the future wireless communication system such as handy phones and ubiquitous networks."

– *By Dr. Hatsukazu Tanaka, SCIS 2006 The 2006 Symposium on Cryptography and Information Security Hiroshima, Japan, Jan. 17-20, 2006, The Institute of Electronics, Information and Communication Engineers*

Qwyit Reference Code

The following is the reference Qwyit function and protocol code, along with test vectors. It is written in Visual Basic 6.0 (discontinued), but easily translated to other platforms. Qwyit LLC has this reference software library available in C, C++ and Java. Contact us for the latest versions.

Qwyit Functions and Protocol – Reference Code and Test Vectors

The Qwyit Reference Primitive Functions in VB6

```
'*********************************************************************************
'
'
'          The following is the Qwyit Reference Implementation Example
'                        CODE TOOLKIT
'
'                 Visual Basic Qwyit Module
'
'Created By:       Qwyit LLC
'DATE:            8/10/2017
'VERSION:          2.0
'
'This material is COPYRIGHTED and PATENTED
'You may use this material ONLY AFTER HAVING PROCURED A
'Qwyit license or permission to use from Qwyit.com
'Your cooperation in this regard is appreciated.
'
'Contact:
'www.Qwyit.com
'info@Qwyit.com
'*********************************************************************************
```

```
' FUNCTION: MOD16 Encrypt Function - String Return
'
' NAME:     MOD16
'
' PURPOSE:  Returns the Add Without Carry result of two input numbers,
'           with the first number (sT1) as the longer (if unequal lengths)
'
' TYPE:     Qwyit specific function - uses strings
'
' CALL:     MOD16(sT1, sT2, Optional sT3)
'           where sT1 is the first (and longer) input number, sT2 is the second,
'             and sT3 an Optional third number (not currently used)
'
' RTRN:     STRING value of Mod16 result, Hex represented
'
' ERROR:    Null return
'
' Example:  MOD16("0BC34","F4321") will return "FFF55"
'
' Test Vector:  The example is correct for two strings,
'               for three strings, MOD16("0BC34","F4321","12345") will return "0129A"
'
Public Function MOD16(sT1 As String, sT2 As String, Optional sT3 As String) As String
Dim nN As Integer
Dim nN_I As Integer
Dim nT1_n As Integer
Dim nT2_n As Integer
Dim sOUT_n As String
Dim sOUT As String
Dim nMod As Integer
Dim sTmp As String

If sT1 = Null Or sT2 = Null Or sT1 = "" Or sT2 = "" Then
   MOD16 = ""
   Exit Function
Else

   nN = 1
   nN_I = 1
   While nN <= Len(sT1)
      nT1_n = Val("&H" & Mid(sT1, nN, 1))
      If nN_I > Len(sT2) Then
         nN_I = 1
      End If
      nT2_n = Val("&H" & Mid(sT2, nN_I, 1))
      nMod = (nT1_n + nT2_n) Mod 16
      sOUT_n = Hex(nMod)
      sOUT = sOUT & sOUT_n
      nN = nN + 1
```

```
        nN_I = nN_I + 1
    Wend
    If sT3 <> "" Then
        sTmp = sOUT
        sOUT = ""
        nN = 1
        nN_I = 1
        While nN <= Len(sTmp)
            nT1_n = Val("&H" & Mid(sTmp, nN, 1))
            If nN_I > Len(sT3) Then
                nN_I = 1
            End If
            nT2_n = Val("&H" & Mid(sT3, nN_I, 1))
            nMod = (nT1_n + nT2_n) Mod 16
            sOUT_n = Hex(nMod)
            sOUT = sOUT & sOUT_n
            nN = nN + 1
            nN_I = nN_I + 1
        Wend
    End If
    MOD16 = sOUT
End If
End Function
```

```
' FUNCTION: MOD16 Decrypt Function - String Return
'
' NAME:     MOD16D
'
' PURPOSE:  Returns the Add Without Carry decrypt result of two input
'           numbers, with the first number (sT1) as the Ciphertext, sT2 is the key
'
' TYPE:     Qwyit specific function - uses strings
'
' CALL:     MOD16D(sT1, sT2)
'           where sT1 is the ciphertext number, sT2 is the key number
'
' RTRN:     STRING value of Mod16D result
'
' ERROR:    Null return
'
' Example:  MOD16D("FFF55", "0BC34") will return "F4321"
'
' Test Vector:  The example is correct
'
Public Function MOD16D(sT1 As String, sT2 As String) As String

Dim nN As Integer
Dim nN_I As Integer
Dim nT1_n As Integer
Dim nT2_n As Integer
Dim sOUT_n As String
Dim sOUT As String
Dim nMod As Integer

If sT1 = "" Or sT2 = "" Then
      MOD16D = ""
      Exit Function
Else

   nN = 1
   nN_I = 1
   While nN <= Len(sT1)
      nT1_n = Val("&H" & Mid(sT1, nN, 1))
      If nN_I > Len(sT2) Then
         nN_I = 1
      End If
      nT2_n = Val("&H" & Mid(sT2, nN_I, 1))
      nMod = ((16 + nT1_n) - nT2_n) Mod 16
      sOUT_n = Hex(nMod)
      sOUT = sOUT & sOUT_n
      nN = nN + 1
      nN_I = nN_I + 1
   Wend
```

```
    MOD16D = sOUT
End If
End Function
```

```
' FUNCTION: One-Way Cut Function - String Return
'
' NAME:     OWC
'
' PURPOSE:  Returns the Message Key result of an even-digit input key,
'           cut in half using the indicated digits
'
' TYPE:     Qwyit specific function - uses strings
'
' CALL:     OWC(sKey, nT, Optional nSkip)
'           where sKey is the Key number as a string,
'           nT is type, with 0 = decimal, 1 (or above) = Hex,
'           nSkip is the number of digit positions to skip for pair selection,
'           (default is adjacent digits where nSkip = 1; if <1, will be set to 1),
'
' RTRN:     STRING value Output Key result
'
' ERROR:    Null return
'
' Example:  OWC("FCB578",1,1) will return an Output Key of "B0F"
'
' Test Vector:  The example is correct
'
Public Function OWC(sKey As String, nT As Integer, Optional nSkip As Integer) As String

Dim nN As Integer
Dim nT1_n As Integer
Dim nT2_n As Integer
Dim sOUT_n As String
Dim sOUT As String
Dim nMod As Integer
Dim nKey As Integer

If sKey = "" Or sKey = Null Then
      OWC = ""
      Exit Function
Else
   nKey = Len(sKey)
   If nSkip = 0 Or nSkip > 0.5 * nKey Then 'if Skip is greater than 1/2 key length, so reset
      nSkip = 1
   End If

   If nT = 0 Then           'Decimal
      'This section is optional, as almost all implementations would use hex keys
      nN = 1
      While nN < nKey
         nT1_n = Val(Mid(sKey, nN, 1))
         sCheck = Mid(sKey, nN + nSkip, 1)
         'If the skip doesn't divide evenly into nKey, pair the remaining adjacently
```

```
        If sCheck = "" Then
          nT2_n = Val(Mid(sKey, nN + 1, 1))
          nN = nN + 1
        Else
          nT2_n = Val(sCheck)
        End If
        nMod = (nT1_n + nT2_n) Mod 10
        sOUT_n = Trim(Str(nMod))
        sOUT = sOUT & sOUT_n
        If nN Mod nSkip = 0 Then
          nN = nN + nSkip + 1
        Else
          nN = nN + 1
        End If
      Wend
  Else                    'Hex
      nN = 1
      While nN < nKey
        nT1_n = Val("&H" & Mid(sKey, nN, 1))
        sCheck = Mid(sKey, nN + nSkip, 1)
        'If the skip doesn't divide evenly into nKey, pair the remaining adjacently
        If sCheck = "" Then
          nT2_n = Val("&H" & Mid(sKey, nN + 1, 1))
          nN = nN + 1
        Else
          nT2_n = Val("&H" & sCheck)
        End If
        nMod = (nT1_n + nT2_n) Mod 16
        sOUT_n = Hex(nMod)
        sOUT = sOUT & sOUT_n
        If nN Mod nSkip = 0 Then
          nN = nN + nSkip + 1
        Else
          nN = nN + 1
        End If
      Wend
  End If
  OWC = sOUT
End If
End Function
```

```
' FUNCTION: Position Digit Algebra Function (PDAF)
'
' NAME:     PDAF
'
' PURPOSE:  Returns a key expansion based on single or dual key input
'
' TYPE:     Qwyit specific function - uses strings
'
' CALL:     PDAF(sValueKey, nKey, optional nMode, optional sOffsetKey,
'             optional nPI, optional nCI, optional nT)
'           where sValueKey is the Value Key from which to select,
'           nKey is the length of key material returned (0 means return all
'           cycles, else number is key length in number of characters
'           (the number of returned digits will be twice this number)),
'           nMode is whether to add only from the Value Key (nMode=0, default)
'           or whether to add from the OffsetKey,
'           sOffsetKey is the key from which to set the offset (if blank, use
'           the VK),
'           nPI is the Pointer Index (where to start the pointer, default=1)
'           (note: this function creates PDAF values in pairs, so the PI will
'           round up to start on the next pair),
'           nCI is the Cycle Index (at what cycle to start the SI and PI, default=0),
'           and nT is the type of return, 0 (default) is a string, 1 is a byte array
'
' RTRN:     String or Byte result of the key material
'
' ERROR:    Null return
'
' Example:  PDAF("1234", 4, 0, "", 0, 0, 0)
'           would return (3658)
'
' Test Vector: PDAF("9203BA8F", 0, 0, "", 1, 0, 0) returns
'              "9D32437ECCBCDC184AA5BAA13183ED8F1BF665B2849E543A222D3229B50BA907"
'             PDAF("9203BA8F", 0, 0, "55F82C01", 1, 0, 0) returns
'              "110EAA718B3D4D1F24BBD5A2B2A2B48A958CE2B9CDF569374C93532E3A263C08"
'             PDAF("682D", 7, 1, "45A1", 1, 0, 0) returns "E5A58E8335F58A"
'             PDAF("29FB", 11, 0, "74E0", 2, 5, 0) returns "4C8B2FBEE2E14510040FFA"
'
Public Function PDAF(sValueKey As String, nKey As Integer, _
    Optional nMode As Integer, Optional sOffsetKey As String, _
    Optional nPI As Integer, Optional nCI As Integer, _
    Optional nT As Integer) As Variant

Dim t_Key() As Byte
Dim t_sKey As String
Dim sKey As String
Dim sAddKey As String
Dim sPointKey As String
Dim sHoldTemp As String
```

```
Dim sHoldFinal As String
Dim sTmpVal As String

If sValueKey = Null Or sValueKey = "" Or nLen < 0 Or nPI > Len(sValueKey) Then
   PDAF = ""
   Exit Function
Else
   If nPI < 1 Then
      nPI = 1
   End If
   If nCI = 0 And nPI = 1 Then
      nSelectF = 1
   End If
   If sOffsetKey = "" Or sOffsetKey = Null Then
      sOffsetKey = sValueKey
   End If
   sAddKey = sValueKey
   sPointKey = sOffsetKey

  If nT = 1 Then 'Byte array return...
   nLen = Len(sAddKey)
   If nKey = 0 Then
      nN_Max = nLen * nLen
      ReDim t_Key(nN_Max)
   Else
      nN_Max = nKey * 2
      ReDim t_Key(nN_Max)
   End If

   While Len(sAddKey) < (2 * nLen) + 17
      sAddKey = Trim(Trim(sAddKey) & Trim(sAddKey))
   Wend
   nLen2 = Len(sPointKey)
   While Len(sPointKey) < (2 * nLen2) + 17
      sPointKey = Trim(Trim(sPointKey) & Trim(sPointKey))
   Wend
   p = 1 'Pointer starts at nPI or 1
   c = 0 'Cycle starts at 0
   nN = 0  'Counter for how much key material
   Do

     Formula1 = Val("&H" & Mid(sAddKey, p, 1))
     If nMode = 0 Then
        nOffset = Val("&H" & Mid(sPointKey, p, 1)) + 1
        Formula2 = Val("&H" & Mid(sAddKey, p + nOffset + c, 1))
     Else
        nOffset = Val("&H" & Mid(sPointKey, p + c, 1)) + 1
        Formula2 = Val("&H" & Mid(sAddKey, p + nOffset, 1))
     End If
```

```
      sTmpVal = Hex((Formula1 + Formula2) Mod 16)
      If nSelectF = 1 Then
         t_Key(nN) = CByte("&H" & sTmpVal)
         nN = nN + 1
      ElseIf c + (d * nLen) >= nCI And p >= nPI Then
         nSelectF = 1
         t_Key(nN) = CByte("&H" & sTmpVal)
         nN = nN + 1
      End If
      sHoldTemp = sHoldTemp & sTmpVal
      p = p + 1
      If nN = nN_Max Then
         Exit Do
      End If
       If p > nLen Then
         p = 1
         c = c + 1
         If sHoldFinal <> "" Then
            sHoldFinal = MOD16(sHoldTemp, sHoldFinal)
         Else
            'sHoldFinal = MOD16(sHoldTemp, sAddKey)
            sHoldFinal = sHoldTemp
         End If
         If c = nLen Then
            d = d + 1
            c = 0
            sAddKey = MOD16(sHoldFinal, sAddKey)
            sPointKey = MOD16(sHoldTemp, sPointKey)
            sHoldFinal = ""
            While Len(sAddKey) < (2 * nLen) + 17
               sAddKey = Trim(Trim(sAddKey) & Trim(sAddKey))
            Wend
            nLen2 = Len(sPointKey)
            While Len(sPointKey) < (2 * nLen2) + 17
               sPointKey = Trim(Trim(sPointKey) & Trim(sPointKey))
            Wend
         End If
         sHoldTemp = ""
       End If
    Loop
    PDAF = t_Key()
  Else  'String return
    nLen = Len(sAddKey)
    If nKey = 0 Then
       nN_Max = nLen * nLen
    Else
       nN_Max = nKey * 2
    End If
```

```
While Len(sAddKey) < (2 * nLen) + 17
   sAddKey = Trim(Trim(sAddKey) & Trim(sAddKey))
Wend
nLen2 = Len(sPointKey)
While Len(sPointKey) < (2 * nLen2) + 17
   sPointKey = Trim(Trim(sPointKey) & Trim(sPointKey))
Wend
p = 1 'Pointer starts at nPI or 1
c = 0 'Cycle starts at 0
nN = 0  'Counter for how much key material
Do
  Formula1 = Val("&H" & Mid(sAddKey, p, 1))
  If nMode = 0 Then
     nOffset = Val("&H" & Mid(sPointKey, p, 1)) + 1
     Formula2 = Val("&H" & Mid(sAddKey, p + nOffset + c, 1))
  Else
     nOffset = Val("&H" & Mid(sPointKey, p + c, 1)) + 1
     Formula2 = Val("&H" & Mid(sAddKey, p + nOffset, 1))
  End If

  sTmpVal = Hex((Formula1 + Formula2) Mod 16)
  If nSelectF = 1 Then
     t_sKey = t_sKey & sTmpVal
     nN = nN + 1
  ElseIf c + (d * nLen) >= nCI And p >= nPI Then
     nSelectF = 1
     t_sKey = t_sKey & sTmpVal
     nN = nN + 1
  End If
  sHoldTemp = sHoldTemp & sTmpVal
  p = p + 1
  If nN = nN_Max Then
     Exit Do
  End If
   If p > nLen Then
     p = 1
     c = c + 1
     If sHoldFinal <> "" Then
        sHoldFinal = MOD16(sHoldTemp, sHoldFinal)
     Else
        'sHoldFinal = MOD16(sHoldTemp, sAddKey)
        sHoldFinal = sHoldTemp
     End If
     If c = nLen Then
        d = d + 1
        c = 0
        sAddKey = MOD16(sHoldFinal, sAddKey)
        sPointKey = MOD16(sHoldTemp, sPointKey)
```

```
            sHoldFinal = ""
            While Len(sAddKey) < (2 * nLen) + 17
               sAddKey = Trim(Trim(sAddKey) & Trim(sAddKey))
            Wend
            nLen2 = Len(sPointKey)
            While Len(sPointKey) < (2 * nLen2) + 17
               sPointKey = Trim(Trim(sPointKey) & Trim(sPointKey))
            Wend
         End If
         sHoldTemp = ""
      End If
   Loop
   PDAF = t_sKey
  End If

End If
End Function
```

```
' FUNCTION: Byte to 4-bit to Byte Translation Function
'
' NAME:     BT4
'
' PURPOSE:  Compresses or expands full byte numbers from/to 4-bit Hex numbers;
'           e.g., in order to decrease the ON, OR transmissions (cut them in half),
'           simply call this function with them as sLong values, then send the
'           compressed sShort version - for decrypt, then send those sShort values
'           back through here and retrieve the sLong values.
'
' TYPE:     General function - might be needed to read/return key bit streams as
'             4-bit hex numbers
'
' CALL:     BT4(sLong, sShort, Optional nT)
'           where sLong is a full byte string - each byte is one number (hex or dec),
'           sShort is 4-bit numbers in a string - each byte is two hex numbers,
'           nT is for byte (1), or string (0, default) return
'
' RTRN:     If sLong is sent, sShort is returned - and the opposite
'
' ERROR:    Null return
'
' Example:  BT4("6D50", "") returns "mP"
'           BT4("", "Qb") returns "5162"
'
' Test Vector: The examples are test vectors
'
Public Function BT4(sLong As String, sShort As String, _
   Optional nT As Integer) As Variant

Dim nLen As Integer

If sLong = "" And sShort = "" Then
   BT4 = ""
   Exit Function

ElseIf sLong <> "" Then     'Long to Short, with Long formatted as in "F59B"
   nLen = Len(sLong)
   If nLen Mod 2 = 1 Then  'Odd sLong check
      nOdd = 1
   End If
   p = 1
   For q = 1 To ((nLen + 1) / 2)
      BT4 = BT4 & Chr(Val("&H" & Mid(sLong, p, 2)))
      If nOdd = 1 And (p + 2) > nLen Then  'Odd sLong
         p = p + 1
         BT4 = BT4 & Chr(Val("&H" & Mid(sLong, p, 1)))
      Else    'sLong is even
         p = p + 2
```

```
      End If
   Next q

ElseIf sShort <> "" Then    'Short to Long, with Short formatted as in "Qb"
   nLen = Len(sShort)
   For q = 1 To nLen
      sTmp = Hex(Asc(Mid(sShort, q, 1)))
      If Len(sTmp) < 2 Then   'Must add a "0" to the string
         sTmp = "0" & sTmp
      End If
      BT4 = BT4 & sTmp
   Next q

End If

If nT = 1 Then  'Byte return
   BT4 = StrConv(BT4, vbFromUnicode)
End If
End Function
```

```
' FUNCTION: Qwyit Combine Function
'
' NAME:     Combine
'
' PURPOSE:  Function to permute the K1 key with the random salt (R):
'
'           Combine R and K1, resulting in a n-bit 'alphabet', A
'
' TYPE:     General function - Qwyit Combine call
'
' CALL:     Combine(sR, sK1)
'           where sK1 is the starting key value, sR is the random salt
'
' RTRN:     A string value of: sA
'
' ERROR:    Null return
'
' Example:
Combine("45384189FE42A1C1A00F795AA9A0819ED39BBEBF19FBF40F6AEB4C6B362A56DC",
'           "0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF") returns
'           "A: 8DF5857C06A9D6DDE421EB4F362E766A1BEA6733FC41F8F0728634720FFF52D7"
'
' Test Vector: The example is a test vector
'
Public Function Combine(sRand As String, sKey1 As String) As Variant

Dim sA As String
Dim sRTemp As String
Dim sK1Temp As String
Dim sRi As String
Dim sK1i As String
Dim nCount As Integer
Dim nCount1 As Integer
Dim i As Integer
Dim nStart As Integer

If sRand = "" Or sKey1 = "" Then
   Combine = ""
   Exit Function
Else

   'Combine R and K1, resulting in a n-bit 'alphabet', A
   'sA = Combine(sR, sK1)
   nCount = Len(sRand)
   nCount1 = Len(sKey1)
   If nCount <> nCount1 Then
      Combine = ""
      Exit Function
   End If
```

```
    sRTemp = sRand
    sK1Temp = sKey1
    nStart = 0

    For i = 1 To nCount
        nVal = nStart + Val("&H" & Mid(sKey1, i, 1)) + 1
        If nVal > nCount Then
            nVal = nVal - nCount
            nStart = 0
        End If
        sRi = sRi & Mid(sRand, nVal, 1)
        nStart = nVal
    Next

    nStart = 0
    For i = 1 To nCount
        nVal = nStart + Val("&H" & Mid(sRand, i, 1)) + 1
        If nVal > nCount Then
            nVal = nVal - nCount
            nStart = 0
        End If
        sK1i = sK1i & Mid(sKey1, nVal, 1)
        nStart = nVal
    Next

    '********THESE ARE TEST VERSIONS - moving the string to be sure of selection..(2 slow!)
    'First, get the R intermediate key string, sRi
    'For i = 1 To nCount
    '    nVal = Val("&H" & Mid(sKey1, i, 1)) + 1
    '    sRi = sRi & Mid(sRTemp, nVal, 1)
    '    sRTemp = Right(sRTemp, nCount - nVal) & Left(sRTemp, nVal)
    'Next

    'Next, get the K1 intermediate key string, sKi
    'For i = 1 To nCount1
    '    nVal = Val("&H" & Mid(sRand, i, 1)) + 1
    '    sK1i = sK1i & Mid(sK1Temp, nVal, 1)
    '    sK1Temp = Right(sK1Temp, nCount1 - nVal) & Left(sK1Temp, nVal)
    'Next
    '*****************************************************************************************

    'Last, mod add the two intermediate strings
    Combine = MOD16(sRi, sK1i)

End If
End Function
```

```
' FUNCTION: Qwyit Extract Function
'
' NAME:     Extract
'
' PURPOSE:  Function to extract the W key from the A alphabet:
'
'          Extract n-bit key W out of A using K1
'
' TYPE:     General function - Qwyit Extract call
'
' CALL:     Extract(sA, sK1)
'          where sK1 is the starting key value, sA is the combined alphabet
'
' RTRN:     A string value of: sW
'
' ERROR:    Null return
'
' Example:  Extract("8DF5857C06A9D6DDE421EB4F362E766A1BEA6733FC41F8F0728634720FFF52D7",
'             "0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF") returns
'          "W: 8F56DEEAF7D62F2C0A6447A13D6BE77DE2B66616574640CF326B3F6F8D6788DA"
'
' Test Vector: The example is a test vector
'
Public Function Extract(sAlphabet As String, sKey1 As String) As Variant

Dim sATemp As String
Dim sW As String
Dim nCount As Integer
Dim nCount1 As Integer
Dim i As Integer

If sAlphabet = "" Or sKey1 = "" Then
   Extract = ""
   Exit Function
Else

   'Extract n-bit key W out of A using K1
   'sW = Extract(sA, sK1)
   nCount = Len(sAlphabet)
   nCount1 = Len(sKey1)
   If nCount <> nCount1 Then
      Extract = ""
      Exit Function
   End If
   sATemp = sAlphabet
   nStart = 0

   For i = 1 To nCount
      nVal = nStart + Val("&H" & Mid(sKey1, i, 1)) + 1
```

```
    If nVal > nCount Then
        nVal = nVal - nCount
        nStart = 0
    End If
    sW = sW & Mid(sAlphabet, nVal, 1)
    nStart = nVal
Next

'************TEST VERSION - just moving the string, but 2 slow!
'First, get the R intermediate key string, sRi
'For i = 1 To nCount
'    nVal = Val("&H" & Mid(sKey1, i, 1)) + 1
'    sW = sW & Mid(sATemp, nVal, 1)
'    sATemp = Right(sATemp, nCount - nVal) & Left(sATemp, nVal)
'Next
'*************************************************************

'Last, mod add the two intermediate strings
Extract = sW

End If
End Function
```

```
' FUNCTION: Qwyit Stream Cipher - Data Security Function - (both Encrypt/Decrypt)
'
' NAME:     QwyitSCX
'
' PURPOSE:  A full bundled function to perform the Qwyit Stream Cipher data security (enc/dec)
'           THIS IS A VB EXAMPLE - it uses an XOR as the enc/dec
'
'Start: For SEND, generate new n-bit random salt OR at system defined length (256-bits recommended)
'       GetRandom(OR)
' Now all is the same for Send and Receive, except Step 4:
'1.  Perform R = MOD16 (EK, OR)
'2.  Combine QK, R, resulting in a n-bit 'alphabet', A
'3.  Extract n-bit key W out of A using QK
'4.  SEND: Perform MOD16(W, PT) for CT
'4.  RECEIVE: Perform MOD16D(CT, W) for PT
'5.  Perform MOD16(W,R) for next R used in next 256-bit block of PT/CT
' Then, Sender sends the following to the recipient: OpenIDSender, OR, C
'    where OpenIDSender is the publicly-known identification of the sender,
'  or Receiver gets PT revealed after cycling through the received CT
'
' TYPE:     General function - Qwyit Stream Cipher call, using XOR
'
' CALL:     QwyitSCX(sQK, sEK, sOR, sPT, sCT)
'           where QK is the Qwyit (Authentication) key, EK is the Exchange Key, sPT is the plaintext, sCT is the
ciphertext
'               and sOR is the Open Return (or Next R, cyclically)
'
' RTRN:     A string value of: sR, sT, sW
'               where sT is either the PT or CT, opposite of what was input, sW is only returned for testing purposes -
never sent!
'               and sT is the Next R to be used for the next PT or CT block - it is never sent either!
'
' ERROR:    Null return
'
' Example:  See QwyitCipher Reference Vectors file and outputs
' Test Vectors: See QwyitCipher Reference Vectors file and outputs
'
Public Function QwyitSCX(sQK As String, sEK As String, sOR As String, Optional sPT As String, Optional sCT
As String) As Variant

Dim sA As String
Dim sW As String
Dim sR As String
Dim sPTtemp As String
Dim sT As String
Dim sNextR As String
Dim i As Integer
Dim j As Integer
Dim sTtemp As String
```

```
Dim nCT As Integer
Dim nPT As Integer
Dim sKeyStream As String

If sQK = "" Or sEK = "" Then
   QwyitSCX = ""
   Exit Function
Else

   nCT = Len(sCT)
   nPT = Len(sPT)
   sR = MOD16(sEK, sOR)

   'Combine R and QK, resulting in a n-bit 'alphabet', A
   sA = Combine(sR, sQK)
   If sA = "" Then
      QwyitSCX = ""
      Exit Function
   End If

   'Extract n-bit key W out of A using QK
   sW = Extract(sA, sQK)
   If sW = "" Then
      QwyitSCX = ""
      Exit Function
   End If

   If nPT > 0 Then
   'Encrypt the plaintext message using W
      j = 0
      nW = Len(sW)
      sKeyStream = sW
      For i = 1 To nPT
         j = j + 1
         If j > nW Then
         'Get Next R for both Enc/Dec
            sNextR = MOD16(sW, sR)
            sA = Combine(sNextR, sQK)
            sW = Extract(sA, sQK)
            nW = Len(sW)
            sKeyStream = sKeyStream & sW
            j = 1
         End If
         sTtemp = Asc(Mid(sW, j, 1)) Xor Asc(Mid(sPT, i, 1))
         sT = sT & Chr(sTtemp)
      Next i
   Else
   'Decrypt the ciphertext message using W
      j = 0
```

```
      nW = Len(sW)
      sKeyStream = sW
      For i = 1 To nCT
         j = j + 1
         If j > nW Then
            'Get Next R for both Enc/Dec
            sNextR = MOD16(sW, sR)
            sA = Combine(sNextR, sQK)
            sW = Extract(sA, sQK)
            nW = Len(sW)
            sKeyStream = sKeyStream & sW
            j = 1
         End If
         sTtemp = Asc(Mid(sW, j, 1)) Xor Asc(Mid(sCT, i, 1))
         sT = sT & Chr(sTtemp)
      Next i
   End If
'*******************

   QwyitSCX = "R: " & sNextR & Chr$(13) & Chr$(10) & _
         "W: " & sW & Chr$(13) & Chr$(10) & _
         "T: " & sT & Chr$(13) & Chr$(10) & _
         "X: " & sKeyStream & Chr$(13) & Chr$(10)

End If
End Function
```

Appendix – Qwyit Functions in Pseudo-Code

For help in understanding the workings of each core code function, here are pseudo-code explanations, along with examples:

' FUNCTION: MOD16 Encrypt Function - String Return

```
'
' NAME:     MOD16
'
' PURPOSE:  Returns the Add Without Carry result of two input numbers,
'           with the first number (sT1) as the longer (if unequal lengths)
'
' TYPE:     Qwyit specific function - uses strings, but other languages use arrays
'
' CALL:     MOD16(sT1, sT2, Optional sT3)
'           where sT1 is the first (and longer) input number, sT2 is the second,
'             and sT3 an Optional third number (not currently used)
'
' RTRN:     STRING value of Mod16 result, Hex represented
'
' ERROR:    Null return
'
' Example:  MOD16("0BC34","F4321") will return "FFF55"
'
' Test Vector:  The example is correct for two strings,
'               for three strings, MOD16("0BC34","F4321","12345") will return "0129A"
'
```

The simple MOD Algorithm (Hexadecimal)
(a + b) Mod 16 = n with only one variable known for any use, and a, b and n are all single digits of the input number.

- *String1 in (the longest of the two)*
- *String2 in*
- *For each digit of the length of String1*
    - *(String1_Digit_Next + String2_Digit_Next) Mod 106= Result_Digit_Next*
    - *If end of String1, quit*
    - *If end of String2, begin again at digit 1*
- *Return all the Result_Digits*

' FUNCTION: MOD16 Decrypt Function - String Return

'
' NAME: MOD16D
'
' PURPOSE: Returns the Add Without Carry decrypt result of two input
' numbers, with the first number (sT1) as the Ciphertext, sT2 is the key
'
' TYPE: Qwyit specific function - uses strings, but other languages use arrays
'
' CALL: MOD16D(sT1, sT2)
' where sT1 is the ciphertext number, sT2 is the key number
'
' RTRN: STRING value of Mod16D result
'
' ERROR: Null return
'
' Example: MOD16D("FFF55", "0BC34") will return "F4321"
'
' Test Vector: The example is correct
'

The simple MOD Reverse Algorithm (Hexadecimal)
((16+n) - b)) Mod 16 = a where n is the ciphertext value and b is the known key value and the result a, is the resulting plaintext next value now known (another key, etc.)

- *String1 in (the longest of the two)*
- *String2 in*
- *For each digit of the length of String1*
    - *((16+String1_Digit_Next) - String2_Digit_Next) Mod 16 = Result_Digit_Next*
    - *If end of String1, quit*
    - *If end of String2, begin again at digit 1*
- *Return all the Result_Digits*

' FUNCTION: One-Way Cut Function - String Return

'

' NAME: OWC

'

' PURPOSE: Returns the Message Key result of an even-digit input key,
' cut in half using the indicated digits

'

' TYPE: Qwyit specific function - uses strings, but other languages use arrays

'

' CALL: OWC(sKey, nT, Optional nSkip)
' where sKey is the Key number as a string,
' nT is type, with 0 = decimal, 1 (or above) = Hex,
' nSkip is the number of digit positions to skip for pair selection,
' (default is adjacent digits where nSkip = 1; if <1, will be set to 1),

'

' RTRN: STRING value Output Key result

'

' ERROR: Null return

'

' Example: OWC("FCB578",1,1) will return an Output Key of "B0F"

'

' Test Vector: The example is correct

'

A true one-way function accomplished by algorithm (MOD) summing of adjacent digits of the input string.

- *For all the digit pairs in the input string separated by the nSkip value, (either a decimal or hex cut)*
  - *Select the next pair of digits*
  - *MOD 16 sum them into a single return value*
  - *If the skip does not divide evenly into the key length, pair the remaining pairs adjacently*
  - *Also, pair the selected digits in 'blocks' of up to the skip value, then jump to next unused digit (E.g., if the nSkip is 4, and the key is "123412345678567890", then the "1"s would be paired, then the "2"s, then the "3"s, then the "4"s, and then that block is complete, so begin again at the "5"s, etc. – until there is just the "90" remaining, and those are paired adjacently (There is no effect on the "security" of this number based on this arbitrary coupling – the security is in the coupling result, not in which digits were coupled)*
- *Return all of the digit pair sums*

' FUNCTION: Position Digit Algebra Function (PDAF)

```
'
' NAME:     PDAF
'
' PURPOSE:  Returns a key expansion based on single or dual key input
'
' TYPE:     Qwyit specific function - uses strings, but other languages use arrays
'
' CALL:     PDAF(sValueKey, nKey, optional nMode, optional sOffsetKey,
'            optional nPI, optional nCI, optional nT)
'          where sValueKey is the Value Key from which to select,
'          nKey is the length of key material returned (0 means return all
'          cycles, else number is key length in number of characters
'          (the number of returned digits will be twice this number)),
'          nMode is whether to add only from the Value Key (nMode=0, default)
'          or whether to add from the OffsetKey,
'          sOffsetKey is the key from which to set the offset (if blank, use
'          the VK),
'          nPI is the Pointer Index (where to start the pointer, default=1)
'          (note: this function creates PDAF values in pairs, so the PI will
'          round up to start on the next pair),
'          nCI is the Cycle Index (at what cycle to start the SI and PI, default=0),
'          and nT is the type of return, 0 (default) is a string, 1 is a byte array
'
' RTRN:     String or Byte result of the key material
'
' ERROR:    Null return
'
' Example:  PDAF("1234", 4, 0, "", 0, 0, 0)
'          would return (3658)
'
' Test Vector: PDAF("9203BA8F", 0, 0, "", 1, 0, 0) returns
'             "9D32437ECCBCDC184AA5BAA13183ED8F1BF665B2849E543A222D3229B50BA907"
'            PDAF("9203BA8F", 0, 0, "55F82C01", 1, 0, 0) returns
'             "110EAA718B3D4D1F24BBD5A2B2A2B48A958CE2B9CDF569374C93532E3A263C08"
'            PDAF("682D", 7, 1, "45A1", 1, 0, 0) returns "E5A58E8335F58A"
'            PDAF("29FB", 11, 0, "74E0", 2, 5, 0) returns "4C8B2FBEE2E14510040FFA"
'
```

The function, as a result of this new type of 'positional math' requires some unique notation:

- *Current Value Pointer – P*
- *Current Cycle counter – CC*
- *Value Key – VK, where this number provides the input values into the LKE*
- *Offset Key – OK, where this number provides the values used to select the VK input values. When the input is a single number, it is both the VK and the OK. When the input numbers are different, then one is the VK, the other the OK.*
- *Position within a number is notated using the "<>" designator, such that VK<P> would indicate the Current Value Pointer's value within the Value Key. If P = 1, then this would mean the first digit.*

Using this notation, a PDAF function returns a string of digits or byte array that is equal to either the total number of cyclic digit position sums (length^2 for all cycles) or to the input requested key stream length. The order in which the digits are returned:

- *Set a Pointer P to 1*
- *Set a Cycle Counter CC to 0*
- *Taking the input material (sKey), for either all of the cycles or just for a subset of them, then*
  a. *Start by returning the digit value at the Pointer P*
     DV = sKey<P>
  b. *Mod 16 the Pointer position digit's value with the value at the value's position to the right of the Pointer and also add the CycleCounter's current value to the position; the Pointer digit position immediately to the right of the*

current value is position 0, two to the right is position one, etc. [E.g., if the digits of the number are "362784290…", and the current CycleCounter value is "4" then the first number sum in that cycle is (3 + 0) Mod 16]

PDAF(P) = (DV + sKey<DV + CC>) Mod 16

c. *Then Increment the Pointer by one and repeat the DV selection and the Mod add for that digit position*

P = P + 1, DV = sKey(P), PDAF(P) = (DV + sKey<DV + CC>) Mod 16

d. *Keep temporary storage of this cycle's values in HoldTemp*

e. *When the Pointer value is greater than the length of the number, increment the CycleCounter by one and reset the Pointer to one and again, repeat the DV selection and the Mod add for that digit*

If P > Length(sKey) Then CC = CC + 1 and P = 1, then DV = …

f. *Keep temporary storage of the just-finished full HoldTemp as the initial HoldFinal, and reset HoldTemp to null. Each subsequent additional cycle will modulus 16 sum the current HoldTemp and the last version of HoldFinal, requiring any cryptanalysis to be aware of every PDAF(P) value created.*

g. *If the returned PDAF key is to be all the cycles, then exit when the CycleCounter is equal to the length of the number (and HoldTemp and HoldFinal are not used). If the returned PDAF key is larger than that, then when the CycleCounter is greater than the number length, replace both the original sKey number where it is the VK, and also where it is the OK. The VK replacement is the last version of HoldFinal (which is the sum of all of the PDAF values thus far and the initial sKey); the OK replacement is the sKey value modulus 16 summed with the last version of HoldTemp. Then reset the Pointer to one, and the CycleCounter to 0, and begin again until either the VK and OK numbers are replaced in the same manner, or the return length is reached.*

If PDAF(Desired Key Stream Length) < all the cycles, then exit when reach DKSL
Else If PDAF(DKSL) = all cycles, then exit when CC = Length(sKey)
Else If PDAF(DKSL) > all cycles, then when CC = Length(sKey),

Replace sKey in the VK position with the cyclic summed value HoldFinal, also replace sKey in the OK position with the last version of HoldTemp summed with the original OK (sKey in this case) which results in a new sKeys of equal length for VK and OK, and then set CC = 0, P = 1 and repeat on new sKey, exchanging the sKey in a like manner until reaching the PDAF(DKSL)

- *Return all the summed numbers in either a byte array or string*

A PDAF example:

PDAF(1234) = (1+3) and (2+1) and (3+3) and (4+1) and (1+4) and (2+2) and (3+4) and (4+2) and … continuing until the cycle pointer is greater than 4, and then replacing 1234 with 5050 as the VK, and 3658 as the OK. Then begin again with (5+5) and (0+5) and … all Mod 16 until the desired length is reached for 436554762547365805….until reaching the Desired Key Stream Length end. It should be noted here that PDAF returns have a more full distribution of "unknowns" when using inputs at over 15 digits in length – the above simple 4-digit input has predictable (at least less attempts) than a longer number (regardless of the distribution of the values in a longer number, the possibilities of the input values increases).

The PDAF can use two different numbers instead of a single number (to start, as after all cycles are reached, the above 'replacement' method substitutes as new OK). One is the Offset Key (OK) and the other the Value Key (VK). The steps in a), b) and g) above differ slightly:

a. *Start by returning the digit value at the Pointer P in the VK*

DV = VK<P>

b. *Mod 16 the Pointer position digit's value with the value at the OK value's position to the right of the Pointer and also add the CycleCounter's current value to the position*

PDAF(1) = (DV + VK<OK<P> + CC>) Mod 16

g. *If the returned PDAF key is to be all the cycles, then exit when the CycleCounter is equal to the length of the number (and HoldTemp and HoldFinal are not used). If the returned PDAF key is larger than that, then when the CycleCounter is greater than the number length, replace both the original sKey number (the VK), and also replace the OK number. The VK replacement is the last version of HoldFinal (which is the sum of all of the PDAF values thus far and the initial sKey); the OK replacement is the original (or last) OK value modulus 16 summed with the last version of HoldTemp. Then reset the Pointer to one, and the CycleCounter to 0, and begin again until either the VK and OK numbers are replaced in the same manner, or the return length is reached.*

If PDAF(Desired Key Stream Length) < all the cycles, then exit when reach DKSL
Else If PDAF(DKSL) = all cycles, then exit when CC = Length(sKey)
Else If PDAF(DKSL) > all cycles, then when CC = Length(sKey),

Replace the VK position with the cyclic summed value HoldFinal, also replace the OK position with the last version of HoldTemp summed with the original (or last) OK which results in new equal length values for VK and OK, and then set CC = 0, P = 1 and repeat if needed, exchanging the VK and OK in a like manner until reaching the PDAF(DKSL)

It should be noted that there are two different Increment Methods (IM) that can be used in step b). The one as written is IM1, in which the CC is added to the VK's position selection; and there is a second option IM2, where the CC is added to the OK's position selection. IM2 is written as:

$$PDAF(1) = (DV + VK<OK<P + CC>>) \text{ Mod } 16$$

FUNCTION: Combine

Registers :
R = {R[0], R[1], … ,R[n-1]}
K = {K[0], K[1], … ,K[n-1]}
A = {A[0], A[1], … ,A[n-1]}

Initial Value :
$i_{-1} = -1, \quad j_{-1} = -1$

Repeat for k from 0 to n-1 :
$i_k = (i_{k-1} + 1) + K[k] \pmod n$
$j_k = (j_{k-1} + 1) + R[k] \pmod n$
$A[k] = R[i_k] + K[j_k] \pmod B$

Example :
n = 10; B = 16
R = (0123456789)
K = (9876543210)
A = (2FA3EDA589)

1. *The combining function details: Combine* **R** *and* **K***, resulting in a n-bit 'alphabet',* **A**
   - 1.1 *Select an* **R** *digit by using the 1st digit of* **K** *as a pointer into* **R** *beginning at the 1st digit position and moving* **K***'s value in digit positions to the right in* **R** *where the starting position in* **R** *is the 0th value position.*
   - 1.2 *Select a* **K** *digit by using the 1st digit of* **R** *as a pointer into* **K** *beginning at the 1st digit position and moving* **R***'s value in digit positions to the right in* **K** *where the starting position in* **K** *is the 0th value position.*
   - 1.3 *Hexadecimal add without carry the selected* **R** *digit from Step 2.1 and the* **K** *digit from Step 2.2. This sum is the first digit of the result number,* **A**.
   - 1.4 *Repeat 1.1, 1.2 and 1.3 using the next digit to the right in* **R** *and* **K** *where the starting digits for the steps is one position to the right of the previously selected digit (the 0th value position). Continue until the result* **A** *is the same length as* **R** *and* **K** *(n-bits, 32 4-bit hex numbers for 128-bits).*

Example

R = *0123456789* **K** = *9876543210*

1.1: *9, using 9 from* **K** *and selecting 9 in* **R**
1.2: *9, using 0 from* **R** *and selecting 9 in* **K**
1.3: **A** *first digit is 2 from (9 + 9) Mod 16 = 2*

1.1: *8, using 8 from* **K** *and selecting 8 in* **R** *having started at the 1st position, which is the first digit position to the right of the previously selected last digit (9)*
1.2: *7, using 1 from* **R** *and selecting 7 in* **K** *having started at the 2nd position, which is the first digit position to the right of the previously select first digit (9)*
1.3: **A** *second digit is F from (8 + 7) Mod 16 = F*

A = *2FA3EDA589 from*
(9+9) Mod 16 = 2
(8+7) Mod 16 = F
(6+4) Mod 16 = A
(3+0) Mod 16 = 3
(9+5) Mod 16 = E
(4+9) Mod 16 = D
(8+2) Mod 16 = A
(1+4) Mod 16 = 5
(3+5) Mod 16 = 8
(4+5) Mod 16 = 9

FUNCTION: Extract

Registers :
$A = \{A[0], A[1], \ldots, A[n-1]\}$
$K_1 = \{K_1[0], K_1[1], \ldots, K_1[n-1]\}$
$W = \{W[0], W[1], \ldots, W[n-1]\}$

Initial Value :
$i_{-1} = -1$

Repeat for k from 0 to n-1 :
$i_k = (i_{k-1} + 1) + K_1[k] \pmod n$
$W[k] = A[i_k]$

Another optional child encrption key formulation:
$W[k] = A[ik] \oplus K1[k] \pmod B$

2. *The extraction function details: Extract n-bit key* **W** *out of* **A** *using* $\mathbf{K_1}$
    2.1. *Select an* **A** *digit by using the 1st digit of* $\mathbf{K_1}$ *as a pointer into* **A** *beginning at the 1st digit position and moving* $\mathbf{K_1}$*'s value in digit positions to the right in* **A** *where the starting position in* **A** *is the 0th value position.*
    2.2. *Use the selected* **A** *digit as the first digit of the result number,* **W**.
    2.3. *Repeat 2.1 and 2.2 using the next digit to the right in* $\mathbf{K_1}$ *and the starting digits in* **A** *as one position to the right of the previously selected digit (and this is the 0th value position). Continue until the result* **W** *is the same length as* $\mathbf{K_1}$ *and* **A** *(n-bits, 32 4-bit hex numbers for 128-bit).*

Example

A = *2FA3EDA589* $\mathbf{K_1}$ = *9876543210*

W = *98A39E8F3E*