Symmetric Key Distribution Protocol – SKDP

Revision 1b, August 15, 2021

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

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

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

**Contents Page**

Foreword 2

Figures 3

Tables 4

1: Introduction 5

2: Scope 8

3: References 9

4: Terms and Definitions 10

5: Structures 11

6: Operational Overview 16

7: Formal Description 27

# **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 serves 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 SKDP, 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 SKDP specification.

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

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

SKDP, the algorithm constituting the SKDP messaging protocol is patent pending, and is owned by John G. Underhill and Digital Freedom Defense Incorporated. The code described herein is copyrighted, and owned by John G. Underhill and Digital Freedom Defense Incorporated.

**Figures**

**Contents Page**

Figure 6.1: SKDP client connect request 16

Figure 6.2: SKDP server connect response 17

Figure 6.3: SKDP client exchange request 19

Figure 6.4: SKDP server exchange response 21

Figure 6.5: SKDP client establish request 23

Figure 6.6: SKDP server establish response 25

Figure 6.7: SKDP client establish verify 26

# **Tables**

**Contents Page**

Table 5.1a: The client key structure 11

Table 5.1b: The device identity structure 11

Table 5.2: The server key structure 12

Table 5.3: The master key structure 12

Table 5.4: The client state structure 12

Table 5.5: The server state structure 13

Table 5.6: The keep alive state 13

Table 5.7: The SKDP packet structure 13

Table 5.8: Packet header flag types 14

Table 5.9: Error type messages 15

# **1 Introduction**

Key distribution, is one of the most challenging problems in cryptography. The internet has grown at an extraordinary pace since its inception, and is now a core communications medium that is used by billions of people around the globe. The information we send over this public medium, must be secured, as the internet has now become a primary tool in global commerce, and a communications infrastructure connecting people everywhere. The problem has always been, how do we distribute keys between connected devices, and do it in such a way as to best guarantee the continued security of that service.

The security mechanisms most widely used today, use asymmetric cryptography; public/private key cryptography for encryption and authentication. These asymmetric primitives use ‘trapdoor’ functions, whereby a part of a difficult mathematical problem is created using a public key, and solved using the private key. The problem with this approach, is that the underlying mathematical problems used by these asymmetric ciphers and signature schemes, are constantly being challenged by new knowledge and advances in computing technology. What seems an intractable problem today, could eventually be reduced, or even solved, at some future time. This is why asymmetric parameters are continually adjusted to make the problem more difficult, and why an entire order of asymmetric cryptography based on large integer factorization, will soon become obsolete, due to the emergence of quantum computers. It has been well established, that intelligence agencies collect and store encrypted streams on a vast scale. This is because even if the technology to break these encryption technologies does not currently exist, at some future point it will, and all of that collected traffic will become readable. We could face the same problem with LWE based cryptography in ten or twenty years, that we face now with cryptography based on elliptic curves or large integer factorization; eventually the technology, and the mathematics, will evolve and combine to create a new threat, a new way to break that cryptographic system. This is further complicated by the choice of parameters used in the design of asymmetric primitives, which are calculated based on projections established only in current knowledge, in a very performance-oriented field, that too often chooses less aggressive parameters in favor of improved performance characteristics. That we know communications are being captured and stored, but we cannot know what breakthroughs in technology are on the horizon, creates a serious problem that must be solved. We do not believe that any system based on asymmetric cryptography, can promise true long-term security, which must now be considered as the lifespan of a human being.

Symmetric cryptography may provide a part of the solution. Given sufficiently ‘strong’ symmetric cryptographic primitives, and longer key lengths, symmetric cryptography can be far more computationally expensive to solve, and perhaps even impossible to break for an indefinite time. The problems that have faced systems that use pre-shared symmetric keys, has always been one of scalability, and the concentration of security on to a single point of failure. For example, some systems use a single pre-shared key and session counter, to key a symmetric cipher and establish an ad hoc VPN; some SSH implementations use this naïve scheme. The problems being, that if the device is ever captured, all past messages will be readable, likewise if the server’s key database is captured, all messages, for all hosts on the network, past, present, and future, will be instantly readable by the attacker. There is no forward secrecy, whereby capturing the key for a session, does not reveal anything about past sessions. This scheme is also difficult to scale, and difficult to protect, as the server’s key database becomes the focal point of attack.

What we propose with SKDP, is a symmetric scheme that uses pre-shared keys in a way that does provide forward secrecy, that is scalable, and solves many of the problems associated with existing schemes that use pre-shared keys. In SKDP, capture of a client hosts embedded key, does not compromise past messages, and capturing the server’s key database does not reveal anything about past encrypted messages, because the symmetric ciphers used in the message stream, are keyed with ephemeral keys, that cannot be derived from the pre-shared key alone. The pre-shared keys are used primarily for authentication and encryption of secret tokens, passed between the server and the client, that cannot be derived from either the key database, or a client’s embedded key. The only way to break this system, is to own the master key, which is used instead of a key database, and to capture the entire message stream. This is the same vulnerability in asymmetric based schemes, if the keys are known, it becomes impossible to protect any system.

We believe that using a combination of the SKDP symmetric encryption scheme, and a quantum secure protocol like QSMP, used to introduce new entropy into the system periodically, that this hybrid scheme may offer true long-term security. SKDP is a good candidate for any institutional transaction-based protocol, where for example the embedded key can be stored on a debit or credit card. It is also a strong candidate for a communications system, where a key is distributed on pluggable memory-storage devices, and the device connects to a central communications hub.

SKDP is highly scalable, and can securely manage millions of devices through a single master key. It derives branch keys from the master key, and device keys from the branch key. Any branch can connect to any client on any other branch, so long as they know that branches identification string. Any client on any branch, can establish a communications session, with any other client on any branch, by traversing a tree-like derivation structure. A branch key can be derived by the root key, and any leaf node on any branch, can have its embedded key derived by the branch node. In this way, large networks can be scaled to local institutions, and those institutions can manage a collection of nodes, with all nodes on all branches being accessible to a single root master node.

* 1. **Purpose**

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

1. The symmetric cipher keys for both the send and receive channels, are ephemeral, and use shared secrets for each channel that are unique to each session (forward secrecy).
2. The capture of the devices shared key does not directly 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.

SKDP 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 the symmetric key that host transmits data on. Symmetric 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 authenticates and encrypts a key sent to the client, and the client encrypts and authenticates a key sent to the server. These keys are used to initialize a post-quantum secure symmetric cipher for each channel, which encrypts the communications stream. A strong emphasis has been placed on authentication with SKDP, with the entire key exchange using authentication to guarantee the exchange, and the symmetric stream cipher using KMAC authentication, with additional data parameters (AEAD) that authenticate the SKDP packet headers.

# **Scope**

This document describes the SKDP secure messaging protocol, which is used to establish an encrypted and authenticated duplexed message stream between two hosts. This document describes the complete symmetric key exchange, authentication, and the establishment of an encrypted tunnel. This is a complete specification, describing the cryptographic primitives, the derivation functions, and the complete client to server messaging protocol.

The C reference code is available at https://github.com/Steppenwolfe65/SKDP

**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 SKDP 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 is required but not necessarily obvious, the key-word **SHALL** will be used to indicate compulsory compliance is required to conform to the specification.

## **References**

**3.1 Normative References**

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

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 SKDP C implementation: https://github.com/Steppenwolfe65/SKDP

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

## **Terms and Definitions**

**4.1 RCS**

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

**4.2 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.

**4.3 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.

**4.4 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.

## **Structures**

**5.1 Device Key**

The device key is an internal structure that stores the device derivation key, the expiration time, and the client identity array.

|  |  |  |  |
| --- | --- | --- | --- |
| Parameter | Data Type | Bit Length | Function |
| Expiration | Uint64 | 64 | Validity Check |
| CID | Uint8 array | 128 | Identification |
| DDK | Uint8 array | 256/512 | Derivation Key |

Table 5.1a: 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 and an error returned.

The key identity array is a 16-byte array that uniquely identifies a device key. This identifier can be used to match the key on a branch server. The key identity array, is divided into subsections, 32-bit identification numbers for the master key, branch key, device key, and set instance.

|  |  |  |  |
| --- | --- | --- | --- |
| **Master ID**  **4 bytes** | **Branch ID**  **4 bytes** | **Device ID**  **4 bytes** | **Set ID**  **4 bytes** |

Table 5.1b: The device identity structure.

The master key array, is hashed with a branch and master identification array, to derive the branch key. More than four billion branches may be created from a single master key. The branch key is hashed with the device identification array, as well as the branch and master identification arrays, to derive more than four billion possible device keys. The set identification array, the last four bytes of the key identification array, is the key version counter, the embedded key version, in a set of time-limited keys assigned to the client device.

**5.2 Server Key**

The server key is identical to the client key except for the bit length of the key identification array is ninety-six bits.

|  |  |  |  |
| --- | --- | --- | --- |
| Parameter | Data Type | Bit Length | Function |
| Expiration | Uint64 | 64 | Validity check |
| SID | Uint8 array | 96 | Identification |
| SDK | Uint8 array | 256/512 | Derivation Key |

Table 5.2: The server key structure.

**5.3 Master Key**

The master key is identical to the client and branch keys except for the bit length of the key identification array is sixty-four bits.

|  |  |  |  |
| --- | --- | --- | --- |
| Parameter | Data Type | Bit Length | Function |
| Expiration | Uint64 | 64 | Validity check |
| MID | Uint8 array | 64 | Identification |
| MDK | Uint8 array | 256/512 | Derivation Key |

Table 5.3: The master key structure.

**5.4 Device State**

The client state is an internal structure that contains all the variables required by the SKDP operations. This includes elements copied from the client key structure at initialization, send and receive channels symmetric cipher states, session cookies, packet counters, and flags.

|  |  |  |  |
| --- | --- | --- | --- |
| Data Name | Data Type | Bit Length | Function |
| Expiration | Uint64 | 64 | Validity check |
| DDK | Uint8 array | 256/512 | Derivation Key |
| DSH | Uint8 array | 128 | Session Hash |
| CID | Uint8 array | Variable | Identification |
| SSH | Uint8 array | Variable | Session Cookie |
| RXSEQ | Uint64 | 64 | Packet Counter |
| TXSEQ | Uint64 | 64 | Packet Counter |
| Cipher Receive State | Structure | Variable | Symmetric Decryption |
| Cipher Transmit State | Structure | Variable | Symmetric Encryption |
| 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 server identification parameter.

|  |  |  |  |
| --- | --- | --- | --- |
| Data Name | Data Type | Bit Length | Function |
| Expiration | Uint64 | 64 | Validity check |
| SDK | Uint8 array | 256/512 | Derivation Key |
| DID | Uint8 array | 128 | Identification |
| DSH | Uint8 array | 128 | Session Hash |
| SID | Uint8 array | Variable | Identification |
| SSH | Uint8 array | Variable | Session Cookie |
| RXSEQ | Uint64 | 64 | Packet Counter |
| TXSEQ | Uint64 | 64 | Packet Counter |
| Cipher Receive State | Structure | Variable | Symmetric Decryption |
| Cipher Transmit State | Structure | Variable | Symmetric Encryption |
| 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 SKDP Packet Header**

The SKDP packet header is 13 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 SKDP\_MESSAGE\_MAX in size.

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

Table 5.7: The SKDP packet structure.

This packet structure is used for both the key exchange protocol, and the encrypted tunnel.

**5.8 Flag Types**

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

|  |  |  |
| --- | --- | --- |
| 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 tunnel. |
| Exchange Request | 0x05 | The key-exchange client exchange request flag. |
| Exchange Response | 0x06 | The key-exchange server exchange response flag. |
| Establish Request | 0x07 | The key- exchange client establish request flag. |
| Establish Response | 0x08 | The key- exchange server establish response flag. |
| Establish Verify | 0x09 | The packet contains an establish verify flag. |
| Keep Alive Request | 0x0A | The packet contains a keep alive request. |
| Session Established | 0x0B | The tunnel is in the established state. |
| 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 SKDP:

|  |  |  |
| --- | --- | --- |
| Error Name | Numerical Value | Description |
| None | 0x00 | No error condition was detected. |
| Authentication Failure | 0x01 | The symmetric cipher had an authentication failure. |
| KEX Failure | 0x02 | The KEX authentication has failed. |
| 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. |
| Establish Failure | 0x05 | The transmission failed at the KEX establish phase. |
| Exstart Failure | 0x06 | The transmission failed at the KEX exstart phase. |
| Invalid Input | 0x07 | The expected input was invalid. |
| Keep Alive Expired | 0x08 | The keep alive has expired with no response. |
| Key Expired | 0x09 | The SKDP public key has expired. |
| Key Unrecognized | 0x0A | The key identity is unrecognized. |
| Packet Un-Sequenced | 0x0B | The packet was received out of sequence. |
| Random Failure | 0x0C | The random generator has failed. |
| Receive Failure | 0x0D | The receiver failed at the network layer. |
| Transmit Failure | 0x0E | The transmitter failed at the network layer. |
| Verify Failure | 0x0F | The expected data could not be verified. |
| Unknown Protocol | 0x10 | The protocol string was not recognized. |
| General Failure | 0xFF | The connection experienced an internal error |

Table 5.9: Error type messages.

# **Operational Overview**

In a multi-tiered, distributed topology, a set of branch identification numbers is determined, and the master key is used to create the set of secret branch keys, which are distributed to servers on the network. The servers generate the keys for the client devices associated with each branch, and assign the secret keys to the devices. The method of distribution of secret keys varies with the type of implementation. For example, keys can be imprinted on debit cards issued by financial institutions, or shared through an encrypted channel with equivalent security to a host device.

Each key has an expiration-time parameter. The expiration of keys should be determined by the application of this technology. Some applications, like for example debit cards, may tolerate longer periods, while other applications like a high-security communications link, might be renewed on a much shorter time period. It is recommended that keys are refreshed periodically, this guarantees that in a worst-case scenario, where either the master or branch keys have been captured, that security is continually restored. A strong post-quantum asymmetric encrypted tunnel, like QSMP, can periodically add entropy to a server’s and a device’s embedded key, by mixing new entropy with that embedded key, a new shared secret combined with the base key to generate a new base derivations key. These keys are counted as key revisions, in the key-setbytes of the key identification string.

Any error during the key exchange or during the communications 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 during the key exchange, authentication failures, and internal errors raised by cryptographic or network functions used by the key exchange and communications stream.

* 1. **Connect Request**

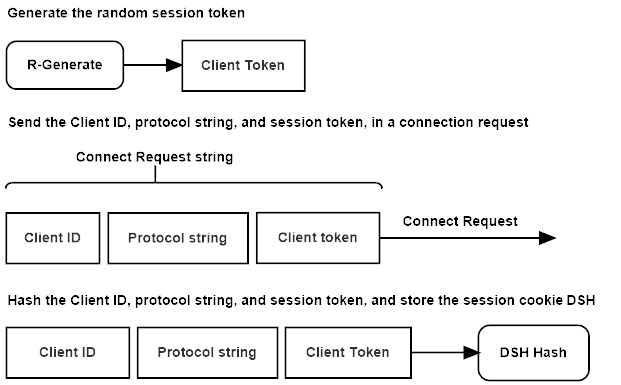


Figure 6.1: SKDP client connect request.

The client device initializes a key exchange operation, by sending the server a **connection request** packet. The message contains the client’s protocol configuration string, key identification array, and a random session token. The client stores a hash of these three values, for use later in the key exchange as the client’s session cookie.

**6.2 Connect Response**

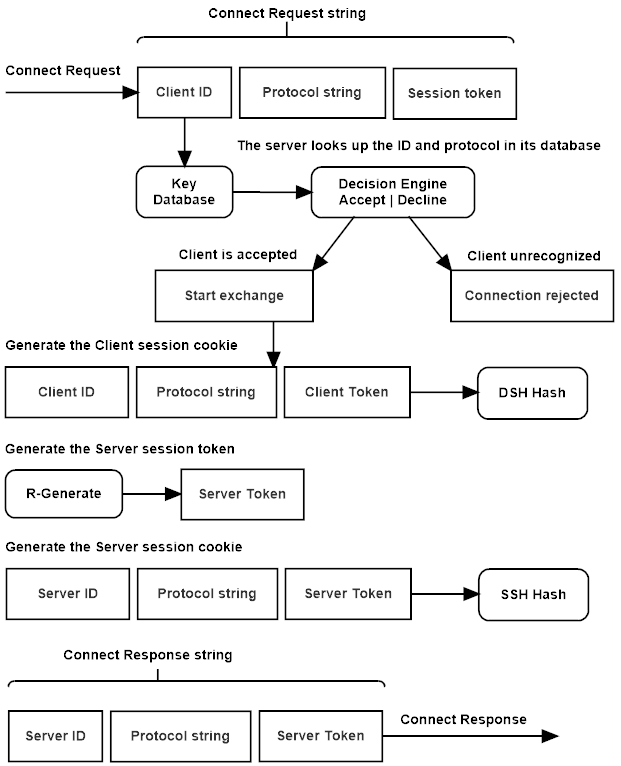


Figure 6.2: SKDP server connect response.

The server receives the **connection request**, checks that the server portion of the key identification array matches its own identity string, and stores the client’s identity string in state. The server compares the client’s protocol configuration with its own, if either the configuration string or key identification do not match, the connection request is rejected, and the client is sent an error notification.

The server stores a hash of the client id, configuration string, and random token, which will be used as the client’s session cookie in the exchange. The server generates a random token, then hashes its own identification array, configuration string, and the random token, and stores this as the server’s session cookie. The server then sends a **connect response** message to the client, containing its own identification array, configuration string, and random token.

**6.3 Exchange Request**

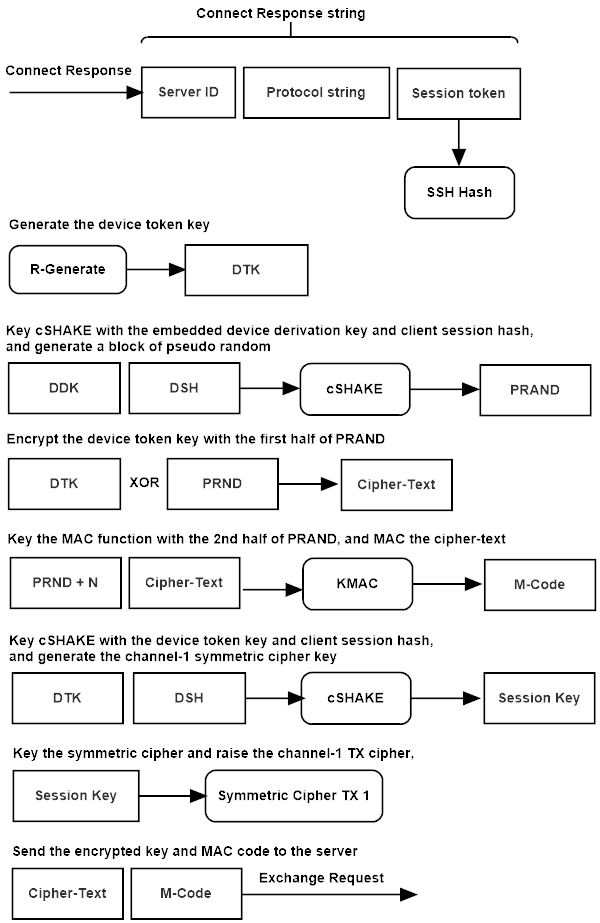


Figure 6.3: SKDP client exchange request.

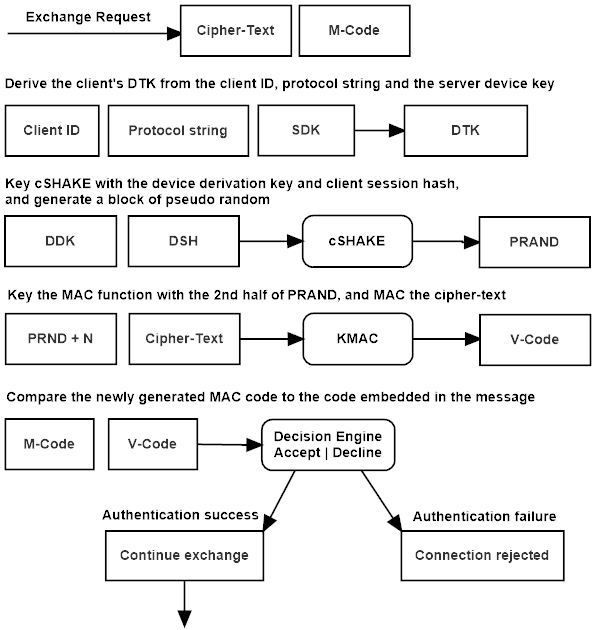
The client receives the **connect response** message from the server, and hashes the server’s identity array, protocol configuration string, and random token to create the server session cookie.

The client generates a random device token-key.

The client combines the device token-key, and the client’s session cookie to key cSHAKE. The client then generates the symmetric stream cipher’s key and nonce from the keyed cSHAKE instance, and initializes the transmit cipher for channel-1.

The client combines its embedded device derivation-key, with the client’s session cookie and keys cSHAKE, to generate a pseudo-random block, used as the token encryption and MAC keys. The client encrypts the random token using a bitwise XOR of the first half of the pseudo-random block, then keys KMAC using the second half of the pseudo-random block, and authenticates the cipher-text, generating the MAC tag. The encrypted session token and MAC tag are added to the **exchange request** message, and sent to the server.

**6.4 Exchange Response**



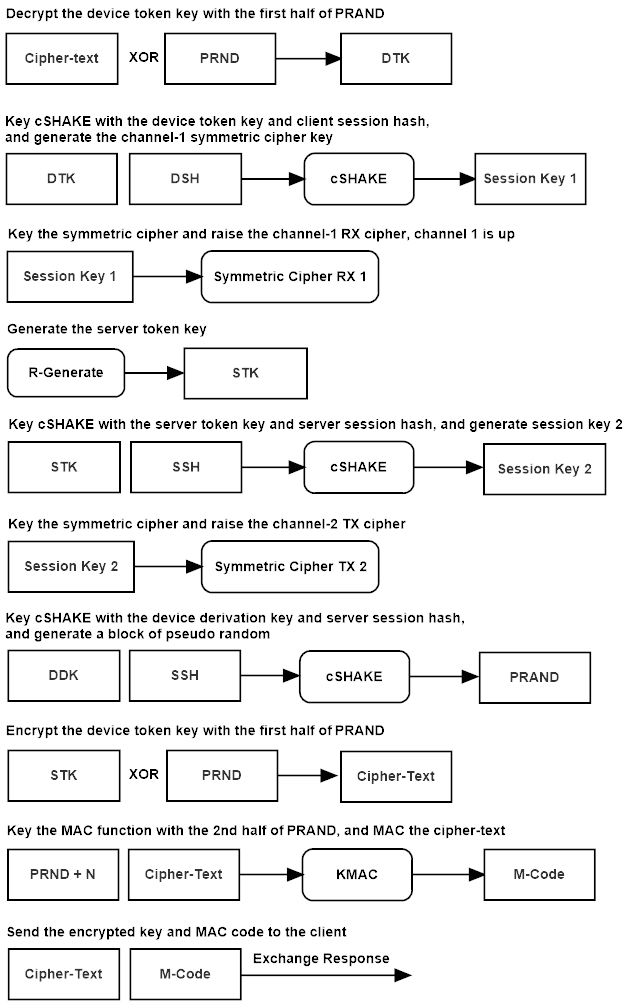


Figure 6.4: SKDP server exchange response.

The server combines the server’s base derivation key, and the client’s key identity array, to key cSHAKE, and derives the client’s device derivation key.

The server combines the device derivation-key, and the client’s session cookie to key cSHAKE, and generates a pseudo-random block, used as the token encryption and MAC keys. The server uses KMAC to authenticate the cipher-text contained in the **exchange request** message sent by the client, and if that authentication succeeds, the server uses the encryption key to decrypt the session token using a bitwise XOR.

The server combines the device token-key, and the device session cookie to key cSHAKE. The server then generates the symmetric stream cipher’s key and nonce, and initializes the receive symmetric cipher instance for channel-1.

The server generates a random session token-key. The server combines the server token-key, and the server session cookie to key cSHAKE, and generates the key and nonce for the server’s channel-2 transmit symmetric stream cipher instance.

The server combines the device derivation-key, and the server’s session cookie to key cSHAKE, which generates the token encryption and MAC keys. The server encrypts the server session token with the encryption key using a bitwise XOR, then keys KMAC with the MAC key and authenticates the cipher-text. The cipher-text and MAC tag are added to the **exchange response** message and sent to the client.

**6.5 Establish Request**

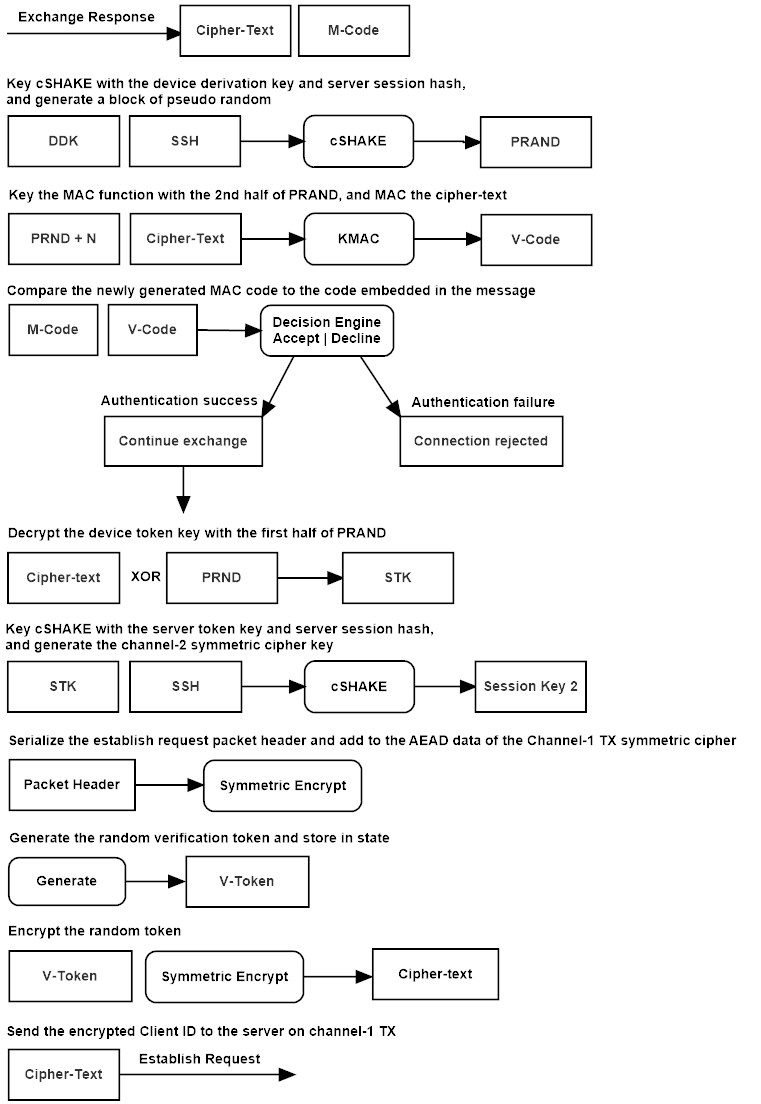


Figure 6.5: SKDP client establish request.

The client combines its device derivation key with the server’s session hash to key cSHAKE, which generates the token MAC and token encryption keys. The client keys KMAC, and authenticates the cipher-text contained in the **exchange response** message sent by the server. If authentication succeeds the client decrypts the server’s secret token using the encryption key and a bitwise XOR of the cipher-text.

The client combines the server’s session token and the server’s session cookie to key cSHAKE, which derives the key and nonce for the channel-2 receive symmetric stream cipher instance.

The client generates a random verification token. The client serializes the establish request packet header and adds it to the additional data parameter of the channel-1 AEAD symmetric stream cipher. The client then encrypts the verification token, and adds the cipher-text to the **establish response** message, and sends it to the server.

**6.6 Establish Response**

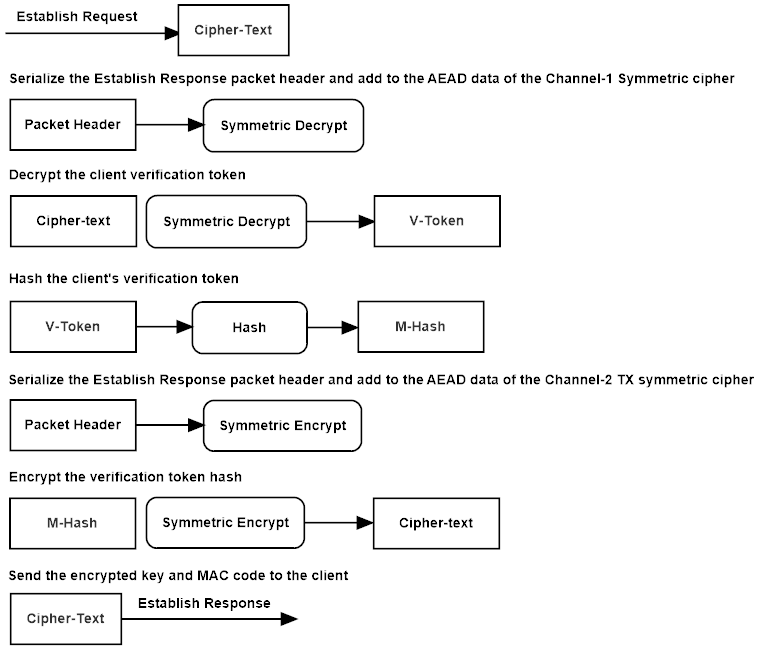


Figure 6.6: SKDP server establish response.

The server serializes the establish request packet header and adds it to the additional data parameter of the AEAD symmetric stream cipher, it then authenticates and decrypts the cipher-text. The server hashes the message, adds the establish response packet header to the cipher’s additional data, and encrypts the message hash. The server then sends the cipher-text back to the client in the establish response packet.

**6.7 Establish Verify**

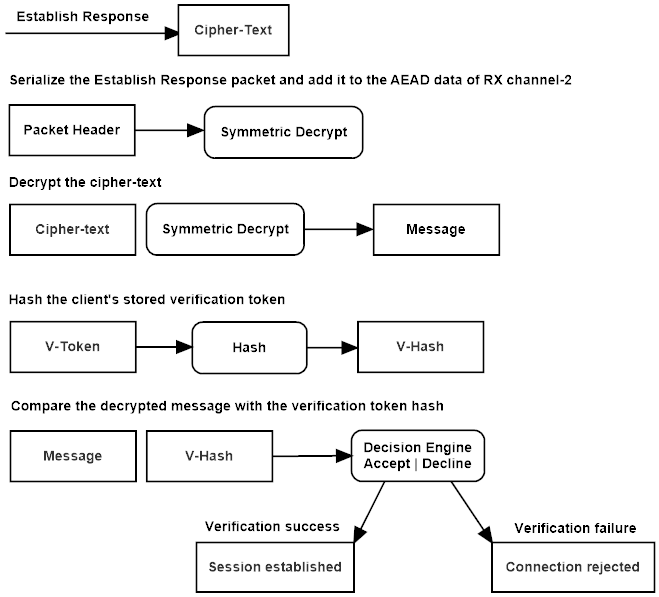


Figure 6.7: SKDP client establish verify.

The client serializes the packet header and adds it to the additional data parameter of the channel-2 receive cipher. The client then authenticates and decrypts the cipher-text sent in the establish response message sent by the server. The client hashes the stored copy of the verification token, and compares it to the message for equality. Upon successful decryption and verification of the message, the client raises its session established flag, and is ready to process data.

# **Formal Description**

**Legend:**

**C** -The client host

**S** -The server host

**cnf** -The protocol configuration string

**cprrx** -A receive channels symmetric cipher instance

**cprtx** -A transmit channels symmetric cipher instance

**ddk** -The device derivation key

**did** -The device identity array

**dsh** -The device session hash

**dtk** -The device token key

**-Ek** -Decrypt using the encryption key

**Ek** -Encrypt using the encryption key

**etk** -The encrypted token

**Exp** -The cryptographic key expansion function

**H** -The hash function

**ke** -The token encryption key

**km** -The token MAC key

**Exp** -The key expansion function: cSHAKE

**M** -The MAC function

**mtag** -The MAC authentication output tag

**RBG** -The random bytes generator

**rtk** -A random token

**sdk** -The servers derivation key

**sid** -The servers identity array

**stk** -The server token

**stokd** -The device session token

**stoks** -The server session token

**Key Exchange Sequence**

**7.1 Connect Request:**

The client sends its identity string, configuration string, and a random token to the server. The client stores a hash of these values in the device session hash.

The client generates a random token.

**stokd = RBG(n)**

The client stores the device-id, configuration string and token in the device session hash.

**dsh = H(did || cnf || stokd)**

**C{ did, cnf, stokd}->S**

**7.2 Connect Response:**

The server responds by either declining the exchange in the event of an error, or signaling the next stage. After verifying that the configuration matches, and that the client-id is known to the server, the server hashes the message and stores a copy of the device session hash.

**dsh = H(did || cnf || stokd)**

The server generates a random token.

**stoks = RBG(n)**

The server stores a hash of the server’s identity, configuration string, and session token in the server session hash.

**ssh = H(sid || cnf || stoks)**

The server sends the id, configuration string, and session token to the client.

**S{ sid, cnf, stoks }->C**

**7.3 Exchange Request:**

The client stores a hash of the server’s configuration string, server-id, and the server session token.

**sth = H(sid || cnf || stoks)**

The client generates a secret random device token key.

**dtk = RBG(n)**

The client combines the device session hash, and its embedded device derivation key to produce the token encryption and mac keys.

**ke,km = Exp(dsh, ddk)**

The client encrypts the secret token, and then macs the cipher-text.

**etk = Eke(dtk)**

**mtag = Mkm(etk)**

The client combines the client’s device session hash and the device token key to produce the channel-1 transmit cipher key, and keys the cipher.

**k,n = Exp(dsh, dtk)**

**cprtx(k,n)**

The client sends the encrypted token and mac-tag to the server

**C{ etk, mtag }->S**

**7.4 Exchange Response:**

The server combines the client’s identity string with the server derivation key, to derive the client’s device derivation key.

**ddk = H(cid || sdk)**

The server combines the device’s session hash, and the device derivation key to produce the token encryption and mac keys.

**ke,km = Exp(dsh, ddk)**

The server verifies the mac code appended to the client message.

**Mkm(etk) = true ? mtag : NULL**

If the mac is verified, the server decrypts the token, and then combines the secret token and the client’s session hash to produce the receive channel-1 cipher key.

**dtk = -Eke(etk)**

**k,n = Exp(dsh, dtk)**

**cprrx(k,n)**

The server generates a secret random token key.

**rtk = RBG(n)**

The server combines the server’s session hash, and the devices derivation key to produce the token encryption and mac keys.

**ke,km = Exp(ssh, ddk)**

The server encrypts the server token-key, and then macs the cipher-text.

**etk = Eke(rtk)**

**mtag = Mkm(etk)**

The server combines the secret token-key and the server’s session hash to

produce the transmit channel cipher key.

**k,n = Exp(ssh, rtk)**

**cprtx(k,n)**

The server sends the encrypted token-key and mac-tag to the client.

**S{ etk, mtag }->C**

**7.5 Establish Request:**

The client combines the server’s session hash, and the device derivation key to produce the token encryption and mac keys.

**ke,km = Exp(ssh, ddk)**

The client verifies the mac code appended to the client message.

**Mkm(etk) = true ? mtag : NULL**

If the mac is verified, the client decrypts the servers token-key, and then combines the server token-key and the server’s session hash to produce the channel-2 receive cipher key.

**stk = -Eke(etk)**

**k,n = Exp(ssh, stk)**

**cprrx(k,n)**

The client generates a random verification token that it stores in state.

**vtok = RBG(*n*)**

It encrypts the verification token and sends the cipher-text to the server.

**cpt = Ek(vtok)**

**C{ cpt }->S**

**7.6 Establish Response:**

The server authenticates and decrypts the message.

**msg = -Ek(cpt)**

The server hashes the decrypted message.

**mhash = H(msg)**

The server encrypts the message hash using the channel-2 cipher, and sends it to the client for verification. Both channels of the server’s communications stream are now initialized.

**cpt = Ek(mhash)**

**S{ cpt }->C**

**7.7 Establish Verify:**

The client authenticates and decrypts the message. Both of the client’s communication channels are established, the connection is now ready to send and receive data.

**msg = -Ek(cpt)**

The client hashes the verification token stored in state, and compares that hash to the decrypted message for equality. If the check is valid, then the tunnel is ready to process data. If the check fails, the client sends and error message to the server, and tears down the connection.

**vhash = H(msg)**

**7.8 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. All of this is done by using the RCS stream cipher’s AEAD and encryption functions.

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 = Ek(m)**

**mc = Mmk(sh, cpt)**

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. If this check fails, the decryption function returns false, returns an empty message array, and must be handled by the application.

**m = -Ek(cpt) = true ? m : NULL**

# **Design Decisions**

SKDP was designed to be flexible and scalable. It can scale to billions of devices using a pyramid hierarchy of client devices connecting to intermediate ‘branch’ servers which can inter-connect through a master server, or it can be used in a single link between two endpoints. It could be implemented on credit or debit cards as an encrypted transport, in removable media to create pluggable lightweight communications channels, or as the encryption protocol used to connect VPN endpoints. The SKDP protocol can be used anywhere a cryptographically-strong, lightweight, post-quantum secure communications channel is required.

SKDP uses Keccak, the NIST SHA3 secure hash and pseudo-random generation functions. These state-of-the-art functions and protocols, that have been studied extensively and are officially recognized as a strong post-quantum resistant family of cryptographic functions.

SKDP can use 256-bit or 512-bit symmetric cipher keys. The authenticated symmetric stream cipher RCS, is based on Rijndael, the symmetric cipher used in AES. It has double the internal block size (Rijndael-256), the transformation function is to the Rijndael specification, but the key schedule, used to generate a large set of ‘round keys’ from a small input cipher key, has been changed from the differentially-weak native expansion function to the strong Keccak cSHAKE function. The number of transformation rounds has been increased from 14 used by AES-256 to 22 rounds, and 30 rounds when using the 512-bit key option. These changes strongly mitigate most attacks against AES, as well as setting the number of transformation rounds to at least ***2n*** the best-known attack. RCS uses KMAC the Keccak MAC function to authenticate cipher-text, making RCS oner of the strongest symmetric ciphers available in the world today.