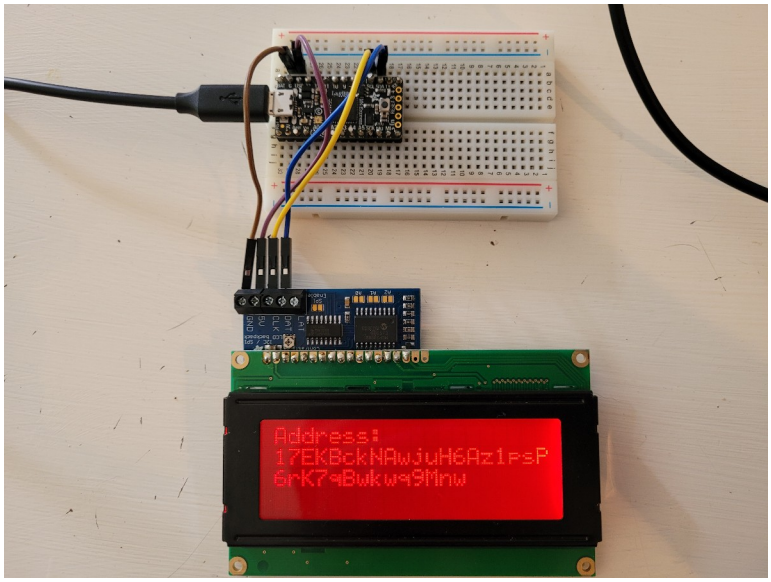
Developing an Offline Bitcoin Address Generator – uBitAddr2 Code Companion

Author	josh
Date	2024-06-15

Overview

Offline or hardware wallets are a common, secure method of generating and storing cryptocurrency keys. Popular commercial solutions such as Trezor, Ledger, or KeepKey offer an easy-to-use solution. But for software developers and the techno-curious, *developing* our own offline wallet is a fun challenge. Combining knowledge of coding, cryptography, and hardware into a fun project makes for great education. uBitAddr2 is a revamp of my original uBitAddr project that takes a new approach to the code. Let's dive into how I created a simple offline wallet using CircuitPython and Adafruit m4 hardware!



Bitcoin address, private key generation with an Adafruit M4 ItsyBitsy and Character LCD screen. The code rotates through the address and private key on screen on a timer.

```
Auto-reload is on. Simply save files over USB to run them or ente
code.py output:
]0;code.py | 9.0.5\
WIF privkey: 5Je8PHUo5YkRsnNvUeG63nmHCz9z1WXErPr3nbmUkWtsKeffhD4
Address public: 1LG1ibbDtSWpL3UiKKvzspZTce2n7tLpNg
]0;Done | 9.0.5\
Code done running.
```

Testing the code by printing the address, key pair to serial output.

Offline Address Generation

Bitcoin & Security Context

This project, like the original, generates a simple address, private key pair on Adafruit M4 microcontrollers. These keypairs are the simplest form of a Bitcoin wallet, and don't have the features that seed phrases and full transaction-signing wallets have. But they demonstrate the core concept of what a Bitcoin wallet is – a public address (public-key hash) that you can send coins to, and the private key, used for signing transactions when we want to spend. By generating and storing the keys offline, on a special-purpose device, we have a higher degree of security than wallets on a phone or PC, where there's greater attack surface. Theoretically, our project here offers the same kind of offline wallet security as commercial solutions like Trezor, Ledger, or KeepKey.

Our offline wallet may be theoretically secure, but it's important to note that DIY with cryptography isn't a great idea. It's hard to get crypto right, and in this case the pitfalls could lead to losing money. That said, this makes for a fun and educational project that explores the topics of cryptography, security, Bitcoin, and coding. Just be aware of the pitfalls of such a project for storing actual funds.

Development Challenges

One of the biggest challenges with developing Bitcoin software, especially on hardware, is finding the necessary cryptographic libraries. Bitcoin, Ethereum, Litecoin, Bitcoin Cash, and other cryptocurrencies rely on several primitives for generating addresses. We'll need hash functions in the form of SHA-256 and RIPEMD160, and the elliptic curve secp256k1. In addition to hashing and ECDSA, we'll need the right encoding libraries to format the address into the characters you see on screen. For Bitcoin, this is base58.
uBitAddr2 takes a slightly different approach than the original for its cryptography needs. The original uBitAddr ported crypto code in the C programming language from Trezor's open source libaries. The rest of the code was developed in CircuitPython. Combining C for my custom code, the Trezor crypto libs, and Python required compiling a custom CircuitPython firmware for the devices. This worked well, but made the development environment more complex.

For this project, I was able to accomplish the goal of writing a pure CircuitPython address generator. This time I did some digging and found several pure Python crypto libaries needed for the task. This does have some tradeoffs – more memory space needed, and some security pitfalls such as side channel attacks. Still, I was able to find the cryptography code needed for Bitcoin address generation and get it running on the Adafruit M4 processors with some minor tweaks. By developing in pure Python, the development process is more straightforward.

The Address Generation Code

The actual address generation code involves several steps. The first, and arguably most important, is to generate the *private key* using a cryptographically secure random source. It's not sufficient to use the built in random function supplied by most programming languages. Instead, we must use a high entropy source suitable for cryptographic keys. Luckily, the Adafruit M4 lines have a secure random source built in. We can access this in CircuitPython using the `os.urandom` function.

We then take our random private key and encode it into a human-readable format called WIF – Wallet Import Format. This is a standard form for individual Bitcoin keys. We first add a standard prefix, for Bitcoin addresses this is `0x80`. We then take our prefix + private key bytes and run that through two rounds of the SHA-256 hash function. The first 4 bytes of that hash are added to the end of the prefix + private key bytes as a *checksum*, to protect users of our WIF key from errors. The final step is to take all of those bytes and encode them in base58 format, which is easier for humans to read and transcribe. The Python code from bitaddrlib.py looks like this:

```
 # Function for generating the WIF private key
    def _generate_privkey_wif(self, key_raw):

        import base58

        # Generate the private key
        wif_prefix_btc_bytes = self.BTC_WIF_PREFIX.to_bytes(1, self.ENDIAN)

        privkey_prefix_bytes = wif_prefix_btc_bytes + key_raw

        wif_checksum_round1 =
adafruit_hashlib.sha256(privkey_prefix_bytes).digest()
        wif_checksum_round2 =
adafruit_hashlib.sha256(wif_checksum_round1).digest()

        wif_raw = privkey_prefix_bytes + wif_checksum_round2[:4]
        wif = base58.b58encode(wif_raw).decode()

        return wif
```

Generating the address requires a few more steps. First, we take the private key bytes (the raw bytes, not the WIF encoded format) and compute the associate public key using elliptic curve cryptography. And, again, we do some hashing. Bitcoin addresses aren't just public keys, they are public-key *hashes*. First, our key goes through a round of SHA-256, and a second round of RIPEMD160. Once we have the pubkey hash, we again add a standard prefix – `0x00` for legacy Bitcoin addresses, and again compute a 2 round SHA-256 checksum for the end of the address. After adding all those bits together, we encode using base58 for our final Bitcoin address! The Python code looks like this:

```
# Function for generating a Bitcoin address
    def _generate_address(self, pubkey_raw):

        from ripemd import ripemd160
        import base58

        round1 = adafruit_hashlib.sha256(pubkey_raw).digest()
        round2 = ripemd160.ripemd160(round1)

        with_prefix = self.BTC_ADDRESS_PREFIX.to_bytes(1, self.ENDIAN) +
round2

        checksum_round1 = adafruit_hashlib.sha256(with_prefix).digest()
        checksum_round2 = adafruit_hashlib.sha256(checksum_round1).digest()
        checksum_length = 4
        checksum_final = checksum_round2[:checksum_length]

        address_raw = with_prefix + checksum_final
        address = base58.b58encode(address_raw).decode()

        return address
```

DIY Address Generation with Hardware

This was a very fun project to work on, and there's room for more improvements to the code. Like the original uBitAddr, I'd like to expand uBitAddr2 to support Bitcoin Cash, Litecoin, Ethereum, or other cryptocurrencies of interest. As we can see, the overall code for generating a simple offline Bitcoin address and private key is reasonably straightforward, but requires overcoming some challenges along the way. Finding the right cryptography libraries, dealing with limited memory space, and understanding the steps to address generation lend some unique challenges. The full code for this project is available on Github, and I encourage giving it a read, modifying, and sharing. I hope this tutorial has inspired you to tinker with code and learn more about development, cryptography, cryptocurrency, and security!