Understanding the Mobile Bitcoin Wallet

Ziqi Liu

December 8, 2015

1 ABSTRACT

This project aims at inspecting the detailed implementation of the mobile version of the Bitcoin wallet. Generally, I intend to answer the question: "What should a developer knows, beyond the basic Bitcoin mechanism, to implement a user-friendly Bitcoin wallet?" To answer the question, I choose an open-source mobile Bitcoin wallet application, BreadWallet (iOS version)[1] to analyze the mobile Bitcoin wallet implementation in code-detail level.

The project focus two aspects - secure key management and convenient transaction operations. The project provides some common practices to solve these two problems of the Bitcoin wallet implementation. In addition, the project discussed briefly about whether the wallet security would be compromised in some scenarios using current solutions.

2 Motivation

The developers need to know more beyond the basic Bitcoin mechanism and components to develop a good Bitcoin wallet. This project focuses on this "extra" information. It delivers how the mobile Bitcoin wallet is implemented by solving the big problem both in design level and in code-level. It would help future Bitcoin wallet developers, especially who working on mobile platforms, gather some general information about common practice. It would be a useful guideline for developers to look at before they start to design and implement.

3 Background

There are some guidelines, like Bitcoin Developer Guide[2] and Wallet Implementation Guide[3], for developers who intend to build a Bitcoin wallet. Unfortunately, none of them can focus on detailed discussion or practical implementation, especially for mobile platforms .Bitcoin Developer Guide is the best one among them. It generally explains most of the techinical components of the Bitcoin, and it's a good theoratical start for developers to know more about Bitcoin.

Other than the guidlines, most of the technical specification covered in the project has the detailed description in BIP on Github. [5][7]

4 Analysis

For users, a mobile Bitcoin wallet app needs to be easy to use but safe under any circumstances. A Bitcoin wallet app should perserve the Bitcoin's decentralized principle and user's anonymity, and keep user's sensitive information as secret with enough security level. Our analysis focuses on safety of key management and transaction protocols in Bitcoin wallet.

I will present what is the common practice in Bitcoin wallet implementation to solve these two problems.

4.1 Key Management Security

The key management problem exists in every Bitcoin wallet. The private key, which grants user's access to the Bitcoin he/she received, is the most essential information for users in the Bitcoin world. Also, as the users needs more than one the public keys, some times for every new transaction, to preserve privacy, the public key also needs to be safely stored in the wallet. Thus, these data needs to be kept safe within the wallet. Furthermore, the keys needs to be backed up safely, as the users ,may want to restore the wallet under some circumstances.

The BreadWallet somehow achieves these goals. It takes advantage of the *iOS KeyChain* and *Hierarchical Deterministic Wallet* to solve the problems. Thus, the app allows users to restore the wallet when the mobile phone got lost and kept the key information securely when using the app or the phone got stolen or controlled by some malware.

4.1.1 Hierarchical Deterministic Wallet

BreadWallet uses a design called *Hierarchical Deterministic Wallet (HD wallet)* , it was fully described in BIP0032(Bitcoin Improvement Project).By using this design, a wallet with its information can be used among different client devices, and it stores all the key pairs by deriving a tree of keypairs from a single seed.

In BreadWallet, the app takes the seed from local storage(which we will discussin section 4.1.2), and computes the private key using seed in the memory. After signing the transaction, the computed private key is released from memory using the macro @ autorelease.

```
- (BOOL)signTransaction:(BRTransaction *)transaction withPrompt:(NSString *)
    authprompt{
  ...
    @autoreleasepool { // @autoreleasepool ensures sensitive data will be
        dealocated immediately
        NSMutableArray *privkeys = [NSMutableArray array];
        NSData *seed = self.seed(authprompt, (amount > 0) ? amount : 0);
        if (!seed) return YES; // user canceled authentication
        [privkeys addObjectsFromArray:[self.sequence privateKeys:
            externalIndexes.array internal:NO fromSeed:seed]];
        [privkeys addObjectsFromArray:[self.sequence privateKeys:
            internalIndexes.array internal:YES fromSeed:seed]];

        return [transaction signWithPrivateKeys:privkeys];
    }
}
```

A non-deterministic wallet requires frequently back-up of keys which users generate for every transactions. Using a single seed, which can be used to derive numbers of keys, hierarchical deterministic Wallet avoids such frequent backups. Also, using a single seed, as long as the seed is kept somewhere, the wallet can be shared among different clients by taking this seed to retrieve all the keys.

HD wallet is deterministic, as the same seed always derives the same keys in the sub-tree. The keys we can derive from seeds including the Bitcoin addresses, public keys and private keys of the user. In practice, as BreadWallet does, the seed is a random 128 bit value presented to the user as a 12 word mnemonic using common English words.Most Bitcoin wallets prompt users to keep a hard copy of the words for safety or restoration use. The full description of how to generate the mnemonic words from the sequence is presented in BIP0039.

This design is hierarchical, as it allows several levels of trees of the key derivation. In the tree structure from seed to keys, each node contains the information to the next level.(see figure 4.1)The root seed is input a hash function and the resulting is used to create a master private key and a master chain code. The master private key then generates a corresponding

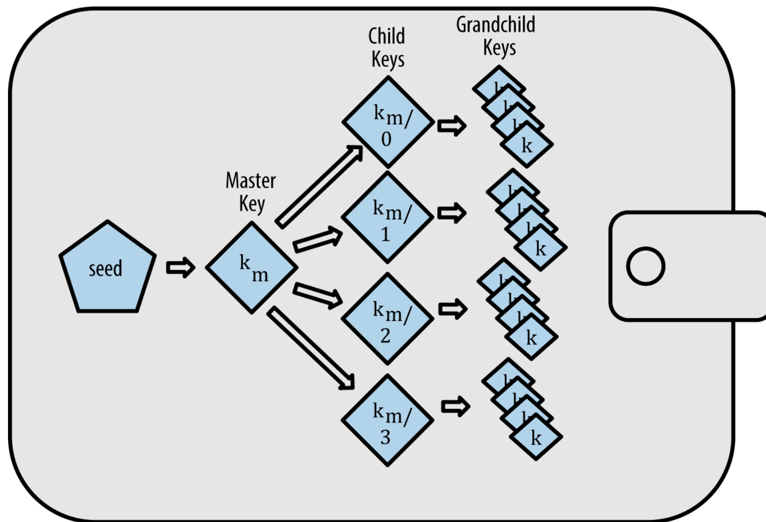

Figure 4.1: Key derivation from a root seed

master public key, using Bitcoin cryptography function. The chain code is used to introduce entropy that creates child keys from parent keys, which will be mentioned later. [4]

The function Child Key Derivation $CKD((k_{par}, c_{par}), i) \rightarrow (k_i, c_i)$computes a child extended private key from the parent extended private key[5].It's important to know that the CKD function is a one way function. It means that there's no way from child key to parent key, or from one branch to another branch. Thus, knowing a master public key, you have no idea of what the master private key or the child extended private key is. This property ensures the security of the key.

Here is a code snippet in BreadWallet, where the app takes the seed and an external/internal option to derive private keys using CKD functions.

```
- (NSArray *)privateKeys:(NSArray *)n internal:(BOOL)internal fromSeed:(NSData
    *)seed
{
    if (! seed || ! n) return nil;
    if (n.count == 0) return @[];

    NSMutableArray *a = [NSMutableArray arrayWithCapacity:n.count];
    UInt512 I;
```

```
    HMAC(&I, SHA512, 64, BIP32_SEED_KEY, strlen(BIP32_SEED_KEY), seed.bytes,
        seed.length);

    UInt256 secret = *(UInt256 *)&I, chain = *(UInt256 *)&I.u8[sizeof(UInt256)
        ];
    uint8_t version = BITCOIN_PRIVKEY;

    CKDpriv(&secret, &chain, 0 | BIP32_HARD); // account 0H
    CKDpriv(&secret, &chain, internal ? 1 : 0); // internal or external chain

    for (NSNumber *i in n) {
        NSMutableData *privKey = [NSMutableData secureDataWithCapacity:34];
        UInt256 s = secret, c = chain;

        CKDpriv(&s, &c, i.unsignedIntValue); // nth key in chain

        [privKey appendBytes:&version length:1];
        [privKey appendBytes:&s length:sizeof(s)];
        [privKey appendBytes:"\x01" length:1]; // specifies compressed pubkey
            format
        [a addObject:[NSString base58checkWithData:privKey]];
    }

    return a;
}
```

4.1.2 Security on Local Seed

In the previous section, we dicussed how the HD wallet uses seed to avoid frequent backups of keys and storage of multiple keys, including private keys. It transfers the problem of key storage to the safety of local seed storage. In practice, the user keeps a hard copy of the 12 English words representations of seed. So when he/she needs to restore the wallet under some circumstances, he/she just needs to enter the words and retrieve the original wallets with all information of transactions and keys. Thus, the seed needs to be stored in the mobile phone under circumstance. Otherwise, the user needs to enter the twelve words every time it make a transaction. As an example, BreadWallet (iOS) places the seed in the Keychain,which

is built right in to iOS. According to Apple Programming Guide,Keychains are secure storage containers, which means that when the keychain is locked, no one can access its protected contents[6]

Under normal circumstance, where the user uses the wallet to do transactions and have securely keep the devices away from threats, the app works perfectly. When using the private keys to signs, or accessing public keys for past transaction details, it retrieves the seed from Keychain and calculate the keys in memory with releasing the memory immediately. In the following paragraphs, we will examine how the HD wallet design along with the seed storage in Keychain against deliberate attempts to compromise security.

Suppose the mobile devices is stolen, or lost, and someone else has physically access to the mobile device, with a safe iOS system, the key is safe. It's because the seed is stored in Keychain, and the Keychain can only be accessed when the devices in unlocked by the PIN. Unless someone breaks the PIN or fingerprint,he/she has no access to the seed in Keychain. Even if the one has ability to dump the memory and get the Keychain, the data is still locked/encrypted by the combination of user's PIN and userID. Correctly obtain this combination with brute force is computational hard to success. As long as the seed is safe, the user's keys, along with Bitcoins, are safe. Thus, we can see, the key management is safe even with malicious physically access of the devices.

Suppose the other applications on the same devices is malicious ,and intends to get the keys by dumping the memory or get access to the Keychain. The Apple's sandboxing keeps the secret away from these threat. According to Apple's Programming Guide, in iOS an application can access only its own keychain items.[6] So other apps, malicious or not, has no access to the keychain seed. Using App Sandbox , if malicious code gains control of a properly sandboxed app, it is left with access to only the files and resources in the appś sandbox. Thus, if a malware intends to get access of the private key when it is calculate before it is released from memory,

during that short period of time, it can only obtain the sandboxed memory data.

However, when restoring the wallet, the secret is not completely safe against threats from some malware. Suppose there exists a screen-capture or tapping -capture malware running successfully in the background when the user enters his/her twelve-word seed, the seed is captured by the malware. Thus, using the same CKD, the malware can gets every keys of the wallet.

Another thing needed to be noticed is that, most of these defence is rely on the mobile operating system design. Suppose the operating system exists some leaks, i.e. a vulnerability in keychain service, then the security of the wallet is compromised. This also means that the wallet is not safe under jailbreaking, which disables most of the security features of the system. In addition, recalling the recent XcodeGhost malware, which compromised the security at complier level, the wallet is not safe against these kind of threats.

4.2 Making the Transaction

The most importance function of Bitcoin is to make transactions. In most basic mode, to successfully make a transaction, the wallet needs to takes the input and output, recipients' addresses from users, signs and broadcast the transaction to the Bitcoin network. In addition, the wallet needs to be able to verify whether the transaction is propagated in the network successfully to confirm a transaction as a recipient.

In our example BreadWallet, in addition to the most basic mechanism, it uses a Payment Protocol BIP0070 to ensure a better user experiences and higher security level in making the transaction.It adds some features for the transaction too. For verification of an incoming transaction, it chooses Simplified Payment Verfication model to verify whether the payment is propagated and obtained certain number of confirmations.It helps the mobile app to save memory and energy, and gain a result with some level of certainty in a short time.

4.2.1 Payment Protocol

The payment protocol is a specific process to pay using bitcoin. As its description in BIP0070, it is " a protocol for communication between a merchant and their customer, enabling both a better customer experience and better security against man-in-the-middle attacks on the payment process." [7]

The protocol helps them exchange the payment infomation in a secured way. The communication is similar to the idea of handshaking protocol, and it can be illustrated in figure 4.2 . All the commication happens on the HTTP paralleling with the Bitcion transaction on the Bitcoin network. The protocol includes PaymentRequest, Payment, and PaymentACK.

The PaymentRequest information is from merchant to customer, which includes the address of the merchants. Using wallet, the users can avoid typing the address as the wallet reads the files and extract the informations automatically. The Payment message is from customer to merchant to send the transaction number which he/she just made based on request information.It also includes the refunds information, which enables user to get refunds to another address automatically if the trade fails. The PaymentACK is a confirmation message from merchants to customer. Below is the code snippet in BreadWallet, where it receives and handle an incoming file (message) for transaction use.

```
- (void)handleFile:(NSData *)file
{
    BRPaymentProtocolRequest *request = [BRPaymentProtocolRequest
        requestWithData:file];

    if (request) {
        [self confirmProtocolRequest:request];
        return;
    }

    BRPaymentProtocolPayment *payment = [BRPaymentProtocolPayment
        paymentWithData:file];

    if (payment.transactions.count > 0) {
```

```
        for (BRTransaction *tx in payment.transactions) {
            [(id)self.parentViewController.parentViewController
                startActivityWithTimeout:30];

            [[BRPeerManager sharedInstance] publishTransaction:tx completion:^(
                NSError *error) {
                 [(id)self.parentViewController.parentViewController
                     stopActivityWithSuccess:(! error)];

                 if (error) {
                     ...
                 }

                 [self.view addSubview:[[[BRBubbleView
                  viewWithText:(payment.memo.length > 0 ? payment.memo :
                      NSLocalizedString(@"received", nil))
                  center:CGPointMake(self.view.bounds.size.width/2, self.view.
                      bounds.size.height/2)] popIn]
                  popOutAfterDelay:(payment.memo.length > 0 ? 3.0 : 2.0)]];
            }];
        }

        return;
    }

    BRPaymentProtocolACK *ack = [BRPaymentProtocolACK ackWithData:file];

    if (ack) {
        if (ack.memo.length > 0) {
    ...
        }

        return;
    }
...
}
```

In addition, The protocol increases the security level of transaction. The native exchanging of receiving address is not secure against man-in-the-middle (MITM) attacks. The attackers who controls the communication channel can modified the receiving addresses for both parties. Using the Payment Protocol, which is can be encrypted using public key infrastructure

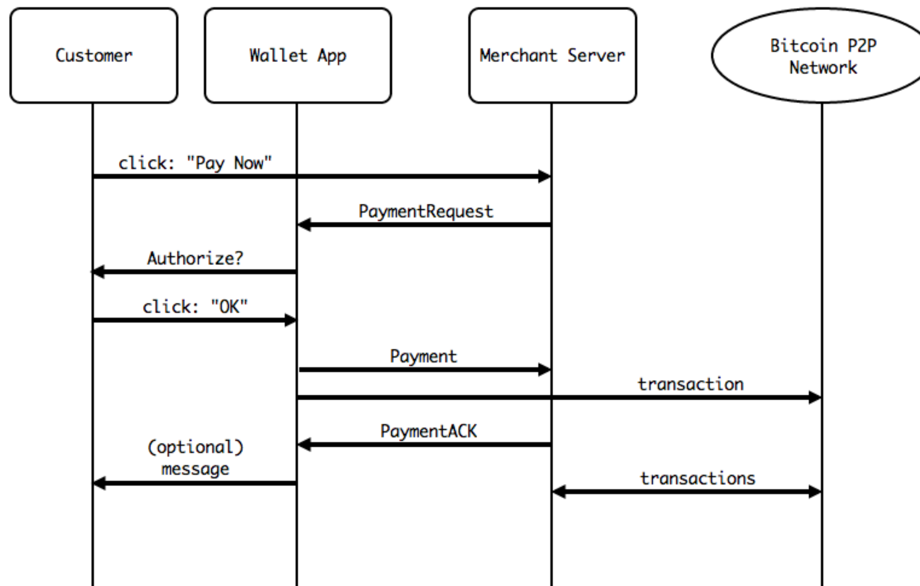

Figure 4.2: Payment Protocol Messages Communication Sequence

(PKI) to sign each message with certificates. Thus, the PaymentRequest cannot be modified as the receiver will verify the message using the sender's public key. In the protocol, it defines a field of PaymentRequest as pki_type, where 0 means no PKI encryption. In BreadWallet, the default of pki_type is 2, but it also takes other options.(shown in code snippet below)

```
typedef enum : NSUInteger {
    request_version = 1,
    request_pki_type = 2,
    request_details = 4,
    request_signature = 5
} request_key;
```

5 Other Interesting Topics

There exist a number of interesting aspects of Bitcoin wallet implementation. Due to the limited of time, I cannot work on them deeply. They are:

1. Simplefied Payment Verification and Bloom Filter

2. Authenticated Bitcoin API calls: BitAuth
3. Merkle tree in wallet

References

[1] Aaron Voisine, *breadwallet - bitcoin wallet http://breadwallet.com*, https://github.com/voisine/breadwallet

[2] Bitcoin Project, *Bitcoin Developer Guide*, https://bitcoin.org/en/developer-guide

[3] CoinSpark, *Wallet Implementation Guide*, http://coinspark.org/developers/wallet-implementation-guide/, 2015

[4] Andreas M. Antonopoulos, *Mastering Bitcoin*, http://chimera.labs.oreilly.com/books/1234000001802/ch04.

[5] Pieter Wuille, *Hierarchical Deterministic Wallets*, https://github.com/bitcoin/bips/blob/master/bip-0032.mediawiki, 2012-02-11

[6] Apple Inc., *Keychain Services Programming Guide*, https://developer.apple.com/library/mac/documentatic 2014-02-11

[7] Gavin Andresen, Mike Hearn, *Payment Protocol*, https://github.com/bitcoin/bips/blob/master/bip-0070.mediawiki, 2013-07-29