Bitcoinj Encrypted Private Key Wallets (version 3)

Introduction

This document discusses the format for storing and using encrypted private keys in Bitcoinj.

The overall goal is for bitcoin private keys to be stored in a strongly encrypted form and that:

- Any software based on bitcoinj will be able to read and write the encrypted private keys using the same passphrase.
- The encryption format makes it possible for non-Java developers to read and write the encrypted private keys relatively easily.

Protobuf message changes

The current protobuf message definition for bitcoinj is stored here:
http://code.google.com/p/bitcoinj/source/browse/core/src/bitcoin.proto?name=release-0.5

The changes to support encrypted private keys are as follows:

The Wallet message is changed to:

```
/** A bitcoin wallet */
message Wallet {
    required string network_identifier = 1; // the network used by this wallet
    // org.bitcoin.production = production network (Satoshi genesis block)
    // org.bitcoin.test = test network (Andresen genesis block)

    // the Sha256 hash of the head of the best chain seen by this wallet.
    optional bytes last_seen_block_hash = 2;
```

```
    repeated Key key = 3;
    repeated Transaction transaction = 4;
    optional EncryptionType wallet_type = 5;                        // New
    optional ScryptParameters scrypt_parameters = 6;                // New
    optional int32 major_version = 7;                               // New
    optional int32 minor_version = 8;                               // New

    repeated Extension extension = 10;
} // end of Wallet
```

EncryptionType is optional so that it does not break existing Wallets, which are grandfathered in as EncryptionType = UNENCRYPTED.
The EncryptionType message is:

```
enum EncryptionType {
    UNENCRYPTED = 1;           // All keys in the wallet are unencrypted
    ENCRYPTED_SCRYPT_AES = 2;  // All keys are encrypted with a KDF of scrypt and AES encryption
}
```

EncryptionType is an enum rather than a boolean so that we can add different KDF + encryption algorithm combinations in the future.

There is a new message ScryptParameters that stores the salt for the Wallet's password generation and is also used to store non-default values for the Scrypt KDF:

```
message ScryptParameters {
  required bytes salt = 1;                      // Salt to use in generation of the wallet password (8 bytes)
  optional int64 n = 2 [default = 16384];       // CPU/memory cost parameter
  optional int32 r = 3 [default = 8];           // Block size parameter
  optional int32 p = 4 [default = 1];           // Parallelisation parameter
}
```

The Key type has an EncryptedPrivateKey field added:

```
message Key {
   enum Type
      ORIGINAL = 1; // Original bitcoin secp256k1 curve
   }
   required Type type = 1;
   // The private EC key bytes without any ASN.1 wrapping, unencrypted.
   optional bytes private_key = 2;
   // The private EC bytes, encrypted with scrypt and AES
   optional EncryptedPrivateKey encrypted_private_key = 6;                    // New
   // The public EC key derived from the private key. We allow both to be stored to avoid mobile clients having to do lots of slow
         EC math at startup.
   optional bytes public_key = 3;

   // User provided label associated with the key.
   optional string label = 4;

   // Timestamp stored as millis since epoch. Useful for skipping block bodies before this point.
   optional int64 creation_timestamp = 5;
}
```

The new EncryptedPrivateKey message contains the data required for an encrypted private key as follows:

```
message EncryptedPrivateKey {
  required bytes initialisation_vector = 1;       // The initialisation vector for the AES encryption (16 bytes)
  required bytes encrypted_private_key = 2;       // The encrypted private key
}
```

Key Derivation Function (KDF)

The key derivation function converts a passphrase entered by the user to an encryption key that is used to encrypt and decrypt the plaintext private key. Scrypt is used as it is designed to be resistant to GPU based brute forcing as it is memory bound.

There is a presentation describing Scrypt here:
http://www.tarsnap.com/scrypt/scrypt-slides.pdf

There is a Java implementation here, with both pure java + JNI versions. An ARM binary for Android is included.
https://github.com/wg/scrypt

The parameters to pass to scrypt are:

- passphrase - Supplied by user
- salt - Random bytes. Stored in the ScryptParameters.salt parameter. Specified once per wallet.
- N - CPU/ memory cost parameter. Stored in the ScryptParameters.n parameter. Specified once per per wallet.
- r - Block size parameter. Stored in the ScryptParameters.r parameter. Specified once per wallet.
- p - Parallelisation parameter. Stored in the ScryptParameters.p parameter. Specified once per wallet.

The default values for N, r, p are taken from Colin Percival's original scrypt paper (http://www.tarsnap.com/scrypt/scrypt.pdf, page13, para 1 and page 12, para 1)

If non-default N, r, p parameters are used in the KDF for a Wallet they must be stored in the ScryptParameters message.

There is some example code implementing the Scrypt KDF combined with AES encryption here:
https://github.com/jim618/multibit/blob/v0.5/src/main/java/org/multibit/crypto/EncrypterDecrypterScrypt.java
https://github.com/jim618/multibit/blob/v0.5/src/test/java/org/multibit/crypto/EncrypterDecrypterScryptTest.java

The purpose of the salt in the KDF (stored one-per-wallet in the ScryptParameters) is to prevent an attacker doing a dictionary attack as follows:

1. Use a precompiled dictionary and create the corresponding derived AES keys using Scrypt. Do this once for all attacks (slow).
2. Using the derived AES keys and the stored IV, try decrypting the encrypted bytes (quick).

3. See if the unencrypted bytes are a Bitcoin key. If so, attack is successful.

Encryption algorithm

A 256 bit AES chain block cipher is used to encrypt the bitcoin private keys. The initialisation vector to use is given in the EncryptedPrivateKey.initialisation_vector field and is specified per key.

Operations that are the same on encrypted and unencrypted wallets

Several operations on an encrypted wallet do not require the encryption passphrase and hence are identical for Wallets of type UNENCRYPTED and ENCRYPTED_SCRYPT_AES. These include:

- Getting the Wallet balance.
- Receiving new blocks.
- Manipulating the transactions (for instance in a reorg).
- Creating an empty Wallet (one with no private keys).
- Deleting a Wallet.
- Storing and loading of the Wallet.
- Deleting a private key.

Descriptions of Java encryption and decryption methods available

This section describes the Java methods available to work with the encrypted wallets.

There is a method on a helper object (which implements the EncrypterDecrypter interface) to convert a passphrase to an AES key (typed as a KeyParameter):

```
public KeyParameter deriveKey(char[] passphrase);
```

The encryption and decryption related methods on Wallet are:

1. public synchronized void encrypt(KeyParameter aesKey);
   Encrypt the wallet instance with the AES key supplied.
2. public synchronized boolean decrypt(KeyParameter aesKey)
   Decrypt the wallet instance with the AES key supplied. The returned boolean is true if the keys all decrypted to valid Bitcoin private keys, false otherwise. If the wallet did not decrypt (i.e. returned boolean = false) the wallet remains in its original, encrypted state - this is most likely caused by an incorrect passphrase.
3. public boolean isCurrentlyEncrypted()
   Returns true if the wallet instance is currently encrypted, false otherwise

Notes:

1. (Wallet is of type ENCRYPTED_SCRYPT_AES). If you call decrypt on a Wallet where isCurrentlyEncrypted() = false a WalletIsAlreadyDecryptedException is thrown and the Wallet remains unchanged.
2. (Wallet is of type ENCRYPTED_SCRYPT_AES). If you call encrypt on a Wallet where isCurrentlyEncrypted() = true a WalletIsAlreadyEncryptedException is thrown and the Wallet remains unchanged.
3. If you call encrypt or decrypt on a Wallet of type UNENCRYPTED, nothing happens.
4. After a password is used to encrypt and/or decrypt it should be overwritten in memory (as best as possible using Java) as soon as possible. This is to avoid attacks looking directly at the machine's memory. Note that this is imperfect in Java as there still may be copies of the password object in the garbage collection heaps.
5. When a Wallet of type ENCRYPTED_SCRYPT_AES that has isCurrentlyEncrypted() = true is stored to disk it should always be written with its private keys encrypted. The Key.private_key field (containing the unencrypted private key) should always be blank. This includes error paths.

Wallet Versions and Mandatory Extensions

The intended use of the major and minor versions is so that applications can understand which wallets it can deal with and which wallets are 'from the future' and should not be loaded.

As of 18th Aug 2012 the existing wallet versions used (for MultiBit) are:

- serialised wallets.
    - major version = 1, minor version not used (grandfather in as 0)
- protobuf according to the first version of bitcoin.proto (unencrypted wallets).
    - major version = 2, minor version not used (grandfather in as 0)
- protobuf according to the bitcoin.proto definition in this document
    - major version = 3, minor version = 0

It is suggested that any application using the Wallet major and minor versions uses these version identifiers when they write their wallets to avoid interoperation confusion.

The major version increments when a breaking change is made to the wallet format. This signifies that extant wallet readers should not try to parse the stored wallet because of risk of data loss or misinterpretation. The minor version increments when a non-breaking change is made and can be used by the parsing code as needed.

Note that you can also add an extension with mandatory = true if you want to force older applications to abort loading a Wallet.