Quantum Secure Messaging Protocol – QSMP

Revision 1a, May 22, 2021

John G. Underhill – john.underhill@protonmail.com

This document is an engineering level description of the QSMP encrypted and authenticated network messaging protocol.

In its contents, a guide to implementing QSMP, an explanation of its design, links to a C reference implementation, as well as references to its comprising primitives and links to supporting documentation.

Contents **Page**

Figures 2
Tables 3
Foreword 4
1: Introduction 5
2: Scope 7
3: References 8
4: Terms and definitions 9

Foreword

This document is intended as the preliminary draft of a new standards proposal, and as a basis from which that standard can be implemented. We intend that this serve as an explanation of this new technology, and as a complete description of the protocol.

This document is the first revision of the specification of QSMP, further revisions may become necessary during the pursuit of a standard model, and revision numbers shall be incremented with changes to the specification. The reader is asked to consider only the most recent revision of this draft, as the authoritative expression of the current working model of the specification.

Future revisions of this standards draft can be found at:
https://github.com/Steppenwolfe65/QSMP

The author of this specification is John G. Underhill, and can be reached at
john.underhill@protonmail.com

1 Introduction

The Secure Shell protocol (SSH), was designed as a secure replacement for Telnet, RSH, and other unsecured remote shell applications. SSH has several different encryption and key exchange methods; it can use a fixed shared symmetric key, a fixed asymmetric public/private key, or a Diffie-Hellman key exchange. SSH is not limited to remote shell applications, it is used extensively as a mechanism for establishing a secure connection between remote hosts. It is employed to establish a secure channel for secure tunneling, VPNs, encrypted proxies, remote filesystems, secure file transfer, and commodity trading applications like SWIFT. QSMP has been designed to be a stronger, safer, post-quantum alternative to SSH.

QSMP is a quantum secure messaging protocol, that employs state of the art asymmetric ciphers and signature schemes, and the authenticated post-quantum strength authenticated symmetric stream cipher. The current incarnation can use the Kyber or McEliece asymmetric ciphers, and the Dilithium or Sphincs+ signature schemes, the leading round 3 candidates in the NIST Post Quantum competition. It uses the authenticated symmetric stream cipher RCS, based on the Rijndael-256 cipher, with increased rounds, a cryptographically strong key-schedule, and AEAD authentication using KMAC. QSMP was designed to be more flexible and more secure than the SSH protocol, and can be used in any context where strong post-quantum security is required.

1.1 Purpose

The QSMP secure messaging protocol, utilized in conjunction with quantum secure asymmetric and symmetric cryptographic primitives, is used to create an encrypted and authenticated bi-directional communications channel. This specification presents a secure messaging protocol that creates an encrypted communications channel, in such a way that:

1) The asymmetric cipher keys for both the send and receive channels, are ephemeral, and encapsulate shared secrets for each channel that are also unique to each session (forward secrecy).
2) The capture of the shared keys does not reveal any information about future sessions (predicative resistance).
3) That each host in the bi-directional communications stream, is responsible for creating the shared secret for the channel they transmit on.

QSMP is a two-channel duplexed communications system. It uses a separate shared secret to key both the transmit and receive channels in a communications stream. Each host is responsible for generating and encapsulating the symmetric key that host transmits data on. Asymmetric cipher keys are ephemeral, and unique keys are generated for each session. The system works in a client/server model, where a client requests a connection from the server to initiate the key exchange. The server signs the asymmetric encapsulation key sent to the client, and the client uses a MAC function to authenticate the asymmetric encapsulation key sent to the server. The signature algorithm is quantum secure, and the public verification key can be signed using the X509 hierarchal signature scheme, to create a ‘chain of trust’. A strong emphasis has been placed

on authentication with QSMP, with the entire key exchange using authentication to guarantee the exchange, as well as the symmetric stream cipher that uses KMAC authentication and additional data parameters that authenticate the QSMP packet headers.

2 Scope

This document describes the QSMP secure messaging protocol, which is used to establish an encrypted and authenticated duplexed channel between two hosts. This document describes the complete asymmetric key exchange, authentication, and the establishment of a VPN. This is a complete specification, describing the cryptographic primitives, the key derivation functions, and the complete client to server messaging paradigm.

Test vectors and C reference code will be available at https://github.com/Steppenwolfe65/QSMP

2.1 Application

This protocol is intended for institutions that implement secure communication channels used to encrypt and authenticate secret information exchanged between remote terminals.

The key exchange functions, authentication and encryption of messages, and message exchanges between terminals defined in this document must be considered as mandatory elements in the construction of an QSMP communications stream. Components that are not necessarily mandatory, but are the recommended settings or usage of the protocol shall be denoted by the key-words **SHOULD**. In circumstances where strict conformance to implementation procedures are required but not necessarily obvious, the key-word **SHALL** will be used to indicate compulsory compliance is required to conform to the specification.

3 References

3.1 Normative References

The following documents serve as references for key components of QSMP:

1. NIST FIPS 202: SHA-3 Standard: Permutation-Based Hash and Extendable Output Functions
2. NIST SP 800-185: Derived Functions cSHAKE, KMAC, TupleHash and ParallelHash
3. NIST SP 800-90A: Recommendation for Random Number Generation
4. NIST SP 800-108: Recommendation for Key Derivation using Pseudorandom Functions
5. NIST FIPS 197 The Advanced Encryption Standard

3.2 Reference Links

1. The QSMP C implementation: https://github.com/Steppenwolfe65/QSMP

2. The QSC Cryptographic library: https://github.com/Steppenwolfe65/QSC
3. The Keccak Code Package: https://github.com/XKCP/XKCP
4. NIST AES FIPS 197: http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf

4 Terms and Definitions

4.1 Kyber

The Kyber asymmetric cipher and NIST Round 3 Post Quantum Competition candidate.

4.2 McEliece

The McEliece asymmetric cipher and NIST Round 3 Post Quantum Competition candidate.

4.3 Dilithium

The Dilithium asymmetric signature scheme and NIST Round 3 Post Quantum Competition candidate.

4.4 SPHINCS+

The SPHINCS+ asymmetric signature scheme and NIST Round 3 Post Quantum Competition candidate.

4.5 RCS

The Rijndael-256 Cryptographic Stream (RCS) authenticated symmetric stream cipher.

4.6 SHA-3

The SHA3 hash function NIST standard, as defined in the NIST standards document FIPS-202; SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions.

4.7 SHAKE

The NIST standard Extended Output Function (XOF) defined in the SHA-3 standard publication FIPS-202; SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions.

4.8 KMAC

The SHA3 derived Message Authentication Code generator (MAC) function defined in NIST special publication SP800-185: SHA-3 Derived Functions: cSHAKE, KMAC, TupleHash and ParallelHash.

5 Structures

5.1 Protocol string

The Protocol string is comprised of four unique components;

1. The asymmetric signature scheme, including the security strength of the asymmetric signature scheme (s1, s2, s3), ex. dilithium-s2.
2. The asymmetric encapsulation cipher, including the security strength, ex. mceliece-s2.
3. The hash function and security strength, ex. sha3-256.
4. The symmetric cipher and security strength, ex. rcs-256

The protocol string is used during the initial protocol negotiation to identify the protocol settings of the client and server. The client and server must support a common parameter set to establish a connection.

Signature Scheme	Asymmetric Cipher	HASH Function	Symmetric Cipher
Dilithium	Kyber	SHA3-256	RCS-256
Sphincs+	McEliece	SHA3-256	RCS-256

Table 5.1: The Protocol string choices in revision 1a.

5.2 Client Key

The client key is an internal structure that stores signature verification key related variables, including the expiration time, the protocol string, the public signature verification key, and the signature key identity string.

Parameter	Data Type	Bit Length	Function
Expiration	Uint64	64	Validity check
Configuration	Uint8 array	320	Protocol check
Key ID	Uint8 array	128	Identification
Verification Key	Uint8 array	Variable	Authentication

Table 5.2: The client key structure.

The expiration parameter is a 64-bit unsigned integer that holds the seconds since the last epoch (01/01/1900) to the time the key remains valid. This value is checked during the initialization of the client, if the key has expired, the connection attempt is halted.

The configuration parameter contains the protocol string associated with the signature verification public-key, the asymmetric cipher, the hash family, and the symmetric cipher. This value is checked during initialization, and if the string does not match the internal protocol string, the connection initialization is aborted.

The key identity array is a 16-byte array that uniquely identifies a public verification key. This identifier can be used to match the key on a server.

The public key, is the public asymmetric signature verification key. This key can be distributed to clients, posted to a website, or distributed in any way public or private. It can also be signed using X509 to create a 'chain of trust', in an extension to this protocol. It is used to verify the signature of an asymmetric encapsulation key, sent to the client during the key exchange.

5.3 Server Key

The server key is identical to the client key except for one additional parameter, the signing key. It contains both the signature schemes verification and secret signing keys, along with the expiration, configuration, and key identity parameters.

Data Name	Data Type	Bit Length	Function
Expiration	Uint64	64	Validity check
Configuration	Uint8 array	320	Protocol check
Key ID	Uint8 array	128	Identification
Verification Key	Uint8 array	Variable	Authentication
Signing Key	Uint8 array	Variable	Authenticating

Table 5.3: The server key structure.

5.4 Client State

The client state is an internal structure that contains all the variables required by the QSMP operations. This includes elements copied from the client key structure at initialization, send and receive channels symmetric cipher states, and asymmetric cipher key-pairs.

Data Name	Data Type	Bit Length	Function
Expiration	Uint64	64	Validity check
Configuration	Uint8 array	320	Protocol check
Key ID	Uint8 array	128	Identification
Verification Key	Uint8 array	Variable	Authentication
Encapsulation Key	Uint8 array	Variable	Asymmetric Encryption
Decapsulation Key	Uint8 array	Variable	Asymmetric Decryption
Cipher Send State	Structure	Variable	Symmetric Encryption
Cipher Receive State	Structure	Variable	Symmetric Decryption
PkHash	Uint8 array	256	Authentication
Session Token	Uint8 array	256	Authentication
ExFlag	Uint8	8	Protocol Check

Table 5.4: The client state structure.

5.5 Server State

The server state is identical to the client state, except for the additional signature key parameter.

Data Name	Data Type	Bit Length	Function
Expiration	Uint64	64	Validity check
Configuration	Uint8 array	320	Protocol check
Key ID	Uint8 array	128	Identification
Verification Key	Uint8 array	Variable	Signature Verification
Signature Key	Uint8 array	Variable	Authentication Signing
Encapsulation Key	Uint8 array	Variable	Asymmetric Encryption
Decapsulation Key	Uint8 array	Variable	Asymmetric Decryption
Cipher Send State	Structure	Variable	Symmetric Encryption
Cipher Receive State	Structure	Variable	Symmetric Decryption
PkHash	Uint8 array	256	Authentication
Session Token	Uint8 array	256	Authentication
ExFlag	Uint8	8	Protocol Check

Table 5.5: The server state structure.

5.6 Keep Alive State

Parameter	Data Type	Bit Length	Function
Expiration Time	Uint64	64	Validity check
Packet Sequence	Uint64	64	Protocol check
Received Status	Bool	8	Status

Table 5.6: The keep alive state.

5.7 QSMP Packet Header

The QSMP packet header is 9 bytes in length, and contains:

1. The **Packet Flag**, the type of message contained in the packet; this can be any one of the key-exchange stage flags, a message, or an error flag.
2. The **Packet Sequence**, this indicates the sequence number of the packet exchange.
3. The **Message Size**, this is the size in bytes of the message payload.

The message is a variable sized array, up to QSMP_MESSAGE_MAX in size.

Packet Flag **1 byte**	**Packet Sequence** **8 bytes**	**Message Size** **4 bytes**
Message **Variable Size**		

Figure 5.7: The QSMP packet structure.

This packet structure is used for both the key exchange protocol, and the established VPN.

5.8 Flag Types

The following are a preliminary list of packet flag types used by QSMP:

Flag Name	Numerical Value	Flag Purpose
None	0x00	No flag was specified, the default value.
Connect Request	0x01	The key-exchange client connection request flag.
Connect Response	0x02	The key-exchange server connection response flag.
Connection Terminated	0x03	The connection is to be terminated.
Encrypted Message	0x04	The message has been encrypted by the VPN.
Exstart Request	0x05	The QSMP key-exchange client exstart request flag
Exstart Response	0x06	The QSMP key-exchange server exstart response flag
Exchange Request	0x07	The key-exchange client exchange request flag.
Exchange Response	0x08	The key-exchange server exchange response flag.
Establish Request	0x09	The key- exchange client establish request flag.
Establish Response	0x0A	The key- exchange server establish response flag.
Keep Alive Request	0x0B	The packet contains a keep alive request.
Remote Connected	0x0C	The remote host has terminated the connection.

Remote Terminated	0x0D	The remote host has terminated the connection.
Session Established	0x0E	The VPN is in the established state.
Establish Verify	0x0F	The VPN is in the established verify state.
Unrecognized Protocol	0x10	The protocol string is not recognized
Error Condition	0xFF	The connection experienced an error.

Table 5.8: Packet header flag types.

5.9 Error Types

The following are a preliminary list of error messages used by QSMP:

Error Name	**Numerical Value**	**Description**
None	0x00	No error condition was detected.
Authentication Failure	0x01	The symmetric cipher had an authentication failure.
Bad Keep Alive	0x02	The keep alive check failed.
Channel Down	0x03	The communications channel has failed.
Connection Failure	0x04	The device could not make a connection to the remote host.
Connect Failure	0x05	The transmission failed at the KEX connection phase.
Decapsulation Failure	0x06	The asymmetric cipher failed to decapsulate the shared secret.
Establish Failure	0x07	The transmission failed at the KEX establish phase.
Exstart Failure	0x08	The transmission failed at the KEX exstart phase.
Exchange Failure	0x09	The transmission failed at the KEX exchange phase.
Hash Invalid	0x0A	The public-key hash is invalid.
Invalid Input	0x0B	The expected input was invalid.
Invalid Request	0x0C	The packet flag was unexpected.
Keep Alive Expired	0x0D	The keep alive has expired with no response.
Key Expired	0x0E	The QSMP public key has expired.
Key Unrecognized	0x0F	The key identity is unrecognized.

Packet Un-Sequenced	0x10	The packet was received out of sequence.
Random Failure	0x11	The random generator has failed.
Receive Failure	0x12	The receiver failed at the network layer.
Transmit Failure	0x13	The transmitter failed at the network layer.
Verify Failure	0x14	The expected data could not be verified.
Unknown Protocol	0x15	The protocol string was not recognized.

Table 5.9: Error type messages.

6 Operational Overview

The server generates a key-pair, the private key that the server uses to sign a key exchange, and the public key, which is distributed to clients, and contains the asymmetric signature verification key, along with the key identity array, protocol configuration string, and key expiration date.

The client initiates a connection, which if the key is valid and known to the server, instantiates a key exchange. Asymmetric cipher keys are authenticated and exchanged between the client and server, which generate a pair of shared secrets, used to key symmetric ciphers in the transmit and receive directions on a duplexed communications channel.

Any error during the key exchange or during the VPN operation, causes the client or server to send an error message to the other host, disconnect, and tear down the session. This includes checks for message synchronization, expected size of sent and received messages, and internal errors raised by cryptographic and network functions used by the key exchange and VPN.

6.1 Connect Request

The client initializes a key exchange operation, by sending the server a **connection request** packet. The packet message contains the client's key identification array, a random session token, and the protocol configuration string. The client stores a hash of a random session token, the protocol string, and the asymmetric signatures verification key in the *pkhash* state value, for use later in the key exchange.

6.2 Connect Response

The server first checks its database for a key matching the key identification sent by the client in the **connection request** packets message. The server then compares the configuration string

contained in the message against its own protocol string for a match. The server also verifies the key's expiration time, and if valid, loads the key into state. If the protocol configuration strings do not match, the server will send an **unknown protocol** error to the client and close the connection. If the client's key has expired, the server will send a **key expired** error message. If the key is not known to the server, the server sends a **key unrecognized** error message to the client. In any of these failures occur, the client is expected to close the connection, and pass the error up to components in the user interface software that can initiate actions or inform the user of the cause of the failure.

The server hashes the key identification array, the client's ransom session token, and its local copy of the asymmetric signature verification key, and stores the hash in its *pkhash* state value, for use as a unique session cookie.

The server generates a public/private asymmetric cipher key-pair. The server hashes the public key, and then signs the hash with the asymmetric signature schemes private key. The client has a copy of the asymmetric signature verification key, that will be used to verify this signature. The server stores the private asymmetric cipher key temporarily in its state.

The server adds the public asymmetric encapsulation key, and the public keys signed hash, to the **connection response** message.

6.3 Exstart Request

The client uses the public signature verification key to check the signature on the public asymmetric cipher-keys hash, that was sent along with the asymmetric ciphers public key in the **connect response** message. If the signature is verified, the asymmetric ciphers public key is hashed, and that hash is compared to the signed hash contained in the servers connect response packet. If the signature verification fails, the client sends an **authentication failure** message and terminates the connection, likewise if the hash check fails, the client sends a **hash invalid** error message.

The client uses the asymmetric cipher key to encapsulate a *shared secret*, producing a cipher-text that will be sent to the server, and generating a shared secret value.

This shared secret is combined with the session *pkhash*, to initialize cSHAKE, which derives the symmetric ciphers key and nonce. This cipher is set to encrypt, and acts as the client's transmit interface for the channel-1 VPN.

The asymmetric cipher-text is added to the **exstart request** packet, and sent to the server.

6.4 Exstart Response

The server decapsulates the cipher-text sent by the client in the **exstart request** packet, and extracts the *shared secret*. This shared secret is combined with the session *pkhash*, to initialize

cSHAKE, which derives the symmetric ciphers key and nonce. This cipher instance is set to decrypt and authenticate, and acts as the server's receive interface for the channel-1 VPN.

The server sets the first byte **exstart response** message to the **remote connected** flag, signaling to the client that the first channel of the VPN has been established.

6.5 Exchange Request

The client checks the message flag of the **exstart request** packets message for the **remote connected** flag, then generates an asymmetric cipher key-pair. The client stores the private key in state, and adds the public key to the **exchange request** packets message. The client generates a MAC key, stores it in state, and copies it to the exchange request message. The client serializes the exchange request packet header and adds it to the associated data of the channel-1 transmit cipher. The client then encrypts the packets message with the channel-1 transmit cipher.

The second key exchange uses a symmetric authentication MAC key, sent to the server over the encrypted channel. That encrypted channels shared secret was verified with the asymmetric signature scheme, and this extends that authentication security to the second asymmetric cipher key exchange. Allowing the server to return a MAC authenticated asymmetric cipher-text over the unencrypted second channel, using the MAC function to authenticate the asymmetric cipher-text.

6.6 Exchange Response

The server serializes the exchange request packet header, and adds it to the associated data of the channel-1 receive cipher. The server authenticates and decrypts the **exchange request** message, and extracts the asymmetric cipher public key, and the MAC key. The server encapsulates a shared secret with the public asymmetric cipher key, producing a shared secret, and the cipher-text which is added to the **exchange response** message. The MAC key contained in the exchange request, is used to authenticate the asymmetric cipher-text, with the authentication code appended to the exchange response message.

The server combines the shared secret, and the *pkhash*, to key cSHAKE and derive the key and nonce for the symmetric cipher. The cipher is keyed and set to encrypt, raising the server's transmit channel in the channel-2 VPN.

6.7 Establish Request

The client MACs the cipher-text in the **exchange response** packets message, using the key stored in state, and verifies the integrity of the cipher-text. The client then decapsulates the asymmetric cipher-text, and extracts the shared secret. The shared secret is combined with the *pkhash*, to key cSHAKE and derive the key and nonce for the channel-2 symmetric cipher. The symmetric cipher is keyed and set to authenticate and decrypt, raising the client's receive

channel on the channel-2 VPN. Both communications channels are now initialized. The client serializes the **establish request** packet header, and adds it to the channel-1 transmit cipher's state, then encrypts a copy of its *key identity* string, adds the cipher-text to the establish request message, and sends the packet to the server to confirm the second channel has been raised.

6.8 Establish Response

The server serializes the **establish request** packet, and adds it to the associated data of the server's channel-1 receive cipher, and then authenticates and decrypts the establish request packets message. The server compares the decrypted key identification array to the one it has stored in state. If the message is verified, the server sets the internal **session established** flag, and is ready to process data on both channels of the VPN. The server serializes the establish response packet header and adds it to the channel-2 transmit cipher, then encrypts the key identification array, and adds it to the establish response message.

6.9 Establish Verify

The client adds the serialized packet header of the establish response to the associated data on its channel-2 receive cipher, and then authenticates and decrypts the message, and compares it to its key identification array in state. Upon successful decryption and verification of the message, the client now raises it's established flag, and is ready to process data.

7 Formal Description

Legend:

C -The client host

S -The server host

cng -The protocol configuration string

cprrx -A receive channels symmetric cipher instance

cprtx -A transmit channels symmetric cipher instance

cpt -The symmetric ciphers cipher-text

cpta -The asymmetric ciphers cipher-text

kid -The public keys unique identity array

pekh -The public asymmetric encapsulation key hash

pvk -The public signature verification key

sec -The shared secret derived from asymmetric encapsulation and decapsulation

spkh -The signed hash of the asymmetric public encapsulation-key

pkh -The session hash token, a hash of the session token, the configuration string, and the public signature verification-key

stok -A random string used as the session-token in the key exchange

DA_{sk} -The asymmetric encapsulation function and public key

EA_{sk} -The asymmetric decapsulation function and secret key

Dk -The symmetric decapsulation function and key

Ek -The symmetric encapsulation function and key

Exp -The key expansion function: cSHAKE

H -The hash function: sha3

M_{mk} -The MAC function and key: KMAC

SA_{sk} -Sign with the secret signature key

VA_{pk} -Verify a signature the public signature key

Key Exchange Sequence

7.1 Connect Request:

The client first checks the expiration date on the public key, if invalid, it queries the server for a new public verification key.

The client sends a connection request with its configuration string, key identity, and a random session token. The key identity (kid) is a multi-part 16-byte address and key identification array, used to identify the intended target server and corresponding key.

The client stores a hash of the session token, the configuration string, and the public asymmetric signature verification-key.

pkh = H(stok || cfg || pvk)

The client then sends the key identity string, session token, and the configuration string to the server.

C{kid, stok, cfg }->S

7.2 Connect Response:

The server responds with either an error message, or a response packet. Any error during the key exchange will generate an error-packet sent to the remote host, which will trigger a tear down of the connection on both sides.

The server first checks that it has the requested asymmetric signature key, using the key-identity array, then verifies that it has a compatible protocol configuration. The server stores a hash of the session token, the configuration string, and the public signature verification-key to create the public key hash.

pkh = H(stok || cfg || psk)

The server then generates an asymmetric encryption key-pair, stores the secret key, hashes the public key, and then signs the hash of the public encapsulation key using the asymmetric signature key. This signed hash can itself be signed by a ‘chain of trust’ model, like PGP or X509, using a signature verification extension to this protocol.

pekh = H(pke)

spkh = S_{sk}(pekh)

The server sends a response message containing a signed hash of a public asymmetric encapsulation-key, and a copy of that key.

S{spkh, pke}->C

7.3 Exstart Request:

The client verifies the signature of the public encapsulation keys hash, then generates its own hash of the public key, and compares them. If the hash matches, the client uses the public-key to encapsulate a shared secret.

cph = V_{sk}(H(pk)) cph := ph

cpta = EA_{pk}(sec)

The client then combines the shared secret and public key hash to key the derivation function, and uses the output to key the clients transmit-channel symmetric cipher.

k,n = Exp(sec || pkh)

cprtx(k,n)

The client transmits the cipher-text to the server.

C{cpta}->S

7.4 Exstart Response:

The server decapsulates the shared-secret, combines it with the public key hash to key the derivation function and generates the symmetric cipher key and nonce, it then keys the servers receive-channel cipher. The channel-1 VPN is now established.

$sec = DA_{pk}(cpta)$

$k,n = Exp(sec, pkh)$

$cprrx(k,n)$

The server sends the client an established message for the first channel.

S{m}->C

7.5 Exchange Request:

The client generates and stores an asymmetric cipher key-pair. The client generates a MAC key and stores it to state. The server then encrypts the MAC key and the asymmetric encapsulation-key using the channel-1 VPN, and sends the encrypted MAC and encapsulation keys to the server.

$pk,sk = G(cfg)$

$mk = G(n)$

$cpt = E_k(pk \,||\, mk)$

C{cpt}->S

7.6 Exchange Response:

The server decrypts the MAC and encapsulation keys, and uses the encapsulation-key to encapsulate a shared-secret for channel 2. The server then uses the MAC key received from the client, to MAC ciphertext, appending a MAC code to the message.

$mk,pk = D_k(cpt)$

$cpta = EA_{pk}(sec)$

The server then expands the shared secret and public key hash, and creates the symmetric ciphers key and nonce.

$k,n = Exp(sec, pkh)$

The MAC function is keyed with the MAC key sent by the client over the encrypted channel, the ciphertext is added to the MAC, and the output code is prepended to the message.

$cc = M_{mk}(cpta)$

The server's channel-2 transmission channel is initialized, and the authenticated cipher-text is sent to the client.

cprtx(k,n)

S{cc, cpta}->C

7.7 Establish Request:

The client uses the stored MAC key to key the MAC function, then adds the ciphertext to the hash. The client compares the hash code appended to the ciphertext with the one generated with the MAC function before decapsulating the shared key.

$mc = M_{mk}(cpta)$, mc := cc

The client then decapsulates the shared secret, combines it with the public key hash, and expands it.

$sec = DA_{sk}(cpta)$

k,n = Exp(sec, pkh)

The client then keys the clients receive channel, the second VPN is established, and the client sends an established message. The established message is the key identity array, encrypted and sent to the client.

cprrx(k,n)

$cpt = E_k(kid)$

C{cpt}->S

7.8 Establish Response:

The server authenticates and decrypts the message, then compares it to its copy of the kid.

$kid = D_k(cpt)$

The server re-encrypts the key identity array, using the channel-2 cipher, and sends it to the client for verification. Both channels of the server's VPN are now established.

$cpt = E_k(kid)$

S{cpt}->C

7.8 Establish Verify:

The client authenticates and decrypts the message, then compares it to its copy of the kid.

$kid = D_k(cpt)$

Both of the client's VPN channels are now established, the VPN is now ready to send and receive data.

7.9 Transmission:

The host, client or server, transmitting a message, first serializes the packet header and adds it to the symmetric ciphers associated data parameter. The host then encrypts the message, updates the MAC function with the cipher-text, and appends a MAC code to the end of the cipher-text.

The serialized packet header, including the message size, protocol flag, and sequence number, is added to the MAC state through the additional-data parameter of the authenticated stream cipher RCS. This unique data is added to the MAC function with every packet, along with the encrypted cipher-text.

$(cpt \mid\mid mc) = Ek(sh, m)$

The packet is decrypted by serializing the packet header and adding it to the MAC state, then finalizing the MAC on the cipher-text and comparing the output code with the code appended to the cipher-text. If the code matches, the cipher-text is decrypted, and the message passed up to the application.

$m = Dk(sh, cpt) == 0 \ ? \ m : NULL$