Qwyit Authentication and Encryption Service (QwyitTalk™)
Security As A Service

Reference Guide

Version 2.3 March 14, 2018

Abstract

This paper provides a technical overview of a Qwyit™ application: operating a QwyitTalk™ *security service* – a direct replacement of Transport Layer Security (TLS). Applications would be able to simply place a small code segment within their communications protocol, resulting in authenticated and encrypted message traffic.

Contents

QwyitTalk™ Authentication and Encryption Service

This document outlines the QwyitTalk™ Authentication and Encryption *Service* (QwyitTalk™, QT™), a direct next generation replication and enhancement of the current, only global secure communications framework: Transport Layer Security (TLS). Qwyit provides the same features, benefits, authentication (embedded) and data security (stream cipher) for communications traffic using the Qwyit Directory Service (QDS) key store. QwyitTalk™ is an implementation capability based on the full Qwyit protocol as outlined in the current version of the *Qwyit Protocol Reference* document, available from Qwyit LLC. QT™ client demo APIs are available from Qwyit LLC; go to www.qwyit.com.

Approach

QwyitTalk™ easily allows any client/server or cloud-based application to add TLS-like authentication and encryption. The demonstration includes a QT™ Directory Server application that can be run on, or added to, any current communications server (web, file, comms, app, DB, etc.), as well as example client code for easy insertion into the communications protocol/processing of a connected device/app.

QwyitTalk™ Components

The following are required for QwyitTalk™:

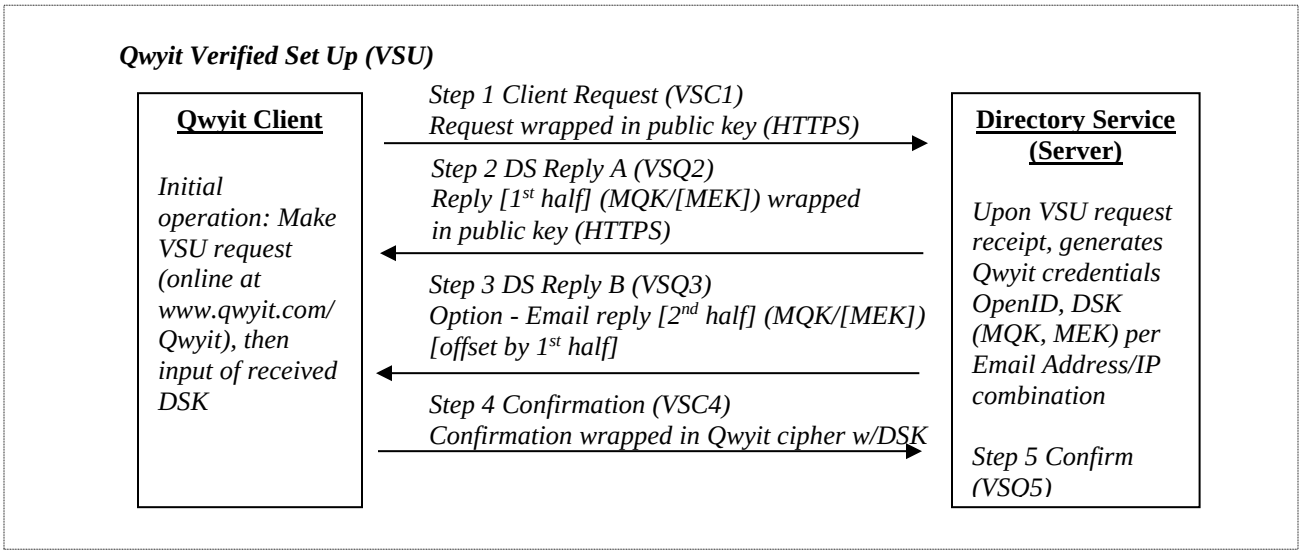
1. QT™ Server (operated by Qwyit LLC at www.qwyit.com/Qwyit)
2. QT™-compatible client application (client w/QwyitTalk™ SDK embedded capability)

QwyitTalk™ Core Required Processes

The following are the core required QwyitTalk™ processes (mirroring TLS):

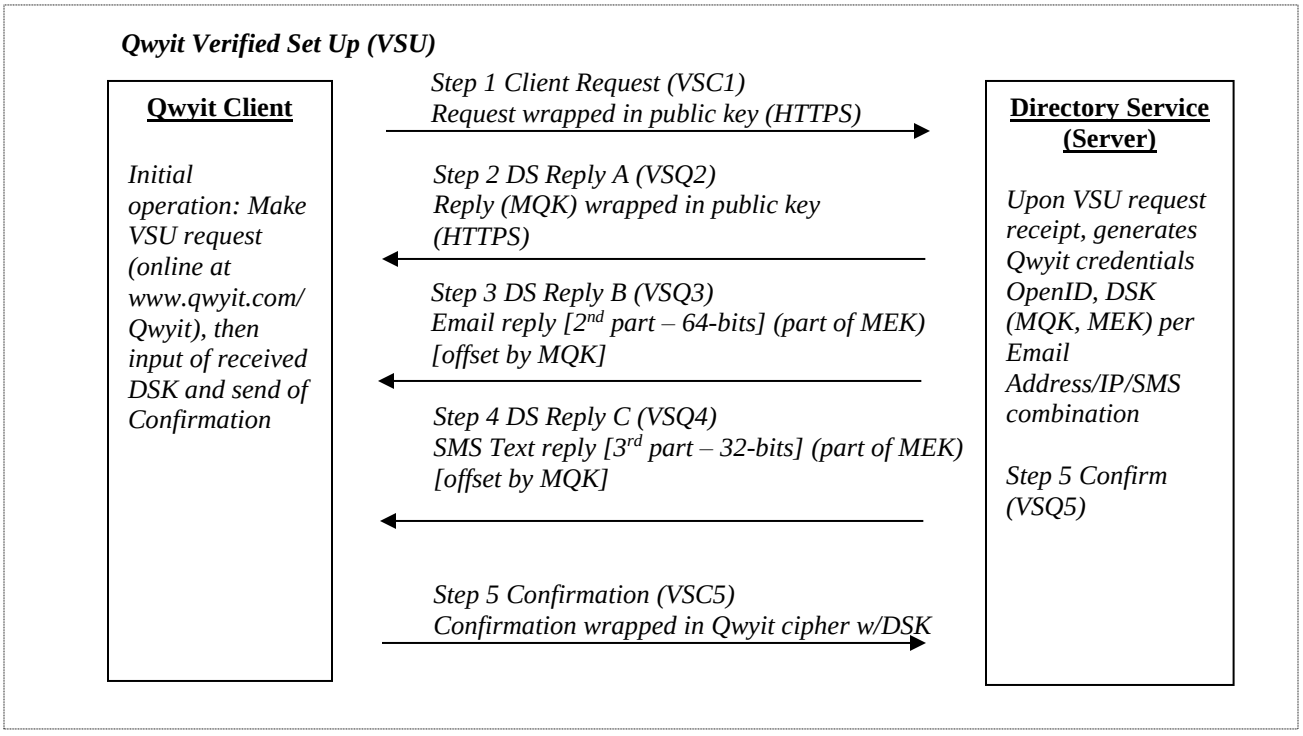
Initial Authentication Token/Credential Distribution (Verified Setup – VSU)

- Initial QwyitTalk™ client authentication token distribution is accomplished through a Verified Setup (VSU) with the QwyitTalk™ Directory Server (QDS). Token is a 512-bit 2-part key, with a public identifier: OpenID [up to 64-bits], MQK [Master QwyitTalk™ Key, 256-bits], MEK [Master Exchange Key, 256-bits])
    - This is identical to certificate obtainment and installation in TLS
        - The main enhancement is that *all QwyitTalk™ participants will perform a VSU*
        - Only Servers install certificates in TLS, as the Clients almost universally do *not*
        - Not only does this universally authenticate all participants, it removes the end-user confusion/required ‘expert-ism’ of multi-layered, multi-point Username/Password complexity



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 only uses a single band wrapped in a public key – *QwyitTalk™ recommends a minimum of 2 bands, as shown above.* One can imagine *QwyitTalk™Plus* (*Q+*) using 3 bands (adding SMS texting), or *QwyitTalk™Max* (*QMax*) using 4 bands (adding a phone call), or even *QwyitTalk™Platinum* (*QP*) using 5 bands (adding a paper delivery (or two!)) all sending either MQK/MEK portions or simple PDAF or OWC offsets.

The system outlined below is a Q+ implementation, using 3 bands. *The security of these different implementations can either be reliant on an initial HTTPS/TLS distribution (using QwyitTalk™ primitive offsets (PDAF/OWC/MOD16) in the other bands as shown below), or increasing security by implementing actions within those bands (calling/texting for offsets, etc.)*



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

NOTE: Following each step description, if there is a message sent, the format is shown in brackets:
[MessageTitleID: parameter1, parameter2, … ended with a QT™ termination character (^)]

NOTE: QT™ is a direct replacement of TLS: The VSU is Certificate obtainment and installation, the AH is the TLS handshake for session key creation and agreement, and QT is the chosen cipher. The AH and QT are orders of magnitude faster (less round trips, less computation), yet provide the exact same benefit/properties: Perfect Forward Secrecy (PFS), authentication and encryption. QT™ also replaces ***all*** of the TLS extensions, providing the same benefits, orders of magnitude simpler and faster (ALPN, SNI, etc.) as well as all of the Public Key Certificate Authority and certificate 'handling' (Revocation, OCSP, etc.).

NOTE: PFS can be implemented in several ways, and at various process points: all with the benefit of *not adding any process interruption, degradation or computational inefficiency. This is done without ANY communication (called NIL communication), where both parties update simultaneously after any single key use by performing a PDAF to a new agreed, authentic value without sending any messages.* Each participant pair (QT™/Client, Client/Client) performs a PDAF with the MQK and MEK after an AH, at QT session end (updating SQK, SEK), etc. at the designated interval; which can be static, or known-random (i.e., from a selection of the uniquely embedded key content – this would be client-app configured.) The PDAF has been shown to retain the initial random entropy of the key halves for at least 10^7 iterations; should a meaningful security limit be found, QT™ implementation would then recommend new key VSU deliveries at safe intervals. PFS key updates shown below. If desired, add an OWC.

Verified Setup (VSU)

- Client, initiate VSU within the application (and as required/desired) with the Directory Service/Server (*Step 1, VSC1*)
    - Goto www.Qwyit.com/Qwyit where an HTTPS session will begin. Enter the required information.
    - This generates a VSU start on the client application and the DS (VSC1)
    - Client app now awaits receipt and entry of the key

- Directory Service/Server reply to client (*Steps 2,3, and 4 – VSQ2, VSQ3, VSQ4*)
    - The client has submitted a VSU on the webpage, and is waiting for the reply
        - Generate OpenID (recommend 16 digit, 64-bit unique IDs), DSK for client
            - Generate a random, unique 16-hex digit, 64-bit OpenID for this client (per email/SMS/IP (if))
            - Generate random 352-bits that includes three (3) parts
                - Master QwyitTalk™ Key (MQK, 64 hex digits, 256-bits)
                - Email Offset Key (EOK, 16 hex digits, 64-bits)
                - SMS Offset Key (SOK, 8 hex digits, 32-bits)
        - Perform a PDAF(EOK, SOK) generating 8 rounds (cycling through each round of SOK moving the start 1 digit to the right)
            - Result is the Master Offset Key (MOK, 64 hex digits, 256-bits)
        - Perform a PDAF(MQK, MOK) generating 1 round
            - Result is a the Master Exchange Key (MEK, second 64 hex digits, 256-bits) – store with MQK as this new client's QwyitTalk™ keys
        - Encrypt EOK and SOK using MQK
            - Concatenate EOK and SOK
            - MOD16 add the first 96-bits (24 hex digits) of MQK with the concatenated EOK and SOK
            - Separate the encrypted EOKe and SOKe and ready to send in email and SMS respectively
        - Reply (VSQ2) with OpenID, (MQK) during the HTTPS session on the webpage
        - Reply Step 3 to email address, sending OpenID and EOKe
        - Reply Step 4 to SMS number, sending OpenID and SOKe

  [VSQ2: OpenID, MQK] [VSQ3: OpenID, EOKe in email] [VSQ4: OpenID, SOKe in SMS]

- Client decrypt of reply and confirmation (*Step 5, VSC5*)
    - Client will cut & paste the session-shown OpenID, EOKe and SOKe into their application, and OpenID and MQK will appear in the app/window, as the HTTPS session has received those
        - Open email message to reveal OpenID and EOKe; cut & paste (or type) into application (both values)

- Open SMS text message to reveal OpenID and SOKe; cut and paste (or type) into application (both values)

- Click button to "Store Key" (or some relevant, pertinent UI text)
  - Applet will check that all 3 OpenIDs match
  - Applet will decrypt EOKe and SOKe, and create MEK
    - Concatenate EOKe and SOKe
    - MOD16D the first 96-bits (24 hex digits) of MQK with the concatenated EOKe and SOKe
      - Separate the decrypted EOK and SOK and ready for use to create MEK
    - Perform a PDAF(EOK, SOK) generating 8 rounds (cycling through each round of SOK moving the start 1 digit to the right)
      - Result is the Master Offset Key (MOK, 64 hex digits, 256-bits)
    - Perform a PDAF(MQK, MOK) generating 1 round
      - Result is a the Master Exchange Key (MEK, second 64 hex digits, 256-bits) – store with MQK as this new client's QwyitTalk™ keys
  - Concatenate MQK and MEK making complete 512-bit DSK
  - Insert into use (store DSK, OpenID in cookie, file, db – method?)
- Reply (VSC5) to DS with Confirmation message
  - Perform The Qwyit™ Cipher using DSK (MQK and MEK), generating message key (W)
  - Use message key to encrypt confirmation message, which is a 128-bit, 32 hex digit ID salt created by using the last 32 digits (128-bits) of the MQK in a PDAF with the last 32 digits (128-bits) of the MEK
    - Perform a PDAF(MQK last 128-bits, MEK last 128-bits); result is Confirmation Salt
    - Perform QwyitTalk™ encrypt using W and CS, result is CS encrypted (CSe)
  - Send (VSC5) output (OpenID, OR, CSe) to DS

[VSC5: OpenID, OR, CSe]

- DS decrypt of confirmation message (*Step 5, VSQ5*)
  - Perform The Qwyit™ Cipher using the DSK (found by sent OpenID) and reveal the message key (W)
    - Use message to decrypt confirmation (by creating the correct same ID salt and comparing)
      - Perform a PDAF(MQK last 128-bits, MEK last 128-bits); result is Confirmation Salt generated
      - Perform QwyitTalk™ decrypt using W and CSe, result is CS received decrypted (CS)
      - Compare CS received decrypted with CS generated
        - If doesn't match, error sent
        - If match confirmed, store IP Address, OpenID, DSK, email address, SMS (and whatever other required session ID was collected) into DS KDC – method?

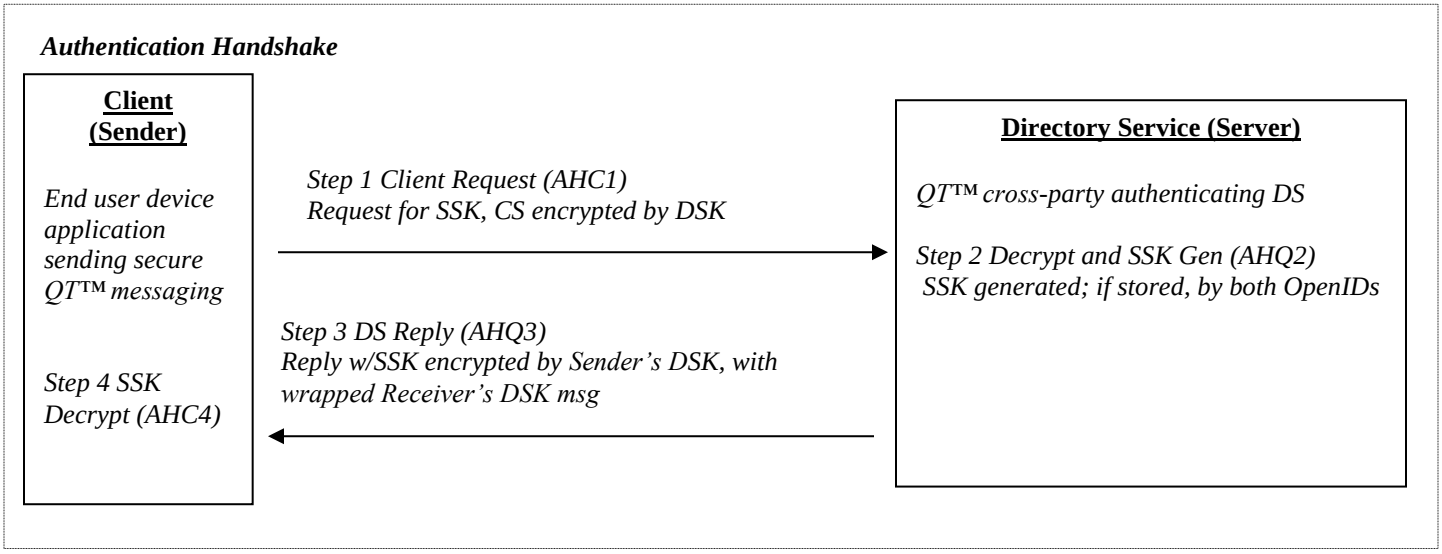
Per Message Token/Credential Distribution (Authentication Handshake – AH)

NOTE: The following is for a P2P architecture, with 1-1 communication. See Appendix C for other group key scenarios

- For each participant-pair messaging session, the sender will initiate a private real-time Authentication Handshake (AH) with a shared QT™ Directory Server in order to create a QT™ 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 QT™ Key, 64 hex digits, 256-bits], SEK [Session Exchange Key, 64 hex digits, 256-bits])
  - This is identical to the TLS handshake
    i. The main enhancement is that *QT™ requires only a single Round Trip*
       1. The RT is w/the QT™ server, not the destination, so security is enhanced
       2. *The initial contact w/the destination is 100% authentic and encrypted*
       3. QT™ AH includes a paired unique P2P SSK for that client pair (initiating and their destination). In this P2P, it's only sent to the initiating client who can decrypt their portion, but not their destination's portion
       4. 'Prepare' is an almost instantaneous Qwyit™ Cipher encrypt, not the complex TLS PKI processing

5. Because of this enhanced speed/timing, there is *no need* for any extensions for session resumptions, etc.
    a. And no trade-off for security, complexity, etc.

Architecture



Details

NOTE: The AH SSK is unique to each messaging pair of participants; after delivery, the DS destroys it. Should there be a requirement to retain SSKs, the QDS will need to be set to do so. 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 would need to be captured in order to be read (requirement to outside agency).

Authentication Handshake (AH)

- Client Sender sends Authorization Request to Directory Service/Server (*Step 1, AHC1*)
    - Send Authorization Request (AUTR) to DS in order to receive an SSK for communication w/intended Client
        - Create Confirmation Salt (64 hex digits, 256-bits)
            - Any system defined CS as per the Sender/Receiver IDs 'marketplace' (app, industry, etc.)
            - Such as: Using the 1st two digits of the OR (rounded up, even) as the length of bits to pull from the MQK (front/back determined by the 3rd digit of the OR in sum w/the 3rd digit of the MEK), used in a PDAF upon itself) – or any creation/summation that fits the performance requirements, as well as somewhat non-static (even though this isn't necessary – the same CS can be used repeatedly, as it is uniquely encrypted)
        - Send AHC1:SenderOpenID, ReceiverOpenID, CS encrypted in The Qwyit™ Cipher using DSK to DS

[AHC1: SenderOpenID, ReceiverOpenID, OR, CT] where CT is the encrypted CS

NOTE: While ReceiverOpenID may be considered 'private' (you may not want anyone to know who you are going to contact; although in TLS this is fundamentally known); digitally, it is impossible to 'hide' your next connection, so it will be known. It is certainly possible to include the ROpID in the encrypted CT; but it leaks information about the message keys, unless twice-encrypted – and all of that work is for naught. *No one has the SSK but you and your intended recipient*, so your communications will be secure.

- Directory Service/Server generates SSK (*Step 2, AHQ2*)
    - Decrypt AHC1 from Client using respective OpenID and DSK in the Qwyit™ Cipher
        - Check that Sender client has submitted a correct, meaningful CS (integrity, authentication)
            - Message integrity is inherent within the encrypted CS – no other needed
        - Check that Receiver client exists
    - Create Session Start Key (SSK, 128 hex digits, 512-bits)
    - If to be stored, by SenderOpenID, ReceiverOpenID

- Directory Service/Server reply to Client (*Step 3, AHQ3*)
    - Send SSK Accept (AHQ3) to Sender, with wrapped Receiver Clients' msg
        - The AHQ3 is sent to the sending Client
            - If Receiver Client does not exist in DS:
                - If this DS is part of a federated group, check w/other DS members as per the QT™ communication key exchange framework of the group
                - Notify Sending Client of situation (if desired/required – no Receiver client)
        - The AHQ3 includes 2 different CTs
            - The SSK is encrypted using the Sender Client's DSK and the OR
            - The SSK is twice-encrypted using the Receiver Client's DSK and the same OR
                - As per the QT Cipher, a 2nd Msg Key is created and used again on the 1st encryption's CT (this is to obscure the Msg Key chain from the Sender who is aware of the PT and could derive the 1st Msg Key)

  [AHQ3: OpenID, OR, CT1, CT2] where OpenID is the Sender Clients, CT1 is the SSK-encrypted ciphertext using the Sender's DSK and CT2 is the twice-encrypted SSK ciphertext using the Receiver's DSK

  NOTE: The integrity of the return message is such that any deviation in either CT will be detected by either client in their inability to communicate (as their SSKs won't match) – or if amazingly, a bit is flipped in both CTs such that the SSK is changed 'correctly' for both participants – great! An outsider has no knowledge to meaningfully do this and gains nothing.

- Sender Client decrypts SSK (*Step 4, AHC4*)
    - Sender Client decrypts AHQ3 from DS using DSK in the Qwyit™ Cipher
        - Client only replies to DS if unable to decrypt key (for whatever reason), sending an AHE for error
        - Sender and Receiver Clients commence QwyitTalk messaging with Sender initiating contact with QwyitTalk™ Start message (including OR, CT2 for Receiver to derive SSK) in Normal Operation

- Directory Service/Server and Sender Client perform a PDAF PFS NIL communication DSK key update
    - Perform a PDAF(MQK 256-bits, MEK 256-bits) for two rounds, creating 512-bit result w/MQK, MEK halves
        - Optionally, extend the above PDAF for 2 rounds (doubling length) and perform an OWC

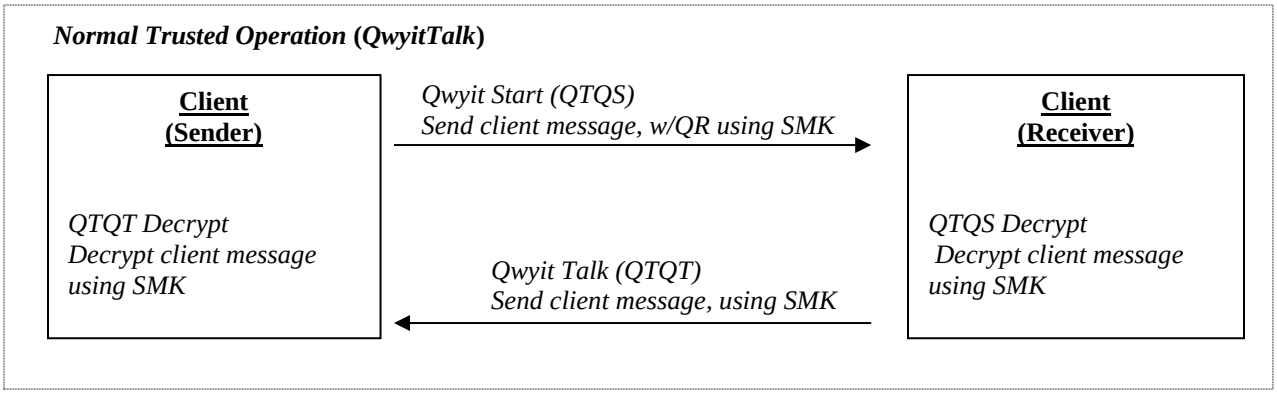
NOTE: Keeping the keys in synch between the QDS and the Client needs careful attention. Each different QwyitTalk™ network may require unique handling.

Messaging – The Qwyit™ Cipher

- Normal Trusted Operation, each participant using the SMK in secure Qwyit™ Cipher exchanges – this is QT™ between the two clients, securely using their own generated session keys
    - This is identical to the TLS session key creation, cipher selection and initial secure use
        i. The main enhancement is that *QT™* uses only its own cipher – the world's fastest and most secure (by an order of magnitude)

(The SMK (SQK, SEK) must be handled and stored securely, only for this particular session.)

Architecture



Details

Normal Trusted Operation

- Sending Client sends QwyitTalk™ Start to Client Receiver (*QwyitTalk™ Start, QTQS*)
  - Sending Client generates a QT™ 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, the Initial Message opening communication)
    - Perform Normal Trusted Operation using The Qwyit™ Cipher on IMSG with SMK (SQK and SEK)
  - Send QwyitTalk™ Start (QTQS) message
    - Send QTQS:OpenID, QR, OR1, OR2, CT2, Ciphertext of IMSG
      - Where OR2 is the AHQ3 OR, and CT2 is the twice encrypted SSK

  [QTQS: OpenID, QR, OR1, OR2, CT2, CT]

- Receiving Client decrypts QTQS (*QT™ Talk, QTQS*)
  - Receiver Client, decrypts in two steps:
    - 1st decryption: using OR2 and their DSK to derive the SSK – this requires two Msg Key decryptions as noted in the AH above
      - Now both clients have a shared SSK for this session
    - 2nd decryption: using the just-derived AH SSK to determine the SMK from the QS message by performing the same MOD16(SSK,QR)
  - Commences Normal Trusted Operation using SMK in The Qwyit™ Cipher decryption of OR1, CT of IMSG

- Clients continue communication by performing Normal Trusted Operation messaging w/Qywit™ Cipher using SMK (*QwyitTalk, QTQT Encrypt/Decrypt*)
  - Send QT™ Qwyit Talk (QTQT) using SMK (SQK, SEK) in The Qwyit™ Cipher on message

  [QTQT: OpenID, OR, CT] where CT is the Ciphertext of message contents
  - QTQT Decrypt w/SMK (SQK, SEK) in The Qwyit™ Cipher to reveal secure message contents
    - Receive and decrypt QTQT in The Qwyit™ Cipher using OR, to reveal Plaintext message contents

- At known, system defined end of SMK key life reached (# of messages, time, etc.), both Clients perform a PDAF PFS NIL communicaiton SMK key update
  - Perform a PDAF(SQK 256-bits, SEK 256-bits) for two rounds, creating 512-bit SMK result w/SQK, SEK halves
    - Optionally, extend the above PDAF for 2 rounds (doubling length) and perform an OWC

Error Messaging (DSE)

- Accompany all of the above w/appropriate error trapping and notifications

[DSE: MessageID, appropriate contents]

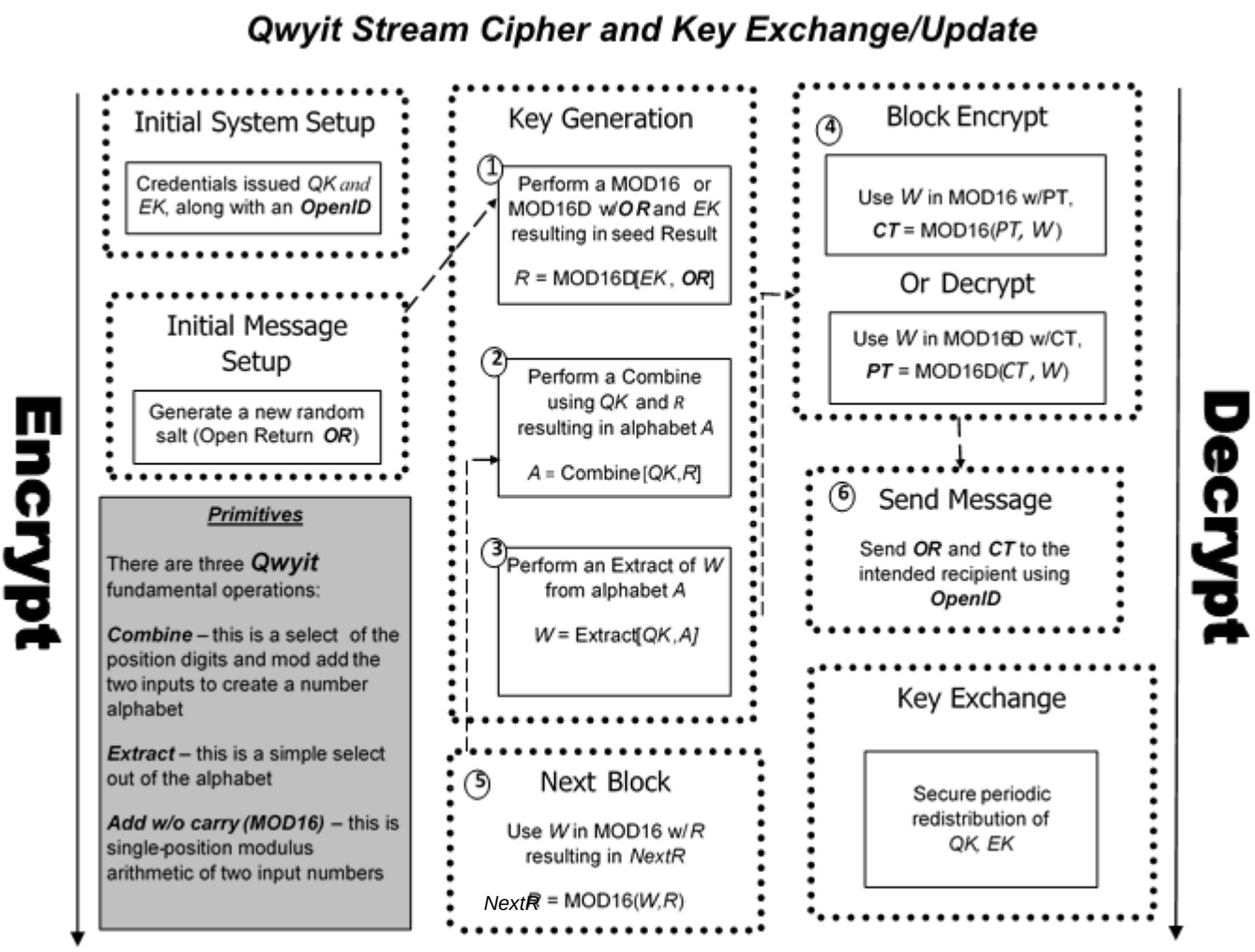
QwyitTalk™ Optional Processes

The following are optional system-defined QwyitTalk™ processes:

Client Application Key Storage (CKS)

- Should there be a requirement/desire to tie the local version of the VSU received DSK to the individual user, as opposed to just the device (or if want a simple, effective encrypted storage technique in addition to the many already available to the client application through the OS, etc.), it is recommended that an n-digit hex PIN (security based on the number of digits) be used to encrypt DSK local storage with simple MOD16 addition, accompanied by limiting the number of wrong use attempts locally, after notification from the DS of incorrect key use.

Appendix A – QwyitTalk™ Stream Cipher



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

- Perform MOD16 using EK and OR, resulting in R [256-bits]
- Perform COMBINE using QK and R, resulting in alphabet A [256-bits]
- Perform EXRACT using QK and A, resulting in message key W [256-bits]
- Perform MOD16D using 256-bits of Ciphertext (C) and W, resulting in Plaintext (P) [or CT XOR W = PT]
- 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

Appendix B – QwyitTalk™ Security Notes

Some QT™ (QT) security aspects:

- QT keys should be securely stored (either as noted above, or using 'best practice' techniques)

- Knowledge of QT keys cannot be gained from the system in action; as all messages are inherently incorruptible (recipients instantly recognize message corruption) and their content is all numeric key values
    - The only exception are the P2P messages where the upper level SMK is one-way gate protected from the system-derived SSK, as well as one-way gate protected from the individual child message keys used in the Qwyit™ cipher on the content

- Active stolen/lost key use is invalid on any previous messages (PFS provided by NIL communication key updates'), and implementing PIN storage tying user to keys, along w/system lockouts and 'best practice' Alert systems will thwart any active use
    - Any particular implementations will be determined by each client-connected app

- The inactive 'Listener' problem still exists: someone steals your QT keys, and doesn't use them actively, they just listen and capture all of your messaging. In TLS, this is akin to someone stealing the Private Key, which is actually another reason QT™ is superior: in TLS, this enables reading *every message by every participant relying on that Trust node!* In QT™, this is just you. And while short PINs are effective in thwarting active use, they are of no value against The Listener. This insidious attack is easily thwarted by injecting a unique P2P key obtained/shared *outside* of QT™. This key would be used in the message content encryption step during a *QTQS* creation. If this type of attack is deemed substantial, it should/will be added as an input value in those QT™ client-connected apps that require it

- QT keys are unimportant; i.e., they are not to be used (as designed in the public QT system) as a system-specific verifiable authentication/trust instrument
    - Qwyit LLC is of the opinion (as yet unrefuted!) that there are *no* global communication systems in which one is allowed to join w/o first being authenticated. As such, wherever QT is enabled through client apps, those clients will eventually, most notably at the point of any commerce exchange, be authenticated within the system holding their value. Phone systems bill their users, eCommerce platforms exchange goods for credit through banks requiring their authentic token (credit card, etc.), etc.
    - *QT keys are authentic to the requestor, authentic throughout their use in the QT message hierarchy, and authentic in P2P and group client communications*
    - A QT system (Public and/or Private) can certainly be built in which the QT key *is* the network authentication token; but the public system does not offer that accountability

- Order of magnitude efficiency – QT messaging hierarchy and exchanges can perform within real-world margins for normal communications. I.e., QT secure messages won't take any longer than insecure messages within their communications network. No other security system in the world can make, and validate, this claim (See 2015 NIST submission paper)
    - This is the only reason existing security techniques have not proliferated where they should – they simply require too much 'system energy' (computing power, time, number of messages, message lengths and content, etc.)

  - This property is the only way any security system will ever allow full global proliferation
  - In hardware implementations, QT can be configured (gates sizes, etc.) to operate within the performance characteristics of open-system data handling
    - QT-on-a-chip can be modeled to perform its simple register-based Mods at almost the same speed as moving open data through a chip – certainly within the specs of any communication standard for open data

- Security based on mathematic principle, not theory
  - QT is an unsolvable, underdetermined system of equations – in any single instance and in total of all messaging
  - Initial key distribution security is based on HTTPS and multi-band delivery
    - Certainly customizable to meet any perceived need/improvement
    - Keys are 'non-critical', therefore most likely HTTPS is sufficient for initial distribution

- QT provides all of the same security properties of TLS (privacy, reliability, authentication, data security, Perfect Forward Secrecy (PFS), etc.)

- QT keys
  - Sizes can be system-specifically modelled
    - QT primitives are based on *any* key length of even-numbered bits, with minimal extension of 1 byte (as time passes and computing power increases, QT keys can simply expand to meet the computational security requirements)
  - Control hierarchies can be included
    - QT keys can actually include Access Control within them, not effecting computation
  - Allow for simple, secure storage techniques (PIN-related offsets, etc. for man-to-machine connectivity w/o need for difficult passphrase implementations – see above, and other Qwyit™ documentation for details)
  - Limit bandwidth requirements (storage, transmission, etc.) to well-defined, easily implemented definitions within any communication specification requirements standards

Appendix C – QwyitTalk™ Group Messaging

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

- For various group communication scenarios, the sender will initiate a private real-time Authentication Handshake (AH) with a shared QwyitTalk™ Directory Server in order to create a QwyitTalk™ VPN tunnel with multiple intended receivers (based on their OpenIDs).
- Changes to the above already defined AH steps, as demonstrated below, can be put in place
- The difference is in the reply to the requestor (Client Sender), and any intended Client Receivers
    - The basic principle of group messaging, along w/maintaining the division between the QDS's SSK and the now-group SMK, requires a distribution QTQS send to all members of the group
        i. Any and all group messaging scenarios (timing, add/drop members, etc.) will be handled by the client application connecting to and using the public QT™ QDS. Including, but not limited to, storing the SSK locally, SMK (through QR distribution) management, etc. If there is a continual need for group messaging, a local, private QwyitTalk™ system, in which group-member management functionality can be integrated into the QDS, should be considered
        ii. The Public QT™ QDS is envisioned to manage universal connectivity through either the above P2P AHQ3 encapsulated reply, or the below, multi-client AHQ3 reply (these additional group member replies could also be stored in the QDS and retrieved by the Clients, as managed by the client application – this will require a new universal definition of that type of *request* by the group members, TBD in a follow-on Reference)

NOTE: The special kind of 'group' messaging of "1-1 simultaneously" (for a large website, such as Amazon, etc.) requires a simple 'reversal' of the P2P messaging architecture already presented. Instead of the originating Client1 beginning the AH with the QT™ QDS, Client1 sends a communication request to the web destination, openly:

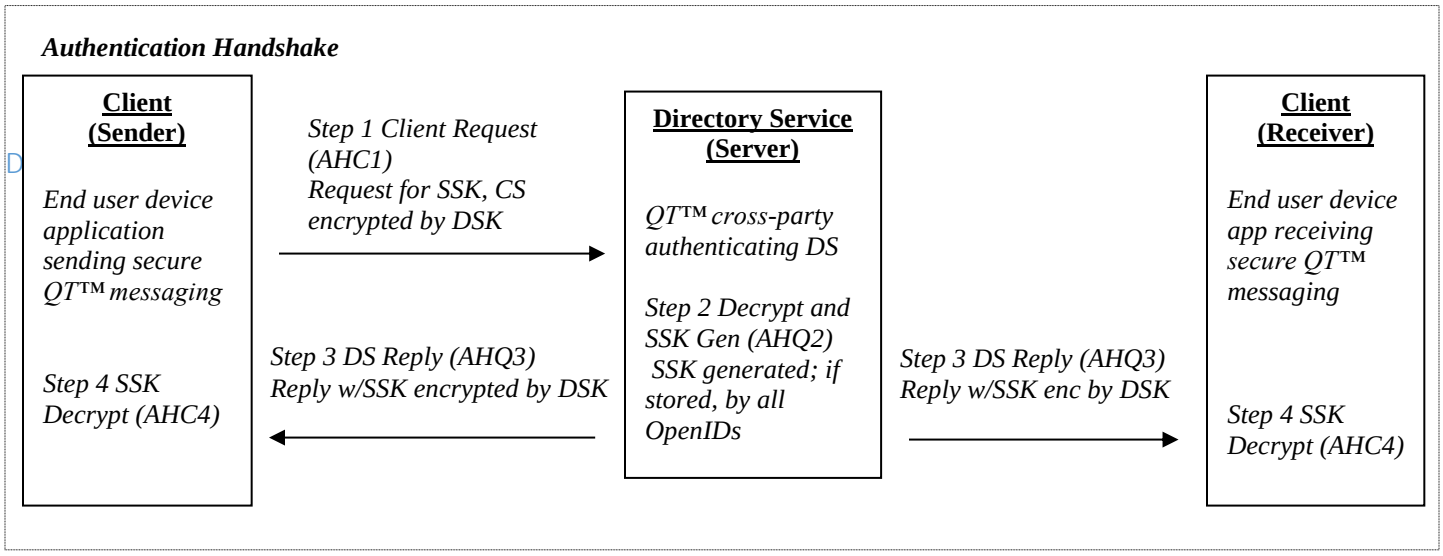
- Client Sender sends Authorization Request to Website (*Step 0, AHC0*)
    - Send Authorization Request to Website in order to receive an SSK for communication
        - Send AHC0:SenderOpenID, ReceiverOpenID to Website

[AHC0: SenderOpenID, ReceiverOpenID] where ReceiverOpenID is the general, OpenID of the site

Now the website performs a complete AH with the QT™ QDS, using the webserver's OpenID/DSK relationship. This allows any large website to manage QT™ QDS connectivity in concert w/their method's used to manage connectivity traffic across multiple servers using their single IP address. I.e., each webserver can have a unique OpenID/DSK (recommended), or they can store a single DSK and share/manage it to each server/connection (not recommended), etc. The webserver then sends an AHC1 to the QT™ QDS, asking to talk P2P securely w/Client1 using their OpenID. The AHQ3 reply will contain the wrapped SSK-encrypted for that Client1, as well as returning the SSK-encrypted w/that unique server's DSK relationship. Then the QTQS from webserver to Client1 creates a unique SMK from the shared SSK; and then the remainder of Client1's web session continues QT™ secure messaging. This 'reversal' would be made 'standard' in applications that exclusively require this type of connectivity – web browsers and servers, etc.

The only possible change in the Public QT™ QDS, as well as managed by these types of applications, is the DSK PDAF key update cycle and timing. It may well be best to have single-stream AH processing per server in order to manage updates sequentially for each use; or some other suggested/managed solution.

Architecture



Details

As above, except as noted in the AHQ1 request, and AHQ3 reply (now multiplied):

- Client Sender sends Authorization Request to Directory Service/Server (*Step 1, AHC1*)
    - As above, except there is a list of ReceiverOpenIDs
        - Send AHC1:SenderOpenID, Receiver1OpenID, Receiver2OpenID…, CS encrypted in The Qwyit™ Cipher using DSK to DS

  [AHC1: SenderOpenID, Receiver1OpenID, Receiver2OpenID,…OR, CT] where CT is the encrypted CS

  NOTE: The QT™ QDS will be prepared to test each input, recognizing that the 3rd [and any succeeding] are shorter than the OR, and will know this is a group-request. And will act accordingly for Step 3

- Directory Service/Server reply to Clients (*Step 3, AHQ3*)
    - Send SSK Accept (AHQ3) to Sender and Receiver Clients
        - Send AHQ3:OpenID, SSK encrypted in QwyitTalk™ Cipher using DSK to Clients
        - The AHQ3 is sent to the sending and all receiving Clients, and DS is authentic if using the correct DSK
            - If Receiver Client does not exist in DS:
                - If this DS is part of a federated group, check w/other DS members as per the QT™ communication key exchange framework of the group
                - Notify Sending Client of situation (if desired/required – no Receiver client)

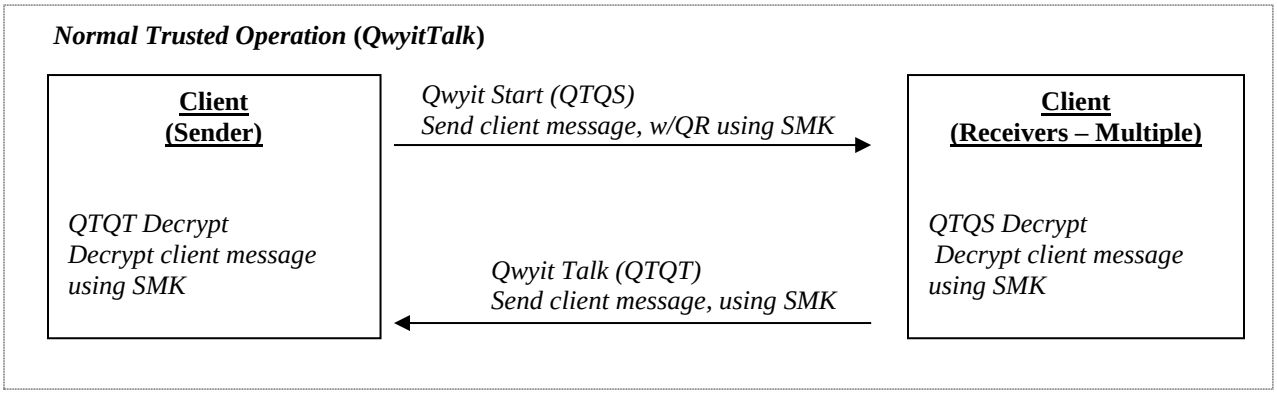
  [AHQ3: OpenID, OR, CT] where OpenID is the Sender and the Receivers in their respective messages, and CT is the ciphertext of the SSK

Messaging - QwyitTalk

- Group messaging requires the Client to send QTQS to each member in their list (that was sent in the AHC1)
    - Each receiving member of the group will recognize the QTQS as a group send

(The SMK (SQK, SEK) must be handled and stored securely, only for this particular session.)

Architecture



Details

As above, except as noted in QTQS, and its receipt by group members:

Normal Trusted Operation

- Sending Client sends QwyitTalk™ Start to Client Receivers (*QwyitTalk™ Start, QTQS*)
  - Send QwyitTalk™ Start (QTQS) message
    - Send QTQS:OpenID, QR, OR, Optionally: Ciphertext of IMSG

  [QTQS: OpenID, QR, OR, (CT)] where CT is an optional ciphertext of a PT intended for the group – if no such message is intended/necessary, there is no CT included]

  NOTE: The receiving clients (managed by their client app), will recognize a QTQS message with only 3 or 4 data elements as being a group QTQS, as opposed to the 6 elements of a P2P QTQS, and will handle accordingly – performing only a single decrypt of any optional CT – they will all create the SMK from the QR

- Receiving Client decrypts QTQS (*QwyitTalk™ Talk, QTQS*)
  - Receiver Client, using the AH SSK received from the QDS for this communication
    - Determines SMK from the QS message by performing the same MOD16(SSK,QR)
    - Decrypts any optionally sent CT using the just created SMK
  - Commences Normal Trusted Operation message Receive decryption using SMK in QwyitTalk™ Cipher on OR, Ciphertext – from/to any members of the group

- Management of the SMK key life will be a consideration of the Client Application performing group messaging – updates may be performed in the normal QT™ manner (PDAF PFS NIL communication SMK key updates)

Appendix D – QwyitTalk™ VSU, AH and Messaging Example

The following are an example of the process flow for the complete QT™ system. This output is created by the *QDS-Full-Example.exe* application available from Qwyit LLC at www.qwyit.com.

QT™ VSU Example

The VSU involves any Client who wants to join the QT™ secure messaging system, contacting the Qwyit Directory Server (QDS). This process is most likely found, performed and managed within a Client Application participating in the QT™ community. The Client and QDS process steps, and output, are as follows:

Client1

I just contacted the Qwyit.com QDS to begin a VSU and join the Qwyit Community!

QDS

I just received a VSU start from Client1 (who submitted a Qwyit.com website form)
First, I'll create a new ID for this Client1
The Client1 ID is: 71A2913C34ED7413
Click Continue in Client1...

Next, I'll create some new keys for this Client1
The Client1 MQK is: 7F3FB0566CA68F6264E8D990DE5CE9399DEFAE32395E1D13C82FEDEE35C4FFF9
The Client1 EOK is: C0BCEBE8CB26B8FA
The Client1 SOK is: B7D4DD28
Click Continue in Client1...

Next, I'll form the MOK and MEK using the PDAF on the EOK and MQK for this Client1
The Client1 MOK is: 7B789794ABE2DEF642B7A90876D613A5B66EC643A74160BC6B729994498237D0
The Client1 MEK is: DED71FBEE58CDD9BF1146299678A16C7A0712F10218B0ED2A7EDD65D24B36F70
Click Continue in Client1...

Now, I'll ready the keys for securely sending to this Client1
The Client1 EOK || SOK is: C0BCEBE8CB26B8FAB7D4DD28
The Client1 EOK || SOK encrypted is: 3FEB9B3E27CC375C1BBCA6B8
Now, The keys are secured and ready to send in email and SMS to this Client1
The Client1 EOK encrypted (EOKe) is: 3FEB9B3E27CC375C

The Client1 SOK encrypted (SOKe) is: 1BBCA6B8
Click Continue in Client1...

Lastly, The keys are sent in HTTPS (MQK), email (EOKe) and SMS (SOKe) to this Client1
[VSQ2: 71A2913C34ED7413, 7F3FB0566CA68F6264E8D990DE5CE9399DEFAE32395E1D13C82FEDEE35C4FFF9] sent in HTTPS to this Client1
[VSQ3: 71A2913C34ED7413, 3FEB9B3E27CC375C] sent in email to this Client1
[VSQ4: 71A2913C34ED7413, 1BBCA6B8] sent in SMS to this Client1
VSU processing now switches to Client1
Click Continue in Client1...

Client1

VSU processing now continues here in Client1
First, all three values from the HTTPS browser window, an email and a text msg SMS will be cut & pasted into the Client1 Qwyit application, and stored
The Client1 EOKe || SOKe is: 3FEB9B3E27CC375C1BBCA6B8
The Client1 EOK || SOK decrypted is: C0BCEBE8CB26B8FAB7D4DD28
Click Continue in Client1...

The Client1 MOK is: 7B789794ABE2DEF642B7A90876D613A5B66EC643A74160BC6B729994498237D0
The Client1 MEK is: DED71FBEE58CDD9BF1146299678A16C7A0712F10218B0ED2A7EDD65D24B36F70
Now all the master Client1 keys (MQK, MEK) have been decrypted and are stored in Client1
Click Continue in Client1...

Next, reply with VSC5 message to QDS using the new keys
First step is to create the Confirmation Salt using the PDAF and the last 128-bits of the MQK and MEK
The Client1 Confirmation Salt is: 7B3DCB65174CEC050DFCCCDC74FDEC22
Click Continue in Client1...

Next, use the QwyitTalk cipher to send the Confirmation Salt in an encrypted VSC5 message back to the QDS
The Client1 VAC5 message key is: 3339537D8F09106BCC4B5DCC1B3DCAF8A0A1977099FA4507DA6FD79E19370835
The Client1 VAC5 ciphertext is: j{†±ÂKF• ýr• Ppú-ÐåÝ³Ü=‰J“Ò^z:g
Click Continue in Client1...

Now send the encrypted VSC5 confirmation message back to the QDS
[VSC5: 71A2913C34ED7413, DBADC51DEC597FCB536F68315F4D63AFE7A169939B7F0849A5ECE330D77382B2, j{†±ÂKF• ýr• Ppú-ÐåÝ³Ü=‰J“Ò^z:g] sent using Port 4180 to the QDS
VSU confirmation processing now switches to the QDS
Click Continue in Client1...

QDS

VSU processing now continues here in QDS
First step is to create the Confirmation Salt using the PDAF and the last 128-bits of the MQK and MEK
The Client1 Confirmation Salt is: 7B3DCB65174CEC050DFCCCDC74FDEC22
Click Continue in Client1...

Next, use the QwyitTalk cipher to decrypt the received Confirmation Salt from the VSC5 message
The Client1 VAC5 received message key is: 3339537D8F09106BCC4B5DCC1B3DCAF8A0A1977099FA4507DA6FD79E19370835
The Client1 VAC5 received plaintext, which is the Confirmation Salt, is: 7B3DCB65174CEC050DFCCCDC74FDEC22
Click Continue in Client1...

Now compare the received with the computed...
As you can see, they are the same. Client1 infor is stored here in the QDS, and is now a Qwyit Community member!
VSU PROCESSING COMPLETE!
Select CLEAR for next demonstration

Client1

VSU PROCESSING COMPLETE!
Select CLEAR for next demonstration

QT™ AH Example

The AH involves any Client who wants to securely message with any other QT™ secure messaging system Client(s). In this example, the 'app' is configured to perform P2P secure messaging where Client1 contacts the Qwyit Directory Server (QDS) and receives an encrypted SSK, as well as a wrapped encrypted SSK for their intended recipient, Client2. This process is performed without any end-user participation, simply as the start of their messaging to Client2. The Client1, QDS and Client2 process steps, and output, are as follows:

Client1

I am going to message Client2. First, I need to perform an AH with the QDS in order to get a shared start key for use with Client2
Here is my current Client1 MQK : 554E3EE1B25DC81611E3A3FE76EF63D561BA19B1489901FCD07B1899B03C44F0
Here is my current Client1 MEK : A6DB598412DDF19E16160728B19E2788BDB88EFE7B05A0DCE88B6651E6220FFF

First, I'll calc my CS to associate with this AH
Click Continue in Client1...

The CS associated with this AH is 27989956E84BC7EB
Now, I'll send the AHC1 to the QDS with the OpenID of Client2
I need to get an OpenReturn in order to encrypt my AHC1
OpenReturn = 55E51600B8EF22340D7C0FB277D1927D738796688747DD965421A9D5A255B858
The Client2 OpenID is A67241A3186F6CCD
Click Continue in Client1...

The AHC1 message key is A4395EB99CDEF81A81D07DD2CBBAD98C45C36998B98F8F66040FAB4B04EFACE5
The AHC1 ciphertext is 73030A010C7C770F7C7B7007050F7403
[AHC1: 0123456789ABCDEF, A67241A3186F6CCD,55E51600B8EF22340D7C0FB277D1927D738796688747DD965421A9D5A255B858, 73030A010C7C770F7C7B7007050F7403] sent using Port 4180 to the QDS
AHC1 processing now switches to the QDS
Click Continue in Client1...

QDS

AHC1 processing now continues here in QDS
First step is to decrypt the received AHC1 using QwyitTalk cipher and the keys associated with the sending Client, in this case, Client1

The Client1 AHC1 received message key is: A4395EB99CDEF81A81D07DD2CBBAD98C45C36998B98F8F66040FAB4B04EFACE5
The Client1 AHC1 received plaintext, which is the CS, is: 27989956E84BC7EB
Click Continue in Client1...

Check that the Calc CS equals the sent, decrypted CS (not done here, but would be)
Check that the Client2 exists as a Qwyit member
Keys retrieved and shown in Client 2...
When this is true, create a Session Start Key (SSK) for this pair of Qwyit members to communicate
The Client1-Client2 SSK is
934FFCFFE84B2A99068A16F5BF251F94349AC6A2887647A120CE0DEC5196DDE1F29A1EFB8C54309716F6641C9D91F3283BEEC9114E94EB7144C21FBEBEB73471
Click Continue in Client1...

Now, I'll send out the AHQ3 to just Client1 with Client2's wrapped SSK
I need to get an OpenReturn in order to encrypt the AHQ3 for Client1
OpenReturn = 4DA1A27A59137E39562917FFE4A677EADED42D0305FDDF86A8E2D58ACA74B03B
Click Continue in Client1...

Only AHQ3 is to Client1...
The AHQ3 Client1 message key is 4D94ADE3BB2B35D47B28F3ECB8DED39FD67373B964A3BF6F683FEEDB68B3ED1A
The AHQ3 first ciphertext is
0D770D7207070375077A060001747D0D07740A7977050376007E7670757500727702 0E727405030B0E0C760576717777040870037501010103097B0501007470727600757001037 17A01077600057D030674740E700774007B7C7D7402000B7E77747276740A7308027178070704017702 0C707474030607747D000476700670
The AHQ3 Client2 message key is AE088ECDB215E3727E058172A7096BC6DEE633999D141BEA7E437BD7869AA9DC
The AHQ3 second ciphertext is
7876047E7E060502070A057777720E0B07730874090771070371020C07047A0277717C777005780B017C060205750470057577760706017 40D070077057D0172077709790900050 67A71040176030E05067376030E0506717873090870717 10E77070073700A08080D0108007400727003717701060406727A737B76720D7372
[AHQ3: 0123456789ABCDEF, 4DA1A27A59137E39562917FFE4A677EADED42D0305FDDF86A8E2D58ACA74B03B,
0D770D7207070375077A060001747D0D07740A7977050376007E7670757500727702 0E727405030B0E0C760576717777040870037501010103097B0501007470727600757001037 17A01077600057D030674740E700774007B7C7D7402000B7E77747276740A7308027178070704017702 0C707474030607747D000476700670,7876047E7E060502070A057777720E0B07730874090771070371020C07047A0277717C777005780B017C060205750470057577760706017 40D070077057D0172077709790900050 67A71040176030E05067376030E0506717873090870717 10E77070073700A08080D0108007400727003717701060406727A737B76720D7372] sent using Port 4180 to Client1
AHQ3 processing now switches to Client1 and Client2
Click Continue in Client1...

Client1

Client1 now decrypts their section of the AHQ3 using QwyitTalk to get the SSK for their session
The Client1 AHQ3 received message key is: 4D94ADE3BB2B35D47B28F3ECB8DED39FD67373B964A3BF6F683FEEDB68B3ED1A
The Client1 AHQ3 received plaintext, which is the SSK, is:
934FFCFFE84B2A99068A16F5BF251F94349AC6A2887647A120CE0DEC5196DDE1F29A1EFB8C54309716F6641C9D91F3283BEEC9114E94EB7144C21FBEBEB73471
Click Continue in Client1...

After sending (the QDS), and receiving (Client1), both perform a PDAF PFS NIL communication DSK key update
This is accomplished using the MQK and MEK in a PDAF. Optionally, for a 1,024-bit length, and then performing an OWC to pare to 512-bits
Here is the PDAF result for Client1 new MQK/MEK keys:
2054844D0EC32E83F881D65485F8BCE96DAA2A4A10299030D33BDCD9C57B8C06175F0FF98ABBF97B471290DF0C78CE1D7E7B014A4190BD302CB05CDC652C3551DA2FFFF2733C99FCBFD988E9299F745E5189D26CB1A40EEB24B05785D4314A01627CBCC73843160740C6195883F0176E28B6E2B1F38A31FC146F089D0E81950EB1E14112C3B02B710750042FBB2E0B653C2714E45851C841BF791DED838095D36D789882138AD240F4488EC7FC6B7CD6626A897D

CB624C4100CE6DE87E7A822A6631411FC0227BE78EC99D49077CFCE4D9C1CCF5D4924230B5C96CD93E1F27933B2D4DD4C5A3676F76D4045F017F13C
11A3525F5DC04F9DAB5B9567EC16A7E2F
Here is OWC final result for Client1 new MQK key: 29C1EF0B7939D77734CE1B930E96124684F82682B39CCFAE521E5983EB19BE86
Here is OWC final result for Client1 new MEK key: C450C425656D73B26D841B630526109CEF111DD5D144164DB874948704B5D051
The AH is Complete! Now Client1 and 2 can message securely...
Lastly, the QDS performs the NIL communication key update...
Click Continue in Client1...

QDS

I perform the exact same PFS key updates for Client1

Here is the PDAF result for Client1 new MQK/MEK keys:
2054844D0EC32E83F881D65485F8BCE96DAA2A4A10299030D33BDCD9C57B8C06175F0FF98ABBF97B471290DF0C78CE1D7E7B014A4190BD302CB05CD
C652C3551DA2FFFF2733C99FCBFD988E9299F745E5189D26CB1A40EEB24B05785D4314A01627CBCC73843160740C6195883F0176E28B6E2B1F38A31F
C146F089D0E81950EB1E14112C3B02B710750042FBB2E0B653C2714E45851C841BF791DED838095D36D789882138AD240F4488EC7FC6B7CD6626A897D
CB624C4100CE6DE87E7A822A6631411FC0227BE78EC99D49077CFCE4D9C1CCF5D4924230B5C96CD93E1F27933B2D4DD4C5A3676F76D4045F017F13C
11A3525F5DC04F9DAB5B9567EC16A7E2F
Here is OWC final result for Client1 new MQK key: 29C1EF0B7939D77734CE1B930E96124684F82682B39CCFAE521E5983EB19BE86
Here is OWC final result for Client1 new MEK key: C450C425656D73B26D841B630526109CEF111DD5D144164DB874948704B5D051

The AH is Complete! Now Client1 and 2 can message securely and I am Finished!...
Click Continue in Client1...

Client2

These are for informational purposes – Client2 is not a participant in the routine AH
Here are the retrieved Client2 MQK : 46594234B75B9DAA00E0DD6A3563870ABCA7A02BEE5D767150EEB97CBF8363EF
Here are the retrieved Client2 MEK : 41B1D16C88CAE1010168B2CB74875AD1F79C5166560AA7C208BEBF243BAF12F2

The AH is Complete! Now Client1 and 2 can message securely...
Next, Client 1 performs the NIL communicaiton key update...
Click Continue in Client1...

QT™ Messaging Example (continuing from the AH..)

Client1

Now I am going to talk to Client2 in our application...

First, I'll create a Qwyit Return (QR) value
QR is
21FBFCFE73E926F4FF0E7F193C49EFA7410D2DDF28BEFAE92B47E81576A272C9B561F30467E4C36A6A34E80981567263EBCA53C59D2D759F1974641DB8D3D6DD
Click Continue in Client1...

Next, I'll use the SSK from the QDS and the just created QR to create a Session Master Key (SMK,in 2 halves, SQK and SEK)
New Client1-Client2 SQK is B43AE8ED5B24408DF588850EEB6EFE3B7597E371A024318A4B05E5F1C7384FAA
New Client1-Client2 SEK is A7FB01FFE338F3F1702A4C151EE7658B16A81CD6DBB150005D3673CB668A0A4E
Click Continue in Client1...

Now ready the message, which will be: Now is the time for all good men to come to the aid of the party
Then, encrypt it using QwyitTalk...
OpenReturn for this message is 7969F5F110C59D7521FDC1C59B10BD8F97D96A0973DECE0DCD5DB0C1E596FD0D
The QTQS message key is 7C4BC934713CEDA7495E45AC98DABD1E81CC99808DAAE2CFDC0BEC7B3EBAAB73
QwyitTalk Qwyit Start (QTQS) is 0123456789ABCDEF,
21FBFCFE73E926F4FF0E7F193C49EFA7410D2DDF28BEFAE92B47E81576A272C9B561F30467E4C36A6A34E80981567263EBCA53C59D2D759F1974641DB8D3D6DD, 7969F5F110C59D7521FDC1C59B10BD8F97D96A0973DECE0DCD5DB0C1E596FD0D,
4DA1A27A59137E39562917FFE4A677EADED42D0305FDDF86A8E2D58ACA74B03B,7876047E7E060502070A057777720E0B0773087409077107037102 0C07047A0277717C777005780B017C06020575047005757776070601740D070077057D01720777097909000506 7A71040176030E05067376030E05067178730908 7071710E77070073700A08080D010800740072700371770106040 6727A737B76720D7372,792C43622A4A13405F5413372C2924175256476555592D635E572B256229542B18452C635A56555518302E61315A2666252A54622A2517365B2062312030434A
Now, QT processing continues in Client 2...
Click Continue in Client1...

Client2

I've just received a QTQS message from Client1...I'll process it!
First, I need to unwrap the SSK...
The Client2 AHQ3 received message key is: AE088ECDB215E3727E058172A7096BC6DEE633999D141BEA7E437BD7869AA9DC
The Client2 AHQ3 received plaintext, which is the SSK, is:
934FFCFFE84B2A99068A16F5BF251F94349AC6A2887647A120CE0DEC5196DDE1F29A1EFB8C54309716F6641C9D91F3283BEEC9114E94EB7144C21FBEBEB73471

Next, I need to unwrap the SMK...
Now, take the just received QR and create the SMK for this message
New Client1-Client2 SQK is B43AE8ED5B24408DF588850EEB6EFE3B7597E371A024318A4B05E5F1C7384FAA
New Client1-Client2 SEK is A7FB01FFE338F3F1702A4C151EE7658B16A81CD6DBB150005D3673CB668A0A4E
Click Continue in Client1...

The QTQS message key is 7C4BC934713CEDA7495E45AC98DABD1E81CC99808DAAE2CFDC0BEC7B3EBAAB73

QTQS plaintext message is Now is the time for all good men to come to the aid of the party
Now I'll reply back to Client1...message is: Why won't anyone throw me a party with cup cake for my birthday?
Click Continue in Client1...

OpenReturn for this message is 9D1A5E78202F95449BBE8EFF5C8925A370FDE77CA6318C852E176AE50C8FE4E8
The QTQT message key is 5E6727967FD25485F855B7B8814BAFF011B88E1B165BAB81F2B7DB7DD075BA4D
QwyitTalk Qwyit Talk (QTQT) is A67241A3186F6CCD, 9D1A5E78202F95449BBE8EFF5C8925A370FDE77CA6318C852E176AE50C8FE4E8, 622D4F17455857114366255C4C5B5650664C5D472D4062555D11556231273444481135514C2D11214446152120295D11205D3017293B17262D42435D26204D7B

Now, QT processing continues in Client 1...
Click Continue in Client1...

Client1

I've just received a QTQT message from Client2...I'll process it using our shared SMK keys and the sent Open Return!

The QTQT message key is 5E6727967FD25485F855B7B8814BAFF011B88E1B165BAB81F2B7DB7DD075BA4D
QTQT plaintext message from Client 2 is Why won't anyone throw me a party with cup cake for my birthday?

Click Continue in Client1...

After sending and receiving and the SMK key life is reached, both Clients perform a PDAF PFS NIL communicaiton DSK key update
This is accomplished using the SQK and SEK in a PDAF. Optionally, for a 1,024-bit length, and then performing an OWC to pare to 512-bits
First, here in Client 1...
Click Continue in Client1...

Here is the PDAF result for Client1 new SQK/SEK keys:
FFB765EB3BAC35B2DDDB3EEC4565654F0FAA53599515392E35F82C9B62D63442F6B9CDCB33F92835AD83ECB1CB8CDCDEE518F8CBBF3C8BC5561DD8DF7B15EA87B8BFB3C8081478F55D6F6369DDA1213C573F816561EE153A0FC98D790ABDE12938823A93BA72F8D5DA6D78E5CF956553A9D1E88C1C988C8FBE7489DFF113F38F84329C4B30A2B512E5316CF31E7F0F75E84B0231D75F21D8753F94CE650A8372AC120CCC43AF90FDD3EFB6E79CE8986F869224E623A4466FC98F2F46AF8C7FF2011F28DB33AADE6B23663F35530EFE461DE71FAFE869FFD983401FAC45ECE75233EA20C08372BEBBA07BF5BC35A010BB7F7CFAF69412A67B39F98013A4582CCF
Here is OWC final result for Client1 new SQK key: E2B9E68DA81A9BB3F48EE6C087E48376549768A87BAC749B3977AF31BE5C268F
Here is OWC final result for Client1 new SEK key: 10A864B15C288EDAE5096FE6B4069A57682CB996A2478A166395D302C884EDEB
Now, update the SMK in Client 2...
Click Continue in Client1...

Client2

Here is the PDAF result for Client2 new SQK/SEK keys:
FFB765EB3BAC35B2DDDB3EEC4565654F0FAA53599515392E35F82C9B62D63442F6B9CDCB33F92835AD83ECB1CB8CDCDEE518F8CBBF3C8BC5561DD
8DF7B15EA87B8BFB3C8081478F55D6F6369DDA1213C573F816561EE153A0FC98D790ABDE12938823A93BA72F8D5DA6D78E5CF956553A9D1E88C1C98
8C8FBE7489DFF113F38F84329C4B30A2B512E5316CF31E7F0F75E84B0231D75F21D8753F94CE650A8372AC120CCC43AF90FDD3EFB6E79CE8986F8692
24E623A4466FC98F2F46AF8C7FF2011F28DB33AADE6B23663F35530EFE461DE71FAFE869FFD983401FAC45ECE75233EA20C08372BEBBA07BF5BC35A
010BB7F7CFAF69412A67B39F98013A4582CCF
Here is OWC final result for Client2 new SQK key: E2B9E68DA81A9BB3F48EE6C087E48376549768A87BAC749B3977AF31BE5C268F
Here is OWC final result for Client2 new SEK key: 10A864B15C288EDAE5096FE6B4069A57682CB996A2478A166395D302C884EDEB
Click Continue in Client1...

Client1

This concludes the AH and QwyitTalk demonstration. Thank you!

Client2

This concludes the AH and QwyitTalk demonstration. Thank you!

Appendix E – QwyitTalk™ VSU, AH and Messaging Code Calls

The following are the code calls for the example of the process flow for the complete QT™ system. These calls create the output, and those required to build the QT™ Security as a Service, of the *QDS-Full-Example.exe* app available from Qwyit LLC at www.qwyit.com. As explanation is provided above for the steps and output, only the corresponding code calls are included here:

QT™ VSU Example

Client1

Client, initiate VSU within the application (and as required/desired) with the Directory Service/Server (Step 1, VSC1)
Goto www.Qwyit.com/Qwyit where an HTTPS session will begin. Enter the required information.
This generates a VSU start on the MyQTApp client application and the QDS (VSC1)
Client app now awaits receipt and entry of the key

QDS

Directory Service/Server reply to client (Steps 2,3, and 4 - VSQ2, VSQ3, VSQ4)
The client has submitted a VSU on the webpage, and is waiting for the reply
Generate OpenID (recommend 16 digit, 64-bit unique IDs), DSK for client
Generate a random, unique 16-hex digit, 64-bit OpenID for this client (stored in the DB per email/SMS/IP (and whatever other parameters in MyQTApp))
sOpenID_Client1 = GetRandom (16)

Generate random 352-bits that includes three (3) parts
Master Qwyit Key (MQK, 64 hex digits, 256-bits)
sClient1_MQK = GetRandom (64)

Email Offset Key (EOK, 16 hex digits, 64-bits)
sClient1_EOK = GetRandom (16)

SMS Offset Key (SOK, 8 hex digits, 32-bits)
sClient1_SOK = GetRandom (8)

Perform a PDAF(EOK, SOK) generating 8 rounds (cycling through each round of SOK moving the start 1 digit to the right)
Result is the Master Offset Key (MOK, 64 hex digits, 256-bits)
sClient1_MOK = PubDS.PDAF(sClient1_EOK, 32, 0, sClient1_SOK, 1, 0, 0)

Perform a PDAF(MQK, MOK) generating 1 round
Result is a the Master Exchange Key (MEK, second 64 hex digits, 256-bits) - store with MQK as this new client's Qwyit keys

sClient1_MEK = PubDS.PDAF(sClient1_MQK, 32, 0, sClient1_MOK, 1, 0, 0)

Encrypt EOK and SOK using MQK
Concatenate EOK and SOK
sClient1_TempEOK_SOK = sClient1_EOK & sClient1_SOK

MOD16 add the first 96-bits (24 hex digits) of MQK with the concatenated EOK and SOK
sClient1_EOK_SOK_encrypted = PubDS.MOD16(Left(sClient1_MQK, 24), sClient1_TempEOK_SOK)

Separate the encrypted EOKe (LEFT 16 or 64-bits) and SOKe (RIGHT 16 or 32-bits) and ready to send in email and SMS respectively

Reply (VSQ2) with OpenID, (MQK) during the HTTPS session on the webpage
Reply Step 3 to email address, sending OpenID and EOKe
Reply Step 4 to SMS number, sending OpenID and SOKe

Client1

Client decrypt of reply and confirmation (Step 5, VSC5)
Client will cut & paste the session-shown OpenID, EOKe and SOKe into the MyQTApp, and OpenID and MQK will appear in the app/window, as the HTTPS session has received those
Open email message to reveal OpenID and EOKe; cut & paste (or type) into application (both values)
Open SMS text message to reveal OpenID and SOKe; cut and paste (or type) into application (both values)
Click button to "Store Key" (or some relevant, pertinent UI text)
Applet will check that all 3 OpenIDs match
Applet will decrypt EOKe and SOKe, and create MEK

Concatenate EOKe and SOKe
sClient1_TempEOK_SOK_encrypted = Left(sClient1_EOK_SOK_encrypted, 16) & Right(sClient1_EOK_SOK_encrypted, 8)

MOD16D the first 96-bits (24 hex digits) of MQK with the concatenated EOKe and SOKe
sClient1_EOK_SOK_decrypted = PubDS.MOD16D(sClient1_TempEOK_SOK_encrypted, sClient1_MQK)

Separate the decrypted EOK and SOK and ready for use to create MEK
Perform a PDAF(EOK, SOK) generating 8 rounds (cycling through each round of SOK moving the start 1 digit to the right)
Result is the Master Offset Key (MOK, 64 hex digits, 256-bits)
sClient1_MOK_decrypted = PubDS.PDAF(Left(sClient1_EOK_SOK_decrypted, 16), 32, 0, Right(sClient1_EOK_SOK_decrypted, 8), 1, 0, 0)

Perform a PDAF(MQK, MOK) generating 1 round
Result is a the Master Exchange Key (MEK, second 64 hex digits, 256-bits) - store with MQK as this new client's Qwyit keys
sClient1_MEK_decrypted = PubDS.PDAF(sClient1_MQK, 32, 0, sClient1_MOK_decrypted, 1, 0, 0)
Concatenate MQK and MEK making complete 512-bit DSK – Insert into MyQTAPP storage (store DSK, OpenID in cookie, file, db – method per MyQTApp requirements)

Reply (VSC5) to QDS with Confirmation message
Perform Qwyit cipher using DSK (MQK and MEK), generating message key (W)
Use message key to encrypt confirmation message, which is a 128-bit, 32 hex digit ID salt created by using the last 32 digits (128-bits) of the MQK in a PDAF with the last 32 digits (128-bits) of the MEK

Perform a PDAF(MQK last 128-bits, MEK last 128-bits); result is Confirmation Salt
sClient1_ConfirmationSalt = PubDS.PDAF(Right(sClient1_MQK, 32), 16, 0, Right(sClient1_MEK_decrypted, 32), 1, 0, 0)

Perform Qwyit encrypt using W and CS, result is CS encrypted (CSe)
sOpenReturn = GetRandom (64) 'return a 64-digit 256-bit random number
sClient1_ConfirmationSalt_encrypted = PubDS.QwyitSCX(sClient1_MQK, sClient1_MEK_decrypted, sOpenReturn, sClient1_ConfirmationSalt)
Retrieve the individual values
sRout = ParFind(sClient1_ConfirmationSalt_encrypted, "R")
sWout = ParFind(sClient1_ConfirmationSalt_encrypted, "W")
sTout = ParFind(sClient1_ConfirmationSalt_encrypted, "T")
sCiphertext = sTout

Send (VSC5) output (OpenID, OR, CSe) sent using Port 4180 to the QDS
[VSC5: OpenID, OR, CSe]

QDS

QDS decrypt of confirmation message (Step 5, VSQ5)
Perform Qwyit decrypt using W and CSe, result is CS received decrypted (CS)
sClient1_ConfirmationSalt_decrypted = PubDS.QwyitSCX(sClient1_MQK, sClient1_MEK, sOpenReturn, , sCiphertext)
Retrieve the individual values
sRout = ParFind(sClient1_ConfirmationSalt_decrypted, "R")
sWout = ParFind(sClient1_ConfirmationSalt_decrypted, "W")
sTout = ParFind(sClient1_ConfirmationSalt_decrypted, "T")
sPlaintext = sTout

Perform a PDAF(MQK 128-bits, MEK last 128-bits); result is Confirmation Salt generated
sClient1_ConfirmationSalt_qds = PubDS.PDAF(Right(sClient1_MQK, 32), 16, 0, Right(sClient1_MEK_decrypted, 32), 1, 0, 0)

Compare CS received decrypted with CS generated
If sClient1_ConfirmationSalt_qds = sPlaintext Then
Match confirmed, store IP Address, OpenID, DSK, email address, SMS (and whatever other required session ID was collected) into QDS MyQTApp section - method?'
Else
Doesn't match, error sent - "Uh oh...they aren't the same... Something is wrong - enact error routines...Client1 is NOT a QwyitTalk member!"
End If

Client1

If no error returned, VSU PROCESSING COMPLETE!

QT™ AH Example

Client1

Retrieve MQK and MEK from myQTApp (the client application that has performed a VSU w/QDS)

First, calc my CS to associate with this AH (this is myQTApp system defined, as recommended by Qwyit – either as per above definition in the AH section, or as here: PDAF the MQK and MEK to full length, then perform an OWC on the result – switch the Value and Offset keys from the VSU CS above)

```
sClient1_CS_PDAF = PubDS.PDAF(sClient1_MEK, 32, 0, sClient1_MQK, 1, 0, 0)
sClient1_CS = PubDS.OWC(sClient1_CS_PDAF,1)
```

Send Authorization Request (AUTR) to QDS in order to receive an SSK for communication w/intended Client

Send AHC1:SenderOpenID, ReceiverOpenID and CS encrypted in Qwyit Cipher using DSK to QDS
[AHC1: SenderOpenID, ReceiverOpenID, OR, CT]

```
sOpenReturn = GetRandom (64) 'return a 64-digit 256-bit random number
```

Retrieve OpenID of my intended recipient from myQTApp (the client app has stored, and/or a method to retrieve other QTApp participant OpenIDs – where the recipient can be chosen from)

Encrypt my CS

```
sClient1_AHC1_encrypted = PubDS.QwyitSCX(sClient1_MQK, sClient1_MEK, sOpenReturn, sClient1_CS)
```

Retrieve the individual values

```
sRout = ParFind(sClient1_AHC1_encrypted, "R")
sWout = ParFind(sClient1_AHC1_encrypted, "W")
sTout = ParFind(sClient1_AHC1_encrypted, "T")
sCipherTextOut = PubDS.BT4("", sTout)
sCiphertext = sTout
```

Send [AHC1: SenderOpenID, ReceiverOpenID, OR, CT] where CT is sCiphertext, using Port 4180 to the QDS

QDS

Decrypt AHC1 from Client using respective OpenID and DSK in Qwyit Cipher
The sOpenReturn is used from the received AHC1 message....and the MQK and MEK are retrieved using the received sOpenID in the AHC1
This is done by retrieving the MEK/MQK from the QDS database of the OpenID from the sending Client 1 (SenderOpenID)
After retrieval, decrypt the received ACH1

```
sClient1_AHC1_decrypted = PubDS.QwyitSCX(sClient1_MQK, sClient1_MEK, sOpenReturn, , sCiphertext)
```

Retrieve the individual values

```
sRout = ParFind(sClient1_AHC1_decrypted, "R")
```

```
sWout = ParFind(sClient1_AHC1_decrypted, "W")
sTout = ParFind(sClient1_AHC1_decrypted, "T")
sPlaintext = sTout
```

Check that Client who is requesting is the same as encrypted request and Receiver client exists
This is done by separating the Client2 OpenID and the CS in the received Plaintext

```
First, check that the Calc CS equals the sent, decrypted CS
sClient1_CS_PDAF = PubDS.PDAF(sClient1_MEK, 32, 0, sClient1_MQK, 1, 0, 0)
sClient1_CS = PubDS.OWC(sClient1_CS_PDAF,1)
If Client1_CS = sPlaintext
        'Continue!
Else
        'Proceed w/any system error handling, including Sending error to Client 1, stating 'Incorrect AHC1 received!', and mark in DB, etc.
End If
```

Proceed by retrieving the MEK/MQK keys for Client2 from the QDS database from the ReceiverOpenID in the received AHC1

Create Session Start Key (SSK, 128 hex digits, 512-bits)
[IF to be stored, by MID and SenderOpenID, ReceiverOpenID – requirement by 3-letter agency ONLY]

```
sClient1_Client2_SSK = GetRandom (128)
```

QDS reply to Clients (Step 3, AHQ3)
Send SSK Accept (AHQ3) to Sender Client encrypted in Qwyit Cipher using Client1 DSK (If Receiver Client does not exist in QDS – Handle as per myQTApp instructions.) The AHQ3 includes the wrapped SSK encrypted by that Receiver client's DSK

```
sOpenReturn = GetRandom (64) 'return a 64-digit 256-bit random number
sClient1_AHQ3_encrypted = PubDS.QwyitSCX(sClient1_MQK, sClient1_MEK, sOpenReturn, sClient1_Client2_SSK)
```

Retrieve the individual values

```
sRout = ParFind(sClient1_AHQ3_encrypted, "R")
sWout = ParFind(sClient1_AHQ3_encrypted, "W")
sTout = ParFind(sClient1_AHQ3_encrypted, "T")
sCipherTextOut = PubDS.BT4("", sTout)
sCiphertext = sTout

Now, the Receiver Client SSK bundle
sClient2_AHQ3_encrypted = PubDS.QwyitSCX(sClient2_MQK, sClient2_MEK, sOpenReturn, sClient1_Client2_SSK)
```

Retrieve the individual values

```
sRout = ParFind(sClient2_AHQ3_encrypted, "R")
sWout = ParFind(sClient2_AHQ3_encrypted, "W")
sTout = ParFind(sClient2_AHQ3_encrypted, "T")
sCipherTextOut2 = PubDS.BT4("", sTout)
sCiphertext2 = sTout
```

Send [AHQ3: sOpenID_Client1, sOpenReturn, sCipherTextOut, sCipherTextOut2] using Port 4180 to Client1

Client1

Client1 decrypts SSK (Step 4, AHC4)
Sender Client1 decrypts AHQ3 from QDS using DSK in Qwyit Cipher

Perform Qwyit decrypt using W and CSe, result is CS received decrypted (CS)
sClient1_SSK_decrypted = PubDS.QwyitSCX(sClient1_MQK, sClient1_MEK, sOpenReturn, , sCiphertext)
Retrieve the individual values
sRout = ParFind(sClient1_SSK_decrypted, "R")
sWout = ParFind(sClient1_SSK_decrypted, "W")
sTout = ParFind(sClient1_SSK_decrypted, "T")
sPlaintext = sTout

Client1 only reply to QDS if unable to decrypt key (for whatever reason), sending an AHE for error – IF NO ERROR, the AH is Complete! Proceed

Client1 will store the sOpenReturn and sCipherTextOut2, or hold in memory after receipt, for including in the following QTQS send to Client 2

Now, Client1 performs a PDAF PFS NIL communciation DSK key update (QDS will in the next section)
Perform a PDAF(MQK 256-bits, MEK 256-bits) for two rounds, creating 512-bit result w/MQK, MEK halves
Optionally, extend the above PDAF for 2 rounds (doubling length) and perform an OWC
sClient1_Key_PFS_Update = PubDS.PDAF(sClient1_MQK, 256, 0, sClient1_MEK, 1, 0, 0)
sClient1_MQK_MEK_New = PubDS.OWC(sClient1_Key_PFS_Update, 1)
sClient1_MQK_New = Left(sClient1_MQK_MEK_New, 64)
sClient1_MEK_New = Right(sClient1_MQK_MEK_New, 64)

QDS

Performs the exact same PFS key updates for Client1 – this can be done after performing the AHQ3 send…and no error returned…so ‘wait time’ is important, AND/OR performed immediately and stored differently, then handled by DB to update…
sClient1_Key_PFS_Update = PubDS.PDAF(sClient1_MQK, 256, 0, sClient1_MEK, 1, 0, 0)
sClient1_MQK_MEK_New = PubDS.OWC(sClient1_Key_PFS_Update, 1)
sClient1_MQK_New = Left(sClient1_MQK_MEK_New, 64)
sClient1_MEK_New = Right(sClient1_MQK_MEK_New, 64)

The AH is Complete! Now Client1 and 2 can message securely and QDS is Finished!...

QT™ Messaging Example (continuing from the AH..)

Client1

Now I am going to talk to Client2 in our MyQTApp...

Client 1 sends Qwyit Start to Client Receiver (Qwyit Start, QTQS)
    Sending Client1 generates a Qwyit Return (QR, 128 hex digits, 512-bits)
    sClient1_2_QR = GetRandom (128)

Using the AH SSK from the QDS for this communication:
Perform MOD16(SSK,QR); result is the Session Master Key (SMK, in 2 halves, SQK and SEK)
    sSMK_Client1 = PubDS.MOD16(sClient1_Client2_SSK, sClient1_2_QR)
    sSQK_Client1 = Left(sSMK_Client1, 64)
    sSEK_Client1 = Right(sSMK_Client1, 64)

Create new message content (IMSG, the Initial Message opening communication)
Perform Normal Trusted Operation using Qwyit Cipher on IMSG with SMK (SQK and SEK)
    sIMSG = "Now is the time for all good men to come to the aid of the party" (EXAMPLE MyQTApp message)
    sOpenReturn = GetRandom (64)
    sClient1_IMSG_encrypted = PubDS.QwyitSCX(sSQK_Client1, sSEK_Client1, sOpenReturn, sIMSG)
Retrieve the individual values
    sRout = ParFind(sClient1_IMSG_encrypted, "R")
    sWout = ParFind(sClient1_IMSG_encrypted, "W")
    sTout = ParFind(sClient1_IMSG_encrypted, "T")
    sCipherTextOut = PubDS.BT4("", sTout)
    sCiphertext = sTout

Send Qwyit Start (QTQS) message from within MyQTApp to Client 2
    QTQS:OpenID, QR, OR, Ciphertext of IMSG, which is [QTQS: OpenID, QR, OR, OR2, CT2, CT] where OR2 and CT2 are from the AHQ3 QDS message, as stored and/or in memory to include in this send

Client2

Receiving Client decrypts QTQS (Qwyit Talk, QTQT)
    Determines SMK from the QS message by performing the same MOD16(SSK,QR)
    sClient2_SSK_decrypted = PubDS.QwyitSCX(sClient2_MQK, sClient2_MEK, sOR2, , sCT2)
Retrieve the individual values
    sRout = ParFind(sClient2_SSK_decrypted, "R")
    sWout = ParFind(sClient2_SSK_decrypted, "W")
    sTout = ParFind(sClient2_SSK_decrypted, "T")
    sPlaintext2 = sTout

Now, create the SQK and SEK used by Client1 in the QTQS
sSMK_Client2 = PubDS.MOD16(sPlaintext2, sClient1_2_QR)
sSQK_Client2 = Left(sSMK_Client2, 64)
sSEK_Client2 = Right(sSMK_Client2, 64)

Commences Normal Trusted Operation message Receive decryption using SMK in Qwyit Cipher on OR, Ciphertext
sClient2_IMSG_decrypted = PubDS.QwyitSCX(sSQK_Client2, sSEK_Client2, sOpenReturn, , sCiphertext)
Retrieve the individual values
sRout = ParFind(sClient2_IMSG_decrypted, "R")
sWout = ParFind(sClient2_IMSG_decrypted, "W")
sTout = ParFind(sClient2_IMSG_decrypted, "T")
sPlaintext = sTout

MyQTApp portrays sPlaintext as the received Client 1 message

Clients continue communication by performing Normal Trusted Operation messaging w/Qywit Cipher using SMK (QwyitTalk, QTQT Encrypt/Decrypt)
sIMSG_2 = "Why won't anyone throw me a party with cup cake for my birthday?" (EXAMPLE MyQTApp reply message)
sOpenReturn = GetRandom (64)
sClient2_IMSG2_encrypted = PubDS.QwyitSCX(sSQK_Client2, sSEK_Client2, sOpenReturn, sIMSG_2)
Retrieve the individual values
sRout = ParFind(sClient2_IMSG2_encrypted, "R")
sWout = ParFind(sClient2_IMSG2_encrypted, "W")
sTout = ParFind(sClient2_IMSG2_encrypted, "T")
sCipherTextOut = PubDS.BT4("", sTout)
sCiphertext = sTout

Send Qwyit Talk (QTQT) from within MyQTApp…
Send [QTQT: OpenID, OR, CT] where CT is the sCiphertext of just encrypted message contents

Client1

Messaging continues with QTQTs sent back and forth in MyQTApp…
QTQT Decrypt w/SMK (SQK, SEK) in Qwyit Cipher to reveal secure message contents
sClient1_IMSG_decrypted = PubDS.QwyitSCX(sSQK_Client1, sSEK_Client1, sOpenReturn, , sCiphertext)
Retrieve the individual values
sRout = ParFind(sClient1_IMSG_decrypted, "R")
sWout = ParFind(sClient1_IMSG_decrypted, "W")
sTout = ParFind(sClient1_IMSG_decrypted, "T")
sPlaintext = sTout

MyQTApp portrays sPlaintext as the received Client 2 message

At known, system defined end of SMK key life reached (# of messages, time, etc.), both Clients perform a PDAF PFS Nil communication SMK key update
Perform a PDAF(SQK 256-bits, SEK 256-bits) for two rounds, creating 512-bit SMK result w/SQK, SEK halves
Optionally, extend the above PDAF for 2 rounds (doubling length) and perform an OWC
sClient1_Key_PFS_Update = PubDS.PDAF(sSQK_Client1, 256, 0, sSEK_Client1, 1, 0, 0)
sClient1_SQK_SEK_New = PubDS.OWC(sClient1_Key_PFS_Update, 1)
sClient1_SQK_New = Left(sClient1_SQK_SEK_New, 64)
sClient1_SEK_New = Right(sClient1_SQK_SEK_New, 64)

Client2

At known, system defined end of SMK key life reached (# of messages, time, etc.), both Clients perform a PDAF PFS NIL communicaiton SMK key update
Perform a PDAF(SQK 256-bits, SEK 256-bits) for two rounds, creating 512-bit SMK result w/SQK, SEK halves
Optionally, extend the above PDAF for 2 rounds (doubling length) and perform an OWC
sClient2_Key_PFS_Update = PubDS.PDAF(sSQK_Client2, 256, 0, sSEK_Client2, 1, 0, 0)
sClient2_SQK_SEK_New = PubDS.OWC(sClient2_Key_PFS_Update, 1)
sClient2_SQK_New = Left(sClient2_SQK_SEK_New, 64)
sClient2_SEK_New = Right(sClient2_SQK_SEK_New, 64)

Client1

Continue as QTQT above, including system defined PFS SMK updates

Client2

Continue as QTQT above, including system defined PFS SMK updates