SILENT CIRCLE ENGINEERING

Silent Circle Instant Messaging Protocol

Protocol Specification

Author: Vinnie Moscaritolo

Date: October 22, 2012

Version: Preliminary 0.10

Table of Contents

Introduction

Silent Circle Instant Messaging Protocol (SCIMP) enables you to have a private conversation over instant message transports such as XMPP (Jabber). You could argue that the world doesn't really need another instant message protocol, but the existing protocols didn't have all of the features we wanted, or had unnecessary complexity. SCIMP draws from a number of related protocols.
In designing SCIMP, we have drawn many ideas from:

- ZRTP: Media Path Key Agreement for Unicast Secure RTP [ZRTP]
- OTR: Off The Record [OTR]
- SSMS: Secure Short Message Service [BELV]
- Cryptocat: The Cryptocat Project [NADM]

SCIMP provides strong encryption, perfect forward secrecy and message authentication. Further, we have incorporated many NIST-approved methods and protocols into its design including:

- Elliptic Curve Diffie–Hellman (ECDH), NIST 800-56A
- Counter with CBC-MAC (CCM), NIST 800-38C
- Key Derivation, NIST 800-108
- Secure Hash Standard, FIPS 180-4
- Advanced Encryption Standard (AES), FIPS 197

While it is not our immediate intent to FIPS-140 validate the SCIMP library, we have experience with that process and the overall design is not incompatible with that goal.

We have also optionally added the Skein family of hash and message authentication function as a cipher suite.

The protocol is placed completely into the public domain, and the implementation code is open source. It is designed and written by Silent Circle, LLC.

High-level SCIMP features

SCIMP has a number of features:

Easily analyzed, easily implemented.
This was important to us, it’s one thing to put our code for open source, but it is next to useless if the code is difficult to analyzed and build.

Relatively few options.
Options are both good and bad in a crypto protocol. If there are too few of them, then the protocol can't change with the times. If there are too many, the protocol is hard to build, test, and vet.

NIST-vetted crypto primitives.
While we don't always want to limit ourselves only to that suite, NIST selection and documentation of crypto primitives is a great place to start, so we did.

Full 128-bit security.
SCIMP is built to have a minimum of 128-bit security through-and-through.

Simple, integrated authentication.
All communications have the problem of how to authenticate to your partner the first time you talk, to reject man-in-the-middle and other eavesdroppers. SCIMP uses simple mechanisms that can integrate with voice communication and share that authentication.

Designed for mobile devices

After examining the popular text encrypting protocols we discovered that they had a number of shortcomings when employed over mobile devices. SCIMP has been designed for use with mobile devices from the beginning.

Basic Protocol Overview

A lot of the SCIMP design derives from ZRTP. The key agreement protocol employs an ephemeral Elliptic Curve Diffie-Hellman (EC-DH) key agreement to establish a shared secret without invoking a trusted third-party. Authenticity is provided by key continuity using hash commitments of a shared secret, similar to ZRTP and optionally a verbal form of user authentication. Together, key continuity and user authentication prevent man-in-the-middle attacks from going unnoticed.

The data messages are protected with authenticated encryption using Counter with Cipher Block Chaining-Message Authentication Code (CCM), as defined in NIST Special Publication SP800-38C.

Key Negotiation

The Key Agreement Protocol for SCIMP establishes an ephemeral shared secret using a minimal set of messages. SCIMP uses the Elliptic Curve Diffie-Hellman (ECDH) primitive for shared secret computation, with key continuity and one-time verbal authentication for man-in-the-middle detection.

SCIMP has the advantage of being completely peer-to-peer. There is no need for third parties and you don’t have to worry about preventing accidental exposure of long term secrets.

SCIMP begins with the initiator sending a hash commitment to the responder on a freshly generated ECDH public key. The responder generates and sends back his public key, without knowing the initiator’s public key.

Because the initiator also generated her public key without knowing the responder’s key, an adversary was prevented from controlling the result of the Diffie-Hellman computation.

Hashes of a cached secret are also included in the first two messages. The cached secret is a bit of key material saved from previous executions of the protocol, if applicable (in the absence of a cached secret a random value is sent). The two parties can use this to determine if they have shared key material that matches. Because someone attempting a man-in-the-middle attack wouldn’t know or have access to this shared key material, it provides both parties a way to identify if someone is attempting to impersonate the other.

The last step of SCIMP involves sending a message authentication code on a known value to the other party, proving that the protocol completed successfully and that the initiator has access to her private key. Once this has been confirmed, both parties refresh their cached secret with a new value derived from the freshly computed shared secret.

This action breaks the advantage of an adversary who may have previously compromised cached secrets. As soon as an authentic key agreement takes place, a new cached secret is generated and secrecy is restored to the protocol.

Once SCIMP has completed, a Short Authentication String (SAS) is displayed to the user to verify the absence of a man-in-the-middle attack. Due to key continuity, the user may verify the SAS with the other party at any time. Using a phone to contact the other party, for example, the user can gain confidence in the identity of the other party based on a number of clues using standard human interaction. An attacker would need to duplicate all these clues when the two parties were checking the SAS and remain undetected while leaving the rest of the conversation undistorted in order to be successful.

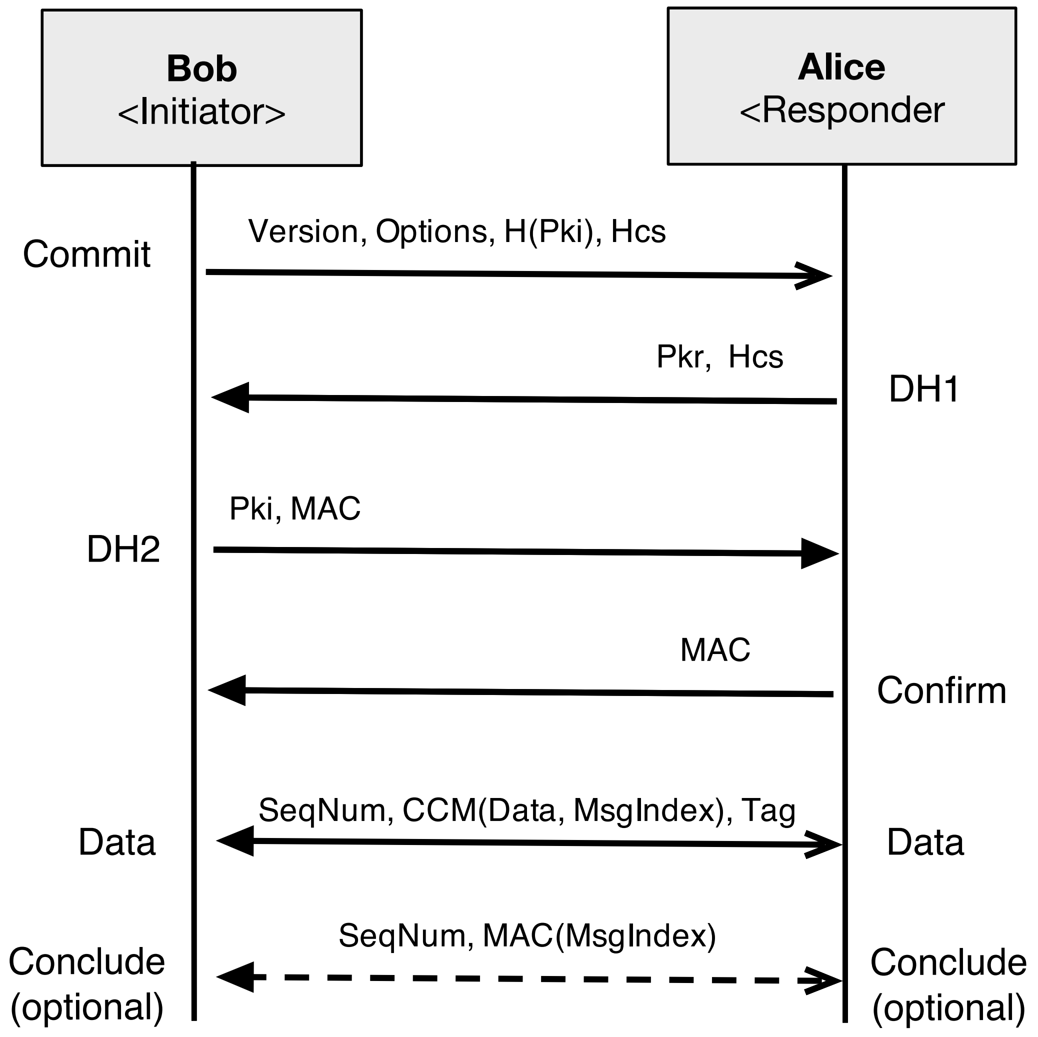


Figure 1: SCIMP Protocol Diagram

Algorithms and Ciphers

The SCIMP initiator specifies what suite of algorithms to use in the commit message. If the responder does not support the algorithms requested, the responder ignores that commit message. The responder can send his own commit message, specifying a different suite of algorithms to use, starting the process over. Table 1 describes the options and their meanings.

Suite	Hash	KDF/MAC	Cipher	Public Key
1	SHA-256	HMAC/SHA-256	AES-128	ECC-384
2	SHA-512/256	HMAC/SHA-512	AES-256	ECC-384
3	SKEIN-512/256	SKEIN-MAC-512	AES-256	ECC-384

Table 1: SCIMP Cipher Suites

Hash Commitment

The hash commitment forces the adversary to select a public key without knowing the other party's key. This restricts the adversary to one attempt at finding a collision in the short authentication string (SAS), thus allowing the SAS to be much shorter than it otherwise would need to be. Without the hash commitment, the adversary could acquire the public keys for his two victims before searching for his own pair of keys that would result in an SAS collision. Such a collision would allow the adversary to set up two agreements that generated the same SAS string on both victim devices for a successful and undetected man-in-the-middle attack.

This attack is restricted in SCIMP by forcing the adversary to select a public key prior to knowing the public key of the other party. Without the other key, the adversary cannot predict what the SAS value will be, and is forced to make a single blind guess.

The hash commitment *Hc* is checked when the initiator sends his public key in DH2 (see Figure 1). If the check fails, a man-in-the-middle attack is in progress. The user is warned, and the protocol stopped. The hash commitment is performed on the *Pki* as it is sent in the DH2 message, using network byte order with BER encoding.

$$Hc = H(Pki)\ (512\ \text{bits}) \qquad (1.1)$$

Commit Contention

Two end points may attempt to initiate a key negotiation at the same time. Each end point may send the other a commit message before receiving the commit message of the former.

The protocol will flag an error and it is up to the application to decide what to do.
One way this contest can broken is to compare the hash values of the hash commitment in big endian integer format, and discarding the message with the lower value. The side that sent the commit with the higher value becomes the initiator and the other side, the responder.

Cached Secret Comparison

Each device stores a long term cached secret for each party it has successfully executed the protocol with in the past. These cached secrets are tied into the key derivation function to achieve key continuity. Key continuity gives us confidence that the user with which are communicating today is the same person with which we communicated last week. Key continuity has been used in a number of protocols including SSH[Gut08].

To determine if two parties have cached secrets (cs), each party sends a non-invertible hash of their cached secret to the other. The initiator does this in the commit message, and the responder includes it in the DH1 message. If no cached secret is available, a random value is substituted for *cs* to avoid leaking information about the cache state. *Hcsi* is sent in the commit message and *Hcsr* is sent in DH1. To further avoid leaking information about the state of the cached secret, the MAC is computed on a salt available in the commit and DH1 messages.

$$Hcsi = MAC(cs, H(Pki) \| \text{“Initiator”})\ (\text{First 64 bits}) \tag{1.2}$$

$$Hcsr = MAC(cs, H(Pkr) \| \text{“Responder”})\ (\text{First 64 bits}) \tag{1.3}$$

These values are computed locally and compared against the values received from the other party to determine the presence of a cached secret. If the hashes match, the cached secret exists and is shared between both parties. If they do not match, a null is used in place of *cs* in the key derivation function.

If a device stores a cached secret for a particular party, the cached secret comparison is expected to succeed. If the comparison fails when a cached secret is available, then one of the parties may have lost their secrets due to a device reset, or there may be a man-in-the-middle (MiTM) attack in progress. In the case of a mismatch, the user must be warned that a MiTM attack may be underway, and advise the user to verify the short authentication string as soon as possible to verify that the MiTM is not present.

If a mismatch occurs, the cache is not updated until after the user has verified the short authentication string (SAS) with the other party. The user should be warned of this condition on every key agreement until the condition has been resolved.

When no attacker is suspected, the cached secret cache is updated with a new value after a successful key agreement has been confirmed by receipt of confirmation MACs

$$cs = KDF(Z, \text{“RetainedSecret”}, Context, 256) \tag{1.4}$$

Because previous cached secret key material is mixed into the key derivation, a successful attack on key negotiation must involve both a compromise of the cached secrets on the device as well as a man-in-the-middle attack on the next key negotiation. If the adversary misses just one key negotiation, the cached secrets are replaced with fresh, secret values and the protocol self-heals.

Short Authentication String (SAS)

The Short Authentication String (SAS) is the first 20 bits of the SAS hash. The commit message includes a designation of what SAS rendering scheme to use:

- **0x01** The SAS is displayed to the user as a 4 character Base32 string using the encoding described in section 5.1.6 of RFC 6189[ZJC11]. That encoding scheme is designed for ease of human use. The output alphabet is all lower case, excludes characters that are easily confused, and padding characters are left out.

- **0x02** The 4 characters of the prior scheme are displayed to the user using the NATO phonetic alphabet as an aid to auditory clarity in verbal confirmation. If the end-point interface is not large enough for NATO words, however, this option downgrades to the first option.

- **0x03** The SAS is displayed to the user as 6 hexadecimal letters

$$Ksas = KDF(Kdk2, \text{“SAS”}, Context, 20) \qquad (1.5)$$

Silent Circle recommends that users make use of an alternative method to establish the identity of the receiving party and verify that the user has the same SAS value. A phone call would be sufficient for this purpose since confidence building cues such as voice timbre and manner of speech are present. These cues make it difficult for an adversary to convincingly impersonate the other party without being detected. If the two parties are physically co-located, they may even be able to compare their short authentication strings by placing their devices side-by-side.

Because the verification of the SAS cannot be automated, the security of SCIMP is dependent on the initiative of the user. This opens a window of vulnerability for lackadaisical users as an adversary may successfully perform a man-in-the-middle attack without detection until the first comparison of the SAS.

This vulnerability is offset by several considerations. The first is that the key continuity properties require the presence of the adversary during every key agreement. An absence during just one key negotiation would alert the user that key continuity has been broken, and subsequent key negotiations would be secure against undetected man-in-the-middle attacks. The second implication from key continuity is that a successful verification of the SAS means that all previous key agreements have succeeded without interference from the adversary.

Partial Public Key Validation

Before using the public keys in DH1 and DH2 for computation, the keys must be verified. Public keys must have the correct modulus and not be the point at infinity. Full validation would also check that the public key is in the same subgroup as our ECC domain parameter curve, but this is an expensive operation for resource constrained devices.

The validation routing is performed as specified in 5.6.2.6 of NIST Special Publication 800- 56A[BJS07]. If any of the checks fail, the user should be alerted that a weak key attack is under way, and the protocol must be terminated.
1. Q must not be the point at infinity.
2. Q must be in the valid range of the elliptic curve group.

Key Agreement

The shared secret is computed using the process described in section 6.1.2.2 of NIST Special Publication 800-56A[BJS07].

Once the public key from the other party has been verified, the Elliptic Curve Cryptography Cofactor Diffie-Hellman (ECC CDH) Primitive is used to compute the shared secret Z as specified in section 5.7.1.2 of NIST SP 800-56A.
Given (*q*, *F R*, *a*, *b*, *G*, *n*, *h*)1 as domain parameters, *dA*, one's own private key and *QB*, the other party's public key, compute *Z*:

$$P = hd_AQ_B \qquad (3.8)$$

$$Z = X_p \text{ (where } X_p \text{ is the x coordinate of } P\text{)} \qquad (3.9)$$

The next task is to convert *Z*, which is a field element, into a random bit string suitable for keying material. A naive approach might be to apply a hash function to *Z*, but this is problematic for two reasons: Hash functions are not guaranteed to have Pseudo-Random Function (PRF) properties, only collision and pre-image resistance. Secondly, hash functions need to be computed on random values in order to guarantee the randomness of their outputs. If the input is not random, the output of a static hash function can be easily reversed. Therefore, Krawczyk[Kra10] and a draft Special Publication 800-56C[Che10] from NIST separate randomness extraction and key expansion into two separate steps.

Extract

Z is transformed into a random key derivation key Kdk using a MAC keyed with a salt value. KAPS uses a hash of all the messages sent and received Htotal for the salt.

Htotal = H(commit || DH1 || Pki) - 256 bit hash (1.10)

Kdk = MAC(Htotal,Z) - where Z is the DH of Pki and PKr (1.11)

Enhance

Before the key derivation key is ready to be expanded for application specific purposes, the cached secret must be mixed in. The mixing is performed using the same function as the expand step, with cached secrets being used in the context field.

Kdk2 =MAC(Kdk,1 || "MasterSecret" ||0x00 || MasterContext || 256) (1.12)

$$MasterContext = AlgorithmID \,||\, InitInfo \,||\, RespInfo \,||\, SuppPubInfo \,||\, SuppPrivInfo \qquad (1.13)$$

Where *AlgorithmID* describes the algorithm: “SCimp-ENHANCE”
InitInfo is identifying information for the initiator:
strlen(Initiator’s JID) II (Initiator’s JID)

RespInfo is identifying information for the responder
strlen(responder’s JID) II (responder’s JID)

SuppPubInfo is a nonce for which Htotal will suffice.

SuppPrivInfo is the cached secret (if it exists):
len(cs) II cs. If no cached secret exists for this agreement, the length of cs is 0 followed by nothing: 0

Expand

Finally, *Kdk2* is expanded using the *KDF* for SCIMP to generate a separate master session key for each direction of the secure symmetric conversation. The context in this case is a simplified version of the master context.

$$KItoR = KDF(Kdk2, \text{“InitiatorMasterKey”}, Context, 256) \qquad (3.14)$$

$$KRtoI = KDF(Kdk2, \text{“ResponderMasterKey”}, Context, 256) \qquad (3.15)$$

The SCIMP key derivation function is defined as the *L* left most bits of the MAC computed in the following way, where *L* must not exceed the size of the MAC function’s output:

$$KDF(K, Label, Context, L) = MAC(K, Counter||Label||0x00||Context||L) \qquad (3.16)$$

K is a secret random bit string.

L is the length in bits of the output key material encoded as a 4 octet integer. Label identifies the purpose of the output key material.

Counter is required by NIST Special Publication 800-108[Che09] and is always 1 since we limit the output length of the KDF to be less than the output length of the MAC. The counter is encoded as a 4 octet integer.

Context is a binary string that ties the output key material to the particular situation in which it is being used.

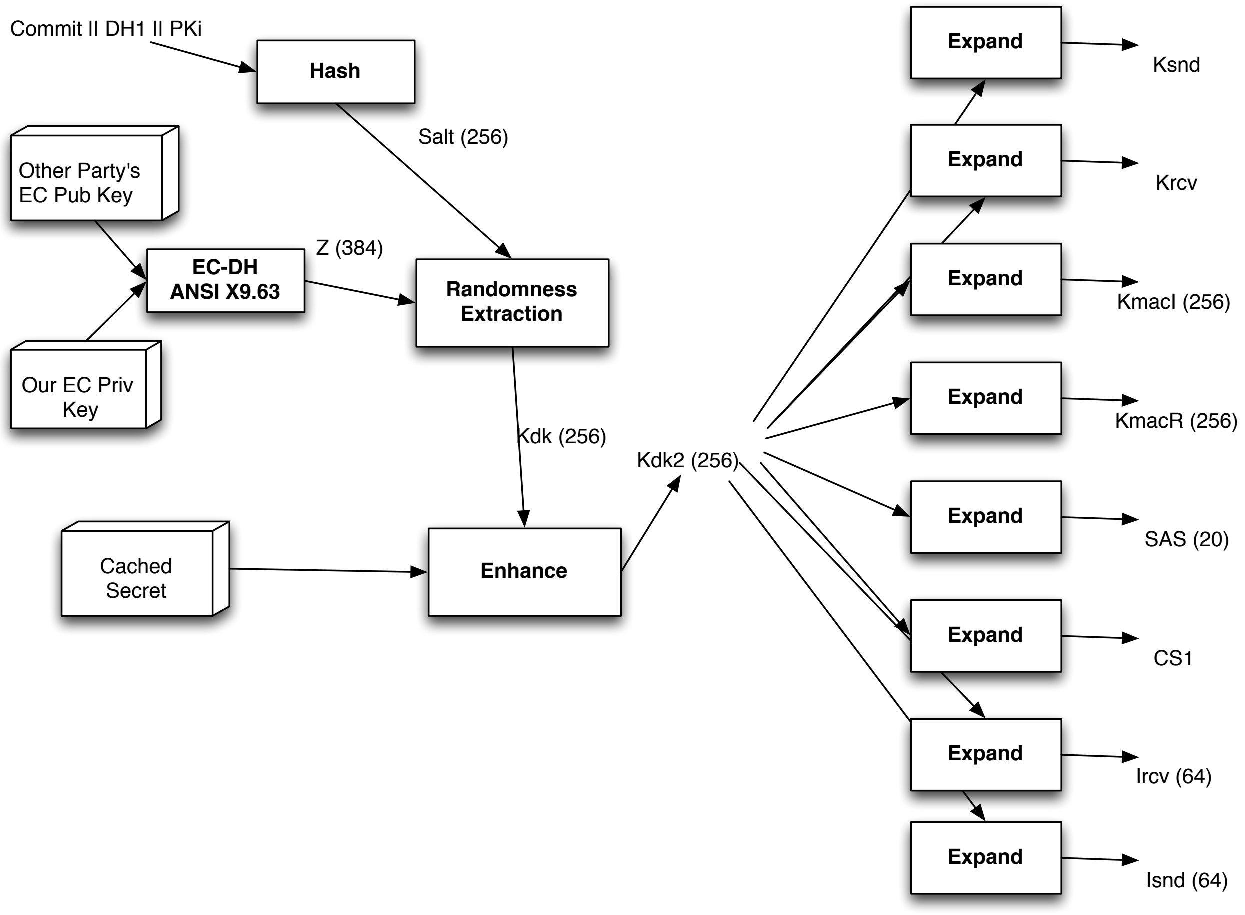
Context = InitiatorJIDInfo | ResponderJIDInfo | SupPubInfo

InitiatorJIDInfo is identifying information for the initiator: len(Initiator JID)| Initiator JID.

ResponderJIDInfo is identifying information for the initiator: len(Responder JID)| Responder JID

SuppPubInfo is a nonce for which Htotal will suffice.

As soon as intermediate keys *Kdk*, *Kdk2*, and other intermediate key material is no longer needed, or if there is an error they must be erased from memory.



Key Confirmation

The last step in the protocol involves giving the other party assurance that the protocol completed successfully and that he is in possession of the correct symmetric key. This is accomplished by sending a MAC computed on a known value under a key derived from *Kdk2*. The initiator's confirmation is sent in the DH2 message, and the responder's confirmation is sent in the Confirm message.

$$KmacI = KDF(Kdk2, \text{“InitiatorMACKey”}, Context, cipherLen) \tag{3.18}$$

$$KmacR = KDF(Kdk2, \text{“ResponderMACKey”}, Context, cipherLen) \tag{3.19}$$

$$maci = MAC(KmacI, Htotal)\ (\text{first 64 bits}) \tag{3.20}$$

$$macr = MAC(KmacR, Htotal)\ (\text{first 64 bits}) \tag{3.21}$$

After the key confirmation has been successfully validated, parties update their cached shared secret cs. (Provided a man in the middle attack was not suspected as noted in section 3.7) The previous shared secret is erased and replaced with the new shared secret as calculated in equation

Message Encryption

SCIMP encrypts the actual message payload using AES in CCM block cipher mode. The CCM mode of operation defines an authenticated encryption scheme, the security of which rests on a single cryptographic primitive, the block cipher. CCM is also attractive due to its ability to authenticate cleartext header data in addition to cipher-text data.

Each message is encrypted with a distinct key. The actual key material is split in half and the upper is used as the key while the lower bits are used for the Initialization Vector. A sequence number is also fed to the CCM algorithm to limit replay protection.

Initial Message Index

SCIMP uses the *KDF* to compute the initial sequence number for the CCM encryption process. Since this function is only called once for any master key, the static nonce is appropriate.

sessionId = H(strlen(initiator JID) ll (initiator JID) ll strlen(responder's JID) ll (responder's JID))

iSnd = KDF(Kmaster , "InitiatorInitialIndex", sessionId ||0, 64)

iRcv = KDF(Kmaster , "InitiatorInitialIndex", sessionId ||0, 64)

Each addition index number is an increment of the previous.

iSnd1 = *iSnd* + 1;

Forward Secrecy

SCIMP uses distinct keys to encrypt each message and in each direction. The initial *Ksnd* and *Krcv* keys are created from the expansion process as described above. SCIMP then derives further key material using a hash based key derivation function that is compliant with NIST Special Publication 800-108[Che09] section 5.1.

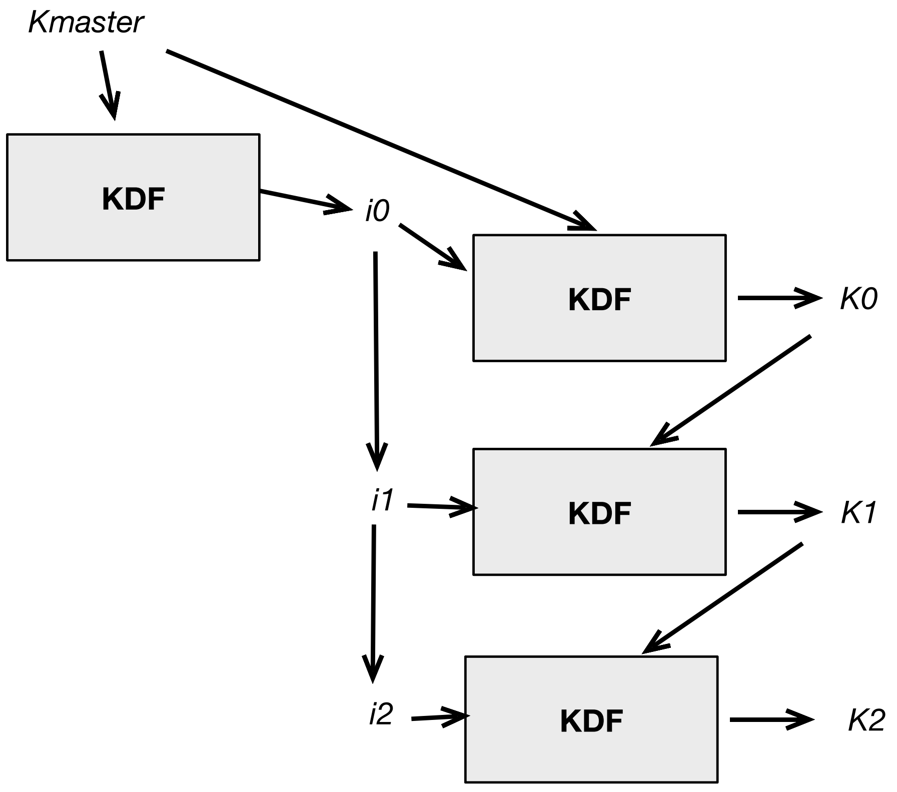
Since each key is computed from the previous using the non-invertible key derivation function the result is a chain of keys that protects the security of prior messages when a single key is compromised. Keys are also erased from memory as soon as they are no longer needed to prevent unneeded leakage in the event that a device is forensically analyzed.

Each key is derived by hashing the previous key with the session identifier as well as the message index. The initial message index is computed from the key expansion process

K0 = KDF(Kmaster,"MessageKey",session identifier||i0)

Kn = KDF(Kn−1, "MessageKey", session identifier||iN)

Where i0 and iN correspond to the appropriate index for that direction (iSnd and iRcv).



SCIMP Message Key Derivation

Padding and CCM

The data passed to CCM is padded. Messages shorter than 32 bytes are padded to 32 bytes and messages longer than 32 are padded to the next multiple of 16. The actual byte used to pad is the number of bytes added to pad. For example a message of 9 bytes long will have an addition 21 bytes of 0x15 added.

Example Message Processing

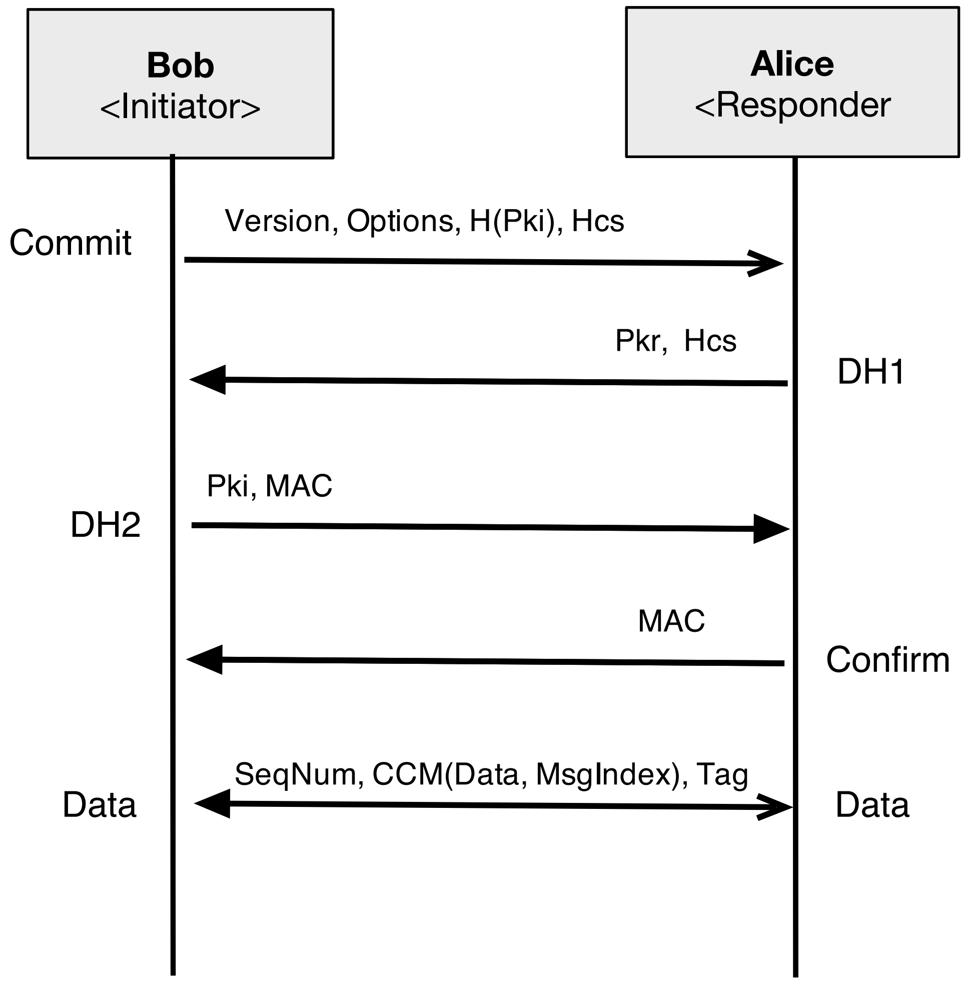
1. Bob generates a ECDH key pair and computes and send the hash Commit Packet.

2. Alice Receives the hash commitment from the initiator and generates her own ECDH key pair, sends the public key in a DH1 message

3. Bob receives DH1 message. Verifies that shared secrets match as expected. If not, warn user of a man-in-the-middle attack. Verifies that the responder's public key is valid using the algorithm specified Computes master and session keys. Sends DH2 with confirmation MAC.

4. Alice Receives the initiator's DH2 public key. Verifies the hash commitment received earlier. Verifies that the shared secrets match as expected. If not, warn the user of a man-in-the-middle attack. Verifies the initiator's public key. Compute master and session keys. Verifies the confirmation code. Update shared secrets. Sends her confirmation code in a confirm packet

5. Bob receives and verifies the confirmation code. Updates shared secrets. and is read to send messages.


Bob
<Initiator>
Alice
<Responder
Commit
Version, Options, H(Pki), Hcs
Pkr, Hcs
DH1
DH2
Pki, MAC
MAC
Confirm
Data
SeqNum, CCM(Data, MsgIndex), Tag
Data

Protocol Details

SCIMP can be transported over a variety of wire formats. SCIMP support JSON, XML or a more compact binary format.

The version of SCIMP the Silent Circle uses encodes the protocol as JSON packets wrapped in a Rad64. AS mentioned above SCIMP uses 4 kinds of packets to establish the secret keys: commit, dh1, dh2 and confirm.

Commit is the start of the keying process, where the initiator creates a ECC key pair but only send a hash of the public key, along with some HMAC information that might indicate if they have had a shared secret in the past.

```
{
    "commit": {
        "version": 1,
        "cipherSuite": 1,
        "sasMethod": 1,
        "Hpki": "s18C+pKU2b81vysPfnRsviMieziZ5i0YrbQXpkdzgSo=",
        "Hcs": "bx/ScjJQU1s="
    }
}
```

When the recipient receives a Commit, it check packet for valid options and then creates it's own ECC key and responds by sending the public key back to the initiator with a DH1 packet. It also includes a HMAC of a previously shared secret.

```
{
    "dh1": {
        "PKr":
"MGwDAgcAAgEwAjEAsI23okNBcGV1qy0RsliSKBpQNf3095weuQXsfcur4s0r9+hpA3jlg75RIcTYiPByAjAto
kuW22OFhAb63ZiDmdE3an06aGJaCT4Ywmy3ywYWDJrkIrkIVcHAdxfupjoN/A0=",
        "Hcs": "Sgd9nFcLvYs="
    }
}
```

At this point the Initiator has enough information to complete the ECC - Diffie Hellman process, and derives both the sending and receive communication keys. The initiator then replies with a DH2 packet which contains a copy of the public key it previously send a hash of and a HMAC of the current shared secret.

```
{
    "dh2": {
        "PKi": "MG0DAgcAAgEwAjEAmS2t3VvYXkSYLWuZ4vDQfJpL2VrCxOrkj2E4v/
6EOxfk9USrdLnVUIhHbSmVpD5vAjEA93WVAfM9STK3zPV2M4NuJlcrGlvCD/
CVhClxtpua6lEBHi5E5bPOTrFszlmrF7qd",
        "maci": "KeO29YxeB80="
    }
}
```

The recipient uses the DH2 information to complete it's ECC - Diffie Hellman process and derives a copy of the communications keys also. To ensure that both sides are on the same page, the recipient replies with a CONFIRM packet which contains an HMAC of the new shared secret.

```
{
    "confirm": {
        "macr": "/jo+3Bz8dPw="
    }
}
```

Once a the keys are established, SCIMP can then send the user message using a DATA packet. The actual message is encrypted using AES in the Counter with CBC-MAC (CCM) mode. This provides us a message authentication code of both the message and the sequence number.

```
{
    "data": {
        "seq": 51323,
        "mac": "ykRzxqvCR4lPj2/yL38C+Q==",
        "msg": "s8o4Ad8qn45uXauVAPAWFQ16Ns2jeV0D3ADYGreCNXW6STs6IW/
dx8Om6VAUgpuKVRg4WPBGUHvbqqv91AQ/Sw=="
    }
}
```

In order to ensure that the messages arrive intact over protocols such as XMPP the JSON packets are encoded in rad64 with a header or "?SCIMP:" and terminated with a period. For example when the last data packet goes over the XMPP, it actually looks like.

Example SCIMP Message

```
<message type="chat"from='velma@silentcircle.com' to='daphne@silentcircle.com'
id="0FF6CF98-32FE-4EED-9DEF-D66A0E50EA8F"><body/><x xmlns="http://
silentcircle.com">?
SCIMP:ewogICAgImRhdGEiOiB7CiAgICAgICAgInNlcSI6IDE1MDcyLAogICAgICAgICJtYWMiOiAiZlp
YYURlQ1ljVTA9IiwKICAgICAgICAibXNnIjogIkloT051Sm9kK0Fjb09KQ1prZ0xHQXliSmJjbC9WNzhl
cmMrSFY4K1FHcUJ2cEdlb2RaSWZwNTRKVWluU2g0N0lZTjFORkJOaXBjTVdubWlsMXVtbi9pcG5rVk8rd
VJZdUJuQjdpZXZEK1pZQzBYV0hHQWQ3WWJtOWRsYkpSd0oyIgogICAgfQp9Cg==.</x></message>
```

Appendix A: Document History

Date	Rev	Author	Change
6/1/12	0.1	vin	First complete draft.
6/5/12	0.2	vin	Updated API names to start with SCimp
6/19/12	0.3	vin	exchange CCM for GCM
6/20/12	0.4	vin	SCIMP messages are Rad64 with SCIMP header
6/21/12	0.5	vin	minor protocol text corrections
7/16/12	0.6	vin	Added message ID to SCimpProcessPacket()
8/21/12	0.7	vin	hTotal needed to be a 256 bit hash
8/31/12	0.8	vin	minor corrections to hash lengths
10/19/12	0.9	vin	cleanup and split API into separate doc
10/22/12	0.10	vin	corrections, kudo JimB

References.

[BELV] Gary Belvin. A Secure Text Messaging Protocol, John Hopkins University. (May 2001)

[CER] CERT. MSC10-J. Limit the lifetime of sensitive data. http://bit.ly/mwdO26.

[Che09] Lily Chen. NIST Special Publication 800-108 Recommendation for Key Derivation Using Pseudorandom Functions (revised). (October), 2009.

[Che10] Lily Chen. DRAFT NIST Special Publication 800-56C Recommendation for Key Derivation through Extraction-then-Expansion. (September), 2010.

[Gut08] Gutmann. Key Management Through Key Continuity(KCM). Internet Draft, 2008.

[Jak] J. Jonsson, On the Security of CTR + CBC-MAC, in Proceedings of Selected Areas in Cryptography – SAC, 2002, K. Nyberg, H. Heys, Eds., Lecture Notes in Computer Science, Vol. 2595, pp. 76-93, Berlin: Springer, 2002.

[800-38B] Draft NIST Special Publication 800-38B, Recommendation for Block Cipher Modes of Operation: the CMAC Authentication Mode. U.S. DoC/NIST, October 2003. Available at http://csrc.nist.gov/CryptoToolkit/modes.

[Krh09] Jan Krhovjak. Cryptographic random and pseudorandom data generators. PhD thesis, Masaryk University, 2009.

[NADM] Nadim Kobeissi , CryptoCat, (http://project.crypto.cat).

[OTR] Ian Goldberg,Nikita Borisov Off-the-Record Messaging Protocol version 2. (http://www.cypherpunks.ca/otr/).

[RAD64] Wikipedia, Base64

[ZRTP (RFC 6189)] P. Zimmermann, A. Johnston, and J. Callas. ZRTP: Media Path Key Agreement for Unicast Secure RTP. RFC 6189, April 2011. (http://tools.ietf.org/html/rfc6189)