Storm4 Cryptography Overview
4th-A Technologies, LLC

Author: Vinnie Moscaritolo

Date: June 9, 2017

Version: Preliminary 1.0

Table of Contents

Introduction

This document is a high level overview of how the Storm4 application and server architecture uses cryptography to secure its documents and associated data. We describe our overall goals, the rationale behind the algorithms we chose, a little bit about how the data is structured and also some possible vulnerabilities as well as future possibilities worth exploring.

Goals and Principles

The Storm4 system was designed with some basic principles in mind.

- Always use strong cryptography and best security practices.
- The user has full control over who can decrypt documents and metadata.
- The storage provider does not have the ability to decrypt the content.
- Make it easy for the user to do the correct thing.

Using these principles we designed Storm4 from the ground up to protect the privacy of the user's content. We use only strong cryptography, and have made our cryptography library, S4, open source and available for inspection. We were very careful about selecting algorithms and chose them based on our experience and the advice of highly experienced cryptographic experts from both academia and industry.

Whenever possible Storm4 uses the following algorithms to protect its content.

- Bernstein/Lange ECC Curve41417 [1]
- TwoFish-256 symmetric key block cipher [2]
- ThreeFish-512 symmetric key tweak-able block cipher [3]
- Skein-256 hash and HMAC [4]
- AES-256 and SHA-256

Why we chose these algorithms

The selection of algorithms used in Storm4 is worthy of discussion. We made a conscious decision to deviate from the standard NIST FIPS-140 approved algorithms for a number of reasons. The primary reason is their possible susceptibility to known and theoretical attacks especially side channel attacks based on cache timing as well as industry concern about weakness in some of these algorithms.

In the subject of public keys we chose to go with elliptic curve cryptography because of the advantages provided on mobile platforms where computing power, memory and battery life are limited.

While NIST curve P-384 was specified in NSA Suite-B Cryptography, there was some concern about *intentional vulnerabilities*[5] in the curve parameters. A safer alternative was Curve41417 as described by Daniel J. Bernstein, Tanja Lange[6].

For file encryption we were seeking some form of disk cipher that has the characteristic of accepting an additional value for each block offset. We wanted to avoid a cipher in chaining mode that would require us to decrypt every previous block to read or make changes to an arbitrary block. This is especially useful for scrubbing into video or music content, saving load time, bandwidth and power usage.

Further we wanted something that was relatively simple and could perform well in the limited computing environment of mobile devices. We looked at a variety of options including calculating an initial vector from the offset. We found that ThreeFish-512 turned out to be very attractive both for its simplicity and because it had a number of provable secure properties. This added substantial confidence to the algorithm.

We chose the Skein hash function family. It is based on the ThreeFish cipher. Skein is incredibly efficient and power conserving making it one of best hashes for mobile devices.

We also chose TwoFish for general block cipher uses. TwoFish was one of the 5 Advanced Encryption Standard (AES) finalists, and although it was not selected, it was a good choice as a NonNIST algorithm.

In addition we employ SHA-256 as a hash when signing our public keys, for server upload, primarily because this algorithm was already available in our server environment. As with all our crypto, we tag the type of algorithm used and could easily swap it out later.

The encryption algorithm AES-256 in CBC mode is used in our iOS and macOS client by the underlying database library SQLCipher.

S4 Crypto Library

Storm4 is based on the open source S4 Crypto Library[7]. We specifically wrote the S4 library for Storm4 and made it available for inspection and peer review. S4 consists of an extensive set of modern cryptographic functions and is designed to be compiled on a variety of platforms. S4 allows the programmer to make high level C calls without having to have expertise in the low level cryptography algorithms and presents the interface in a consistent usable structure. The API was designed with object oriented principles and can easily be wrapped by most object orient languages like Java or Swift.

S4 is layered on top of LibTomCrypt, but we have enhanced it with the Bernstein/Lange Curve4141, Skein hash functions as well as the TwoFish and ThreeFish ciphers. S4 was designed to be portable, and can be cross-compiled for different architectures, including macOS, iOS, Linux, Android, and Windows.

The designers of S4 have lots of experience in building cryptographic libraries and understand the importance of testing and validation. The library was designed with the intent of undergoing a FIPS-140 or similar validation.

While a number of algorithms available in S4 are not FIPS approved, their implementations adhere to the same guidelines we would apply to any FIPS-140 approved algorithm:

- All access to Critical Security Parameters (CSPs) can only be made through the API calls and all such CSPs cannot leave the library unencrypted.
- All CSPs are zeroized upon freeing any object contexts.
- S4 includes an extensive set of operational tests to verify the validity of every API call as well as test vectors to perform both know-answer tests, as well as pairwise consistency tests of the various encryption and hashing algorithms as appropriate.
- The Xcode version of S4 is configured to run the operational test using XCTest framework as part of integration process.
- The library is cryptographically signed by the developer's key before deployment.

Encryption in the Cloud

Each Storm4 document in the cloud is protected with a separate, randomly generated, 512-bit key. Using the ThreeFish symmetric-key tweak-able block cipher we encrypt the files and any associated metadata, thumbnails and attributes. Even the filename is encrypted and never exposed. Similarly the file objects names in the cloud are created by a cryptographic pseudo random number generator (PRNG).

The Storm4 file encryption is performed independently of, and often layered on, any encryption or protection implemented by the cloud provider. Unless specifically exported by the user, (printing, photo album, email, etc), unencrypted data never leaves the device through the Storm4 system.

The 512-bit file encryption symmetric keys are then encrypted to each user that we share the file with using the 414 bit ECC Curve41417 public key algorithm in a form of ECC–DH Encryption. Similar, but not exactly compliant to ANSI X9.63, we produce a random key, hash it, and XOR the digest against the file encryption key.

Each user's public key is identified with a keyID derived from a truncated Skein256 hash of that public key.

File Encryption details

Each document in the cloud is stored as a pair of files with the ".rcrd" and ".data" extensions. The record (".rcrd") file is a JSON document that describes, among other things, which users have access to the data part. The record file is not encrypted as a whole, but does contain encrypted components within it.

"rcrd" file JSON tags

Tag	Description
version	packet version
fileID	random uuid string
keys	dictionary of encryption keys
meta	encrypted file meta-data
children	list of children nodes

One of the entries in the record file is the "meta" tag. This typically consists of file metadata items such as filename & mime type. We encrypt the entire value with TwoFish-256 in CBC mode using the same file encryption key as the data itself.

For example the decoded (post-decryption) JSON packet for a 'meta' tag:

```
{
  "type": "file/pdf",
  "filename": "Quarterly Revenue.pdf"
}
```

The data file component is always encrypted as it contains the actual content. We encrypt this using ThreeFish with a 512-bit cryptographically secure pseudo-random generated key.

This is an encrypted file split into 4 sections:

- header
- file metadata
- file preview (small image/thumbnail)
- file data

The structure is designed so a light-weight client (such as an iPhone), can download just the metadata & thumbnail without the downloading the entire file content.

The section of the record file that describes the encryption for each user is a JSON dictionary keyed by userID and contains a set of access permissions as well the public key encoded file encryption key.

```
 keys: {
   "UID:abc123": {
     "perms": “rws",
     "key": <key.abc123>
  },
    “UID:def456": {
      "perms": "rw",
      “key": <key.def456>
  }
}
```

While the server cannot get access to the encryptionKey for the file, it can see who the file is being shared with. This is required, in order for the server to properly send push notifications to the correct users/devices.

Where <key.abc123> is base64 encoded JSON structure that describes the S4 public key wrapped file encryption key (FK). In effect:

$EFK_{abc123} = E(FK, Pub_{abc123})$
$EFK_{def456} = E(FK, Pub_{def456})$

Where the Encryption process (E) using the ECC Curve41417 works as follows:

1. First we generate an ephemeral ECC public key pair ($KS_{priv.pub}$).

2. The Elliptic Curve Cryptography Cofactor Diffie-Hellman (ECC CDH) Primitive is used to compute the 414-bit shared secret (K) as specified in section 5.7.1.2 of NIST SP 800-56A. Such that :

   $K = EC\text{-}DH(KS_{priv}, K_{Pub})$

3. Since the key KS_{priv} is ephemeral to this encryption, and what we really need is pre-image resistance for the shared secret K, it sufficient to do a simple hash expansion with the shared secret such that we have a key that matches our encryption key size, so that:

   $S_{key} = Hash(SHA\text{-}512, K)$

4. Encrypt the Encryption key using that expanded S_{key} such that
   `for(x = 0; x < keylen; x++) EK[x] = sKey[x] ^ FK[x];`

5. Encode the result EK along with the public component of the key we generated in step one KSpub in an ANS.1 format so the other side can also calculate the shared secret and decode it.
   $EFK = (hashOID + KS_{pub} + EK)$

The decrypting side can then take KS_{pub} and using their private key recalculate the shared secret K. Run the same hash process and decode the original FK from the EK.

As example the actual JSON packet for such a key would look like:

```
{
  “version"  : 1,
  “encoding" : "Curve41417",
  “keyID"    : "kMsNAfQEkGs9yiHKQkRIyQ==",
  “keySuite” : "ThreeFish-512",
  “mac"      : "Lv5f6R3fZPY=",
  “encrypted": "MIHEBglghkgBZQMEA...9oSjelMxZV93bAQ=="
}
```

- "version" indicates the S4 packet format version.
- "encoding" describes the kind of public key

- "keyID" is first 128 bits of Skein-256 Hash of Public Key in ANSIx963 format. This is used to uniquely identify each public key.
- "keySuite" specifies what kind of key is being encrypted.
- "mac" is the first 64 bits of a Skein-256 HMAC of the encrypted key. This is used to verify that decryption was correct.
- "encrypted" is the 512 bit file encryption key (64 bytes) encrypted to the public key (199 bytes) in a radix64 ANS.1 encoded format.

Public and Private Keys

During account creation, each Storm4 user creates an ECC Curve41417 public/ private key pair. A copy of the public key is uploaded as a file to the root of that user's AWS bucket and is available for reading by other users to establish sharing of files.

The public version of a users key file would look like:

.pubKey file

```
{
  "version": 1,
  "keySuite": "Curve41417",
  "keyID": "kMsNAfQEkGs9yiHKQkRIyQ==",
  "pubKey": “BB6sJjf57oOKnFu5GxWmR5sBd3…SfKm5UGS3”,
  "userID": "641ihdfw7qf5pj78pfxbunwkkwonu5rg"
}
```

The user’s private key is then encrypted to cryptographically strong random 256-bit key, which we call the “clone key”. We encrypt the private key using the TwoFish-256 algorithm as **the private key never leaves the device unencrypted**.

The *clone key* is very critical to security and is never made available outside the user's control.

In order to properly sync files across each one of the user’s devices, each device needs to have a copy of the private key. Even in the case where the user only

owns one device, recovering the key material and settings is critical in the case where a user loses a device, locks or remotely wipes the device.

To simplify cloning and recovery we upload a copy of the *strongly encrypted* private key to the server, with owner only permissions set. By doing this we only require that the clone key be transferred across the user's devices.

Note that without the 256-bit clone key, the private key is useless to an attacker.

The private version of the users key file

.privKey file


```
{
  "version": 1,
  "encoding": "Twofish-256",
  "keySuite": "Curve41417",
  "mac": "KjPAIcIwYiA=",
  "keyID": "kMsNAfQEkGs9yiHKQkRIyQ==",
  "privKey":
"PSsnd3JzxdJ7EuEo7ljW5aSMlB8q4dQiRM1aTalNuYAEhEtEo2yjh0amSgxAgdnw2kdvBd2CfHe9
TcG+0bC31MMM8ZgWgkM5yOzOjBsN+2NIpoGIMEIslRJTjWaugXlzlHj1KK8fapoY4icL7h2R8WxKm
P1gaz/
QRk0y33DgR0wyjRZDJBDb0L2oyt5DjV+wllbvl0fq0cb6UdFu0moTJVFhrOp8LDT4lz3EX7pwrJQw
6Orw4/XjZ4q5ookZ11W4",
  "userID": "641ihdfw7qf5pj78pfxbunwkkwonu5rg"
}
```


- "version" indicates the S4 packet format version.
- "keySuite" specifies what kind of key is being described"
- "keyID" is first 128 bits of Skein-256 Hash of Public Key in ANSIx963 format. This is used to uniquely identify each public key.
- "pubKey" is a radix64 encoded version of the Public Key in ANSIx963 format
- "privKey" is the ECC Private key encrypted to the 256-bit clone key using the Two-Fish algorithm in CBC mode with enough additional padding to the next 16 byte boundary. The publicly available keyID is used as the initial vector.
- "mac" is the first 64 bits of a Skein-256 HMAC the ECC Private key bytes. This is used to verify the private key has been de
- "userID" links the key to the appropriate Storm4 key owner.

Cloning and device recovery

Since one of Storm4's primary functions is the ability to securely sync encrypted documents to whatever devices a user chooses, we need a way to securely install the user's private key.

Once the user has properly authenticated and attained the needed credentials, they are able to download the ***encrypted*** version of the private key placed on the server on during account creation. But as we mentioned earlier, the private key is still encrypted with a 256-bit clone key.

Thus the cloning process requires us to only transfer the ***clone key*** from one device to another.

Storm4 currently offers two ways to both clone and recover the private key from one device to another. One method involves the creating a QR code that can be read by the device camera. Another method involves splitting the key to a series of words that must be manually entered into a device.

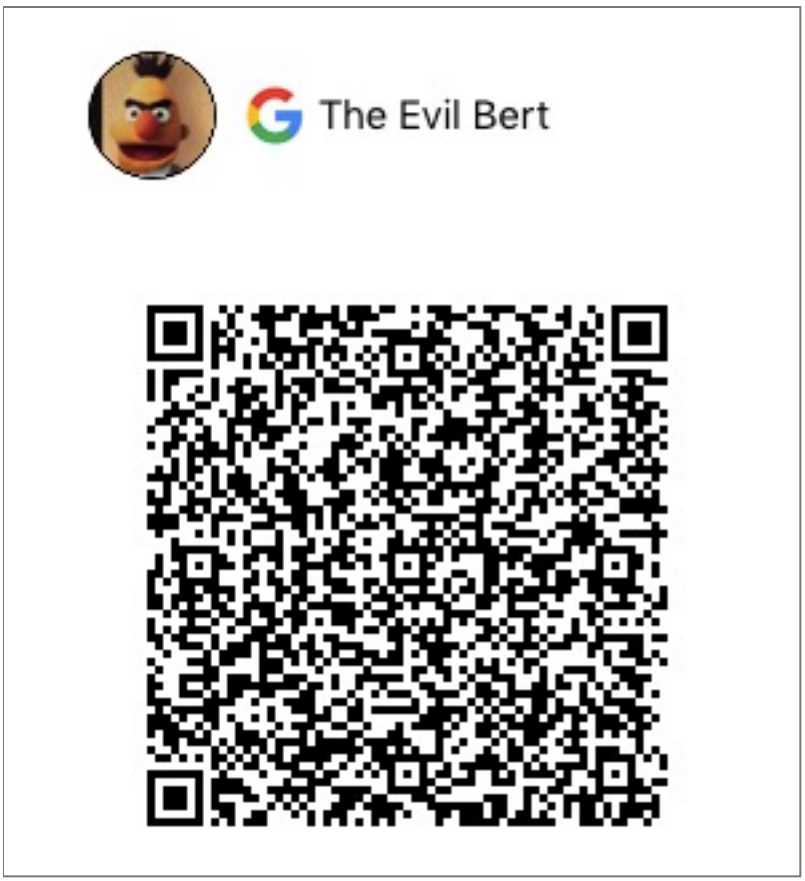
QR Code Cloning

One of the methods that Storm4 uses employs the availability of a device to scan a QR code with its built in camera.

To facilitate this we encrypt the clone key to a random 256 bit key using the TwoFish algorithm. This is called the “session key”. We then further encrypt the session key with a passphrase using the PBKDF-2 key derivation function. The resultant wrapped key as well as the PBKDF-2 salt, rounds and product are then encoded as a QR code image and that is read by the new device's camera acting as a QR code scanner.

$K_{Session}$ = random 256 bit session key
$Q_{r1} = E_{2Fish}(\text{ cloneKey}, K_{Session})$

$K_{PBKDF2} = E_{PBKDF2}(K_{Session}, <passphase>)$
QRcode= storm4://clone2/<aws_id>/< $K_{PBKDF2\ .base64}$>/ < Qr1..base64>



Sample cloning QR code

This QR code can either be ephemeral and immediately scanned in to the cloned device or saved (possibly on paper, in a fireproof safe) for recovery.

In addition to securing the cloning process, the clone code can act as a recovery key in the event that the user loses or damages their devices. We therefore highly recommend that the user backup a copy of the clone code.

Note that **without the clone code**, it is not possible for the user to recover the secure cloud data. We have purposely designed the system so that the provider does not have access to the data.

There are a few other options available in our innovation pipeline to create redundant backup systems for the clone code, but any such option needs to be balanced with security.

BIP 0039 Mnemonic code.

An alternative method of transferring and backing up the clone keys involves a variation of the Bitcoin Improvement Proposal 0039[8] (BIP-0039); “Mnemonic code for generating deterministic keys”. This process has been suggested for backing up Bitcoin wallets. BIP-0039 splits the key into a series of memorable words from a 2048 word dictionary which the user can write down and save as they see fit.

Storm4 uses a variation of this method to split the 256-bit clone key into a series of 24 words that use be entered in the proper order to unlock the private key.

The wordlists are available in a variety of languages and is created in such way that it's enough to type the first four letters to unambiguously identify the word.

1. First we calculate a session key based on a high entropy passphrase only

   K_{salt} = “mnemonic” + <passphase>
   $K_{Session} = E_{PBKDF2}(K_{salt}$, <passphase>)

2. We then encrypt the clone key to the session key using AES-256, thus producing a 256 bit value, known as K1.

   $K_1 = E_{AES\text{-}256}$(cloneKey, $K_{Session}$)

3. We then calculate a cryptographic hash from the original cloneKey and extract the first 8 bytes of the hash and append it to the end of K1, producing a 264 bit key string.

   Checksum = SHA256(<passphase>)
   keyString = K1[0-31] + Checksum[0-8];

4. For each of the bits in the resultant key string we extract 11 bits at a time and convert those 11 bits to an offset into a table of words appropriate for that language. We append the selected the words to produce a mnemonic string of 264/11 = 24 words.

   ```
   FOR ( each of the bits in keyString in 11 bit groups)
   {
       index =  an integer calculated from the  next 11 bits in keyString  ;
       output the word from the dictionary offset by the index above
    }
   ```

This resultant collection of 24 words can then be secured by the user in a variety of ways ranging from writing them down on a piece of paper to embedding them in a stainless steel tile assembly such as the Cryptosteel Wallet[9]

Encryption on Device (macOS/iOS)

While each implementation of the storm4 client takes advantage of appropriate security features of its platform, document protection on the device also follows the same rigorous standards as we do in the cloud. For macOS and iOS we encrypt the data on the device in the following way:

- The files are encrypted with ThreeFish-512 for the most part in the same format as found in the cloud. Each file with its own encryption key.
- File thumbnails are each encrypted with that file's particular encryption key using Twofish-256 in CBC mode.
- The database that manages the files, file encryption keys, meta-data, directory structure and user information is protected using SQLCipher which employs AES-256 in CBC mode.

The 256 bit storage key used to protect the database is randomly generated on application first run. Since regenerating the key would require us to re-encrypt the local database, we never alter the key for the lifetime of the application install.

To ensure security we also never export the storage key from memory unencrypted. Whatever methods used to persist a copy of the storage key must use strong encryption and require that the key be unlocked by the user on application startup.

On macOS and iOS the storage key can be protected by the Apple Keychain Services or encrypted to a user passphrase, or both, depending on the user's needs.

Storage key protected by Keychain Services

Apple Keychain Services provides a number of secure ways to manage keys that take advantage of Apple's built in user authentication. Apple has gone to great lengths to protect any data handed to Keychain Services.

On modern mobile devices such as iPhone and iPad, Keychain Services uses the Secure Enclave[10] coprocessor. The Secure Enclave maintains the integrity of data protection even if the kernel has been compromised. It also is responsible for processing fingerprint data from the Touch ID sensor, determining if there is a match against registered fingerprints.

Storm4 can use Keychain Services to store a decryption key such that it cannot be accessed after a restart until the device has been unlocked once by the user.

On devices that support TouchID, the user may optionally encode request that a decryption key be secured by the Secure Enclave coprocessor such that it can be unlocked by Apple's fingerprint sensing system.

If the user selects to protect the storage key using Keychain Services, Storm4 will generate a random 256 bit protection key (PK) and encrypt it using Twofish-256 rather than hand the Keychain Services the actual storage key. Such that:

ESK = E(SK, PK)
SK = D(ESK,PK)

- "SK" is the 256 bit storage key we use to encrypt our database
- "PK" is the 256 bit protection key secured by Keychain Services.
- "E" and "D" represents the TwoFish-256 encryption and decryption function we use to secure the 256 bit storage key.
- "ESK" is the resultant storage key encrypted to the PK protection secured by Keychain Services.

The protection key (PK) is stored by Keychain Services with the following attributes

Protection key attributes in Keychain Services

Attribute	Value
kSecAttrService	com.4th-a.storm4.keyChainPassphrase com.4th-a.storm4.biometricPassphrase
kSecValueData	<protection key NSData>
kSecClass	kSecClassGenericPassword
kSecAttrSynchronizable	kSecAttrSynchronizableAny
kSecAttrAccessible	kSecAttrAccessibleAfterFirstUnlock

The parameters necessary to decrypt the storage key are then stored in a the ***storm4.pbkdf2*** file in the Application Support folder.

Keychain Services entry in *storm4.pbkdf2* file

```
{
  "version": 1,
  "encoding": "Twofish-256",
  "keySuite": "Twofish-256",
  "mac": "UaSLQr/0w2Y=",
  "encrypted": "qwwAmsnBRdrWCu4+Qlse5u8JNPkUe1p1Vo3BmsmUu9I=",
  "passPhraseSource": "keychain"
}
```

TouchID entry in *storm4.pbkdf2* file

```
{
  "version": 1,
  "encoding": "Twofish-256",
  "keySuite":  "Twofish-256",
  "mac": "4zGOdLpXEHM=",
  "encrypted": "GughbT/ZQ5i8QLHC4ocLJtnB5OogRvac8OMtYIJShnQ=",
  "passPhraseSource": "biometric"
}
```

- "version" indicates the S4 packet format version.
- "encoding" indicates that the key is protected by Twofish-256 encryption.
- "keySuite" specifies what kind of key is being described in our case Twofish-256.
- "mac" is a radix64 representation of first 64 bits of a Skein-256 HMAC of the storage key. This is used to verify that decryption was correct.
- "encrypted" is a radix64 representation of the 256 bit Encrypted storage key (ESK) described above.
- "passPhraseSource" designates source of the protection key (PK) passed to the decryption process.

Storage key encrypted to passphrase

A user may wish to opt-out of the protection provided by Apple Keychain Services and lock the Storm4 application with a user selected passphrase. In this case we use the PBKDF-2 key derivation function to securely encode the 256 bit storage key.

The key derivation is described as follows:

DK = PBKDF2(PRF, Password, salt, rounds, dkLen)
ESK = E(SK, DK)
SK = D(ESK,DK)

- "SK" is the 256 bit storage key we use to encrypt our database
- "PBKDF2" - is the Password-Based Key Derivation Function 2 process
- "PRF" is the pseudorandom function HMAC-256
- "Password" is the password specified by the user.

- "salt" is pseudo-randomly generated 64 bit value intended to prevent attacks involving precomputed hashes for passwords.
- "rounds" is the number of iterations such that the process would take more than .1 seconds per try. This will vary depending on CPU speeds.
- "dkLen" is the desired length of the derived key (in our case 256 bits)
- "DK" is the derived key generated by the PBKDF2 process.
- "E" and "D" represents the TwoFish-256 encryption and decryption function we use to secure the 256 bit storage key.
- "ESK" is the resultant storage key encrypted to the DK derived key.

Using the S4 library a client application can combine the proper passphrase and the PBKDF-2 information to compute the resultant symmetric key. The parameters necessary to decrypt the storage key are then stored in a the ***storm4.pbkdf2*** file in the Application Support folder.

```
{
  "version": 1,
  "encoding": "pbkdf2-Twofish-256",
  "keySuite": "Twofish-256",
  "salt": "Gh1fWwWhcvI=",
  "rounds": 121951,
  "mac": "WfXnBXB5lR4=",
  "encrypted": "ajmVAfdyK1LIy3Ownc+Qt3qXc5JiCM8DNLGe5srKl2s=",
  "passPhraseSource": "keyboard"
}
```

- "version" indicates the S4 packet format version.
- "encoding" indicates that the key is protected by PBKDF-2 process.
- "keySuite" specifies what kind of key is being described in our case Twofish-256.
- "salt" is the radix-64 encoded 64 bit salt value described above.
- "rounds" is the number of rounds to pass into PBKDF-2

- "mac" is an HMAC of the derived key "DK" described above, we use it to verify the validity of the decoded passphrase process.
- "encrypted" is a radix64 representation of the 256 bit Encrypted storage key (ESK) described above.
- "passPhraseSource" designates the passphrase is entered by the user from a keyboard.

Storage Recovery Key

Storm4 on macOS has an ***experimental*** option to protect the storage key in the event that user forgets the passphrase. **This option might not make it into the final version**.

It is possible to generate a QR-code based recovery key sheet that can be used to unlock the application. In this case a random 256 bit protection key is created, then base64 encoded and passed to the PBKDF2 process similar to the passphrase protection scheme.

In this case the the ***storm4.pbkdf2*** file would have the following entry:

```
{
  "version": 1,
  "encoding": "pbkdf2-Twofish-256",
  "keySuite": "Twofish-256",
  "salt": "iy8sMYJcsnw=",
  "rounds": 112359,
  "mac": "4zGOdLpXEHM=",
  "encrypted": "GughbT/ZQ5i8QLHC4ocLJtnB5OogRvac8OMtYIJShnQ=",
  "passPhraseSource": "recovery"
}
```

Possible issues and mitigation of threats

Public Key verification

The authenticity of a peer's public key is critical to the security of sharing a document with them.

One of the possible vulnerabilities the have plagued most public key system designs is the possibility of man-in-the-middle attack (MITM). This is where the attacker secretly relays and possibly alters the communication between two parties who believe they are directly communicating with each other. In the case of Bob sharing a file to Alice, an attacker would intercept and modify the communication between Bob and the server and present Bob a counterfeit public key for Alice. Bob would then encrypt a file to the counterfeit public key before sharing it.

And although attacker would have to overcome a number of safeguards, the least of which is to attack the TLS connections and certificates between the client app and the server (in this case Amazon Web Services), the attack is still possible.

Traditionally the two ways systems have addressed this attack, is to either sign the recipient's public key through some trusted certificate chain, or to require the user to verify the hash of the public key fingerprint or hash.

Sadly, experience developing and deploying a number of large scale cryptographic communication systems has shown us that users rarely, if ever, take the time to manually verify such things, and often find them to be an encumbrance. Yet, like seat-belts, driving without them decreases ones survivability.

As an alternative, one technology that shows promise is the blockchain. A blockchain is a distributed immutable database where each entry or block has a timestamp and a cryptographic digest link to a previous block such that it is very difficult for the data in a block to be altered retroactively. In effect, we have a tamper resistant distributed immutable ledger.

We plan to use the blockchain technology to announce the association of a user and their public key in such a way that a man-in-the-middle attack becomes infeasible. Currently we have a preliminary version utilizing the Ethereum blockchain platform. The intent is that our client code (and optionally the user through manual means) can verify the validity of the public key with such a high degree of authenticity that tampering is unlikely.

Clone code due diligence

The clone code is critical to the security of the user's Storm4 repository. If a user loses a device it can be used to restore access. Conversely without the clone code it is not possible to recover the private key. Some basic guidelines that a user should consider include:

- Don't lose it
- Don't let others photograph it
- Use a good passphrase and don't forget it
- If storing a physical backup of the key, consider the effects of a fire or flood. An alternative to an (expensive) fireproof safe would be something like the Cryptosteel wallet (and then storing the BIP-0039 mnemonic code).

Appendix A: Document History

Date	Rev	Author	Change
4/14/17	0.1	vinnie	First complete draft.
6/9/17	0.2	vinnie	Added discussion of clone codes.

References

[1] Daniel J. Bernstein, Tanja Lange , SafeCurves: choosing safe curves for elliptic-curve cryptography.
https://safecurves.cr.yp.to

[2] Bruce Schneier, Twofish block cipher
https://www.schneier.com/academic/twofish

[3] Bruce Schneier, Threefish
https://www.schneier.com/academic/skein/threefish.html

[4] Niels Ferguson ,Stefan Lucks , Bruce Schneier, Doug Whiting, Mihir Bellare, Tadayoshi Kohno, Jon Callas, Jesse Walker - The Skein Hash Function Family
https://www.schneier.com/academic/skein/

[5] Matthew Green, A riddle wrapped in a curve.
https://blog.cryptographyengineering.com/2015/10/22/a-riddle-wrapped-in-curve/

[6] Daniel J. Bernstein, Tanja Lange. "Security dangers of the NIST curves." September 2013.
https://cr.yp.to/talks/2013.09.16/slides-djb-20130916-a4.pdf

[7] 4th A Technologies, LLC. S4 - Security Library 4
https://github.com/4th-ATechnologies/S4

[8] BIP 0039 Mnemonic code for generating deterministic keys
https://github.com/bitcoin/bips/blob/master/bip-0039.mediawiki

[9] Cryptosteel Cold Storage Wallet
http://cryptosteel.com/product/cryptosteel-mnemonic/

[10] Apple Inc. iOS Security—White Paper | March 2017
https://www.apple.com/business/docs/iOS_Security_Guide.pdf