

# Mycelium Wallet Backup

Creation Date: January 30, 2023  
Made With: Mycelium Wallet 3.16.0.15  
Backup Format: Mycelium Backup 1.1  
Active Records: 1  
Archived Records: 0  
Total Keys: 1  
Total Addresses: 1

This document contains an encrypted backup of your Mycelium Wallet.

The backup contains sensitive data consisting of one or more private keys and an optional master seed. For your protection this data is encrypted with a 15 character random password.

The password was shown on display while creating the backup, and is different for every backup. It is not possible to restore the backup and access your bitcoins without the password.

Write the 15-character password and the checksum character from the display here:

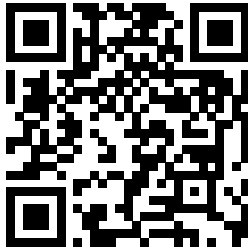
     

Alternatively you can write it down elsewhere. Keep it safe!

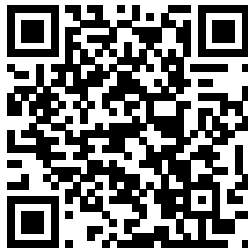
To import a key or a master seed in the Mycelium wallet you need to scan the corresponding QR code and enter the encryption password.

Note that the embedded PDF viewer in Windows 8 cannot display the QR codes properly.

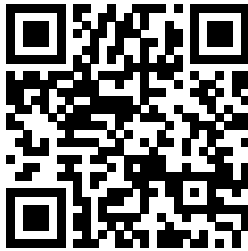
Bitcoin Addresses



1Ba8Fh72zSrgBMj81  
UDCKUGz17HipEC1xM

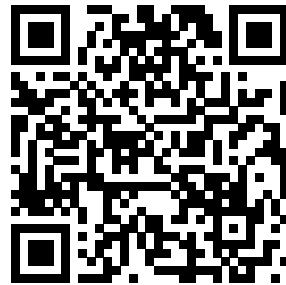


bclqw06s5y2ayuz2k6uxh  
40y64xfyv8r8u882cnxgq



34sLZsubrt8SB9JAT  
pkpXu9MSAfAAxMidb

Encrypted Private Key



xEncEXICqz2G4K5wFxm5u7VTMx7Wp5I  
jAqDyq1j0znAR8l4L7cptfJWuvjPX2A

The Mycelium Bitcoin Wallet performs the steps described below when decrypting and verifying an encrypted private key. The description is quite technical and allows a developer to create software that allows you to decrypt your private keys. This allows you to access your funds if the Mycelium software is no longer available. If you wish to read or review the implementation used by the Mycelium Bitcoin Wallet you can find it here:  
<https://github.com/mycelium-com/wallet-android/blob/master/bitlib/src/main/java/com/mrd/bitlib/crypto/MrdExport.java>

## Parsing The QR Code

1. Scan the QR code to get a Base64 encoded string.
2. Decode the Base64 encoded string to get exactly 46 bytes. The Base64 variant used is designed for URLs as specified in RFC 4648 section 5.
3. The first 3 bytes are the the magic cookie 0xC4 0x49 0xDC: decoded[0...2]
4. The next 3 bytes are the header bytes: H = decoded[3...5]
5. The next 4 bytes is the random salt: SALT = decoded[6...9]
6. The next 32 bytes are the encrypted private key: E = decoded[10...41]
7. The next 4 bytes are the checksum: C = decoded[42...45]

## Decoding the 3 Header Bytes

Regard the header as an array of 24 bits and decode the following values:

```
version    = XXXX???? ???????? ???????? must be 1
network    = ???X??? ???????? ???????? 0 = prodnet, 1 = testnet
content    = ?????XX? ???????? ???????? 000 = private key with uncompressed public key
                                         001 = private key with compressed public key
                                         010 = 128 bit master seed
                                         011 = 192 bit master seed
                                         100 = 256 bit master seed

HN         = ???????? XXXXX??? ???????? 0 <= HN <= 31
Hr         = ???????? ?????XX? X? ?????? 1 <= Hr <= 31
Hp         = ???????? ???????? ?XXXXX? 1 <= Hp <= 31
reserved   = ???????? ???????? ????????X: must be zero
```

## AES Key Derivation

1. Make the user enter a 15-character password using characters A-Z, all in upper case. Convert the characters to 15 bytes using normal ASCII conversion. An implementations may use additional checksum characters for password integrity. They are not part of the AES key derivation.
2. Run script using parameters  $N = 2^{HN}$ ,  $r = Hr$ ,  $p = Hp$  on the password bytes and SALT, to derive 32 bytes.
3. The 32 output bytes are used as the 256-bit AES key used for decryption.

## Decrypting the Content Data

The decryption function is 256-bit AES in CBC mode.

1. Generate the AES initialization vector (IV) by doing a single round of SHA256 on the concatenation of SALT and C, and use the first 16 bytes of the output.
  2. Split E into two 16-byte blocks E1 and E2
  3. Do an AES block decryption of E1 into P1 using the derived AES key
  4. X-or the initialization vector onto P1:  $P1 = P1 \text{ xor } IV$
  5. Do an AES block decryption of E2 into P2 using the derived AES key
  6. X-or E1 onto P2:  $P2 = P2 \text{ xor } E1$
  7. The 32 byte plaintext data is the concatenation of P1 and P2:  $P = P1 || P2$
- If content is 000 or 001 the 32 bytes are a private key  
If content is 010 the first 16 bytes are a master seed  
If content is 011 the first 24 bytes are a master seed  
If content is 100 the 32 bytes are a master seed

## Verifying the Checksum

1. Convert the generated bitcoin address to an array of ASCII bytes
  2. Do a single SHA256 operation on the array of bytes
  3. The checksum is the first 4 bytes of the output
  4. Verify that the calculated checksum equals C.
- If a wrong password was entered the checksums will not match.

## Cryptographic Functions Used

AES - <http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf>  
SHA-256 - <http://csrc.nist.gov/publications/fips/fips180-4/fips-180-4.pdf>  
script - <http://www.tarsnap.com/script/script.pdf>