گزارش پروژه اول رمز ارز
امید پناکاری ۸۱۰۱۹۸۵۲۸

قسمت اول: تولید آدرس

در این قسمت می‌خواهیم یک آدرس معتبر برای شبکه‌ی تست‌نت بیت‌کوین پیدا کنیم. کد پایتون مربوط به ساخت یک آدرس رندوم در ادامه آمده است.

```
def generate_address():
    # Generate a random private key
    random_string = os.urandom(30)
    private_key =
hashlib.sha256(hashlib.sha256(random_string).digest()).digest()

    # Create ECDSA public key
    private_key_hex = private_key.hex()
    print(private_key_hex)
    private_key_ecdsa = ecdsa.SigningKey.from_string(private_key,
curve=ecdsa.SECP256k1)
    public_key = private_key_ecdsa.get_verifying_key()
    public_key_bytes = b'\x04' + public_key.to_string()

    # Hash public key
    public_key_hash = hashlib.sha256(public_key_bytes).digest()
    hash_object = hashlib.new('ripemd160')
    hash_object.update(public_key_hash)
    public_key_hash = hash_object.digest()

    # Add testnet prefix 0x6f to public key hash
    public_key_hash = b'\x6f' + public_key_hash

    checksum =
hashlib.sha256(hashlib.sha256(public_key_hash).digest()).digest()[:4]

    # Encode as base58 address
    address = base58.b58encode(public_key_hash + checksum)

    # No compression
    wif = b'\xef' + private_key
    checksum = hashlib.sha256(hashlib.sha256(wif).digest()).digest()[:4]
    wif = base58.b58encode(wif + checksum)
```

```
return wif.decode('ascii'), public_key_bytes.hex(), address.decode('ascii')
```

در ادامه مراحل مختلف این کد توضیح داده می‌شوند:

۱. در ابتدا لازم داریم که یک کلید خصوصی رندوم به طول ۲۵۶ بیت برای خود تولید کنیم. برای این کار معمولا با استفاده از یک source of randomness یک مقدار رندوم به هر طولی ایجاد می‌شود و سپس با اعمال دو بار هش SHA-256 یک کلید خصوصی به دست می‌آید. من نیز در اینجا یک رشته رندوم به طول ۳۰ تولید می‌کنم و از آن هش می‌گیرم.
۲. در ادامه نیاز داریم تا کلید عمومی متناظر با کلید خصوصی تولید شده را به دست بیاوریم. برای این کار از خود کتابخانه‌ی مربوط به الگوریتم Elliptic Curve یعنی ecdsa استفاده شده است. پس از به دست آمدن کلید عمومی یک بایت به صورت 0x04 به ابتدای آن اضافه می‌کنیم که نشان‌دهنده فرمت آن می‌باشد.
۳. برای رسیدن به آدرس از روی کلید عمومی در ابتدا نیاز داریم که دو بار از کلید عمومی به دست‌آمده هش بگیریم. در هش اول از SHA-256 و در هش دوم از Ripemd-160 استفاده می‌کنیم. سپس یک بایت پیشوند به هش به دست آمده اضافه می‌کنیم که نشان‌دهنده‌ی شبکه مورد نظر است. این بایت برای شبکه اصلی بیت کوین برابر 0x00 و برای تست‌نت برابر 0x6f است.
۴. برای عبارت نهایی به دست آمده checksum را محاسبه می‌کنیم که برابر ۴ بایت پایانی پس از اعمال دوبار هش SHA-256 می‌باشد. این checksum به پایان عبارت اضافه می‌شود.
۵. در پایان برروی عبارت به دست آمده یک انکود base58 اعمال می‌کنیم تا آدرس به دست بیاید.

همچنین خواسته شده که کلید خصوصی را در فرمت WIF نمایش دهیم. برای تبدیل کلید به این فرمت ابتدا به اول آن یک بایت پیشوند که دوباره نشان‌دهنده شبکه می‌باشد اضافه می‌کنیم. این بایت برای شبکه اصلی برابر 0x80 و برای تست‌نت برابر 0xef می‌باشد. حال به طور مشابه بالا checksum را به انتها اضافه کرده و انکود base58 می‌گیریم.

در بخش دوم این قسمت خواسته شده که یک vanity address ایجاد کنیم. برای این کار کافی است که با استفاده از روش بالا تا زمانی که یک آدرس با مشخصات مورد نظر ایجاد شود، آدرس تولید کنیم. از آنجا که این فرآیند حتی برای سه کاراکتر نیز بسیار زمان‌بر است، من موفق به تولید آن نشدم و تنها کد آن را نوشته‌ام. همچنین در این بخش برای افزایش تنوع آدرس‌های تولیدی، طول رشته رندوم تولید شده در ابتدا را از ۳۰ به ۴۰۰ افزایش داده‌ام. کد این بخش در ادامه مشاهده می‌شود.

```
target = 'low'
```

```
while True:
    wif, public_key, address = generate_address()
    if address[1:4] == target:
        print("Private key (WIF):", wif)
        print("Public key:", public_key)
        print("Address:", address)
        break
```

قسمت دوم: انجام تراکنش

در ابتدا با استفاده از کد قسمت قبل یک آدرس تولید کرده‌ام که در سوال‌ها از آن استفاده می‌شود. مشخصات این آدرس به صورت زیر می‌باشد.

Private Key:
e1fa538dce3d871e2af1066abbf405f4e8f293bd5403a97a00eeaf0a147f0cd0

WIF:
93JSTrN44NEK6APN58y2Q7WZwxKhirnzH1oeeA1Dq6QD9hFyAx1

Public Key:
046e218f752ce09b893f13dc02b5aeaf576e363883bfac352b03e654bb53128f7e8b
f758a9e55485dff020e7458089236fcfaafdeeab65ce6f9e068d0281896039

Address:
mvr9m741DSUc1Q2M1ncsvrQsGXaHNW5bFy

سوال ۱) در بیت کوین هر تراکنش از تعدادی ورودی و تعدادی خروجی تشکیل شده است. هر کدام از ورودی‌ها به یک خروجی خرج نشده(UTXO) از تراکنش‌های قبلی اشاره می‌کند که فرد سازنده‌ی تراکنش جدید توانایی خرج کردن آن را دارد. هر UTXO دارای یک scriptPubKey می‌باشد که در واقع مجموعه‌ای از دستورات استکی می‌باشد که وظیفه‌ی آن‌ها بررسی درخواست‌های خرج UTXO و تایید یا رد آن‌ها می‌باشد. همچنین فرد خرج‌کننده نیز یک scriptSig ارائه می‌دهد که باز دوباره مجموعه‌ای از دستورات استک می‌باشد که ورودی‌های مورد نیاز برای تایید شدن توسط scriptPubKey را برروی استک قرار می‌دهد. برای مثال در حالت معمول فرد خرج‌کننده، امضای دیجیتال و کلید عمومی خود را برروی استک قرار می‌دهد و سپس scriptPubKey چک می‌کند که کلید عمومی گذاشته شده با فرد مورد نظر مطابقت داشته باشد و همچنین امضای دیجیتال توسط همین فرد انجام شده باشد.

scriptPubKey: [OP_DUP, OP_HASH160, <pub_key>, OP_EQUALVERIFY, OP_CHCKSIG]

scriptSig: [<signature>, <pub_key>]

حال در این سوال برای آن که کسی نتواند از یک UTXO استفاده کند، کافی است که برروی scriptPubKey دستور OP_RETURN را اضافه کنیم. این دستور در هر حالت درخواست را invalidate می‌کند که باعث می‌شود که کسی نتواند آن را خرج کند. برای آن که هر کس بتواند آن را خرج کند نیز بر روی scriptPubKey عملگر OP_CHECKSIG که صرفا چک می‌کند که امضای دیجیتال گذاشته شده توسط کلید عمومی گذاشته شده انجام شده باشد و دیگر برابری کلید عمومی را چک نمی‌کند. بنابراین هرکس می‌تواند با یک امضا با کلید عمومی خودش این UTXO را خرج کند.

آدرس مربوط به دو تراکنش این سوال در ادامه مشاهده می‌شود:

تراکنش اول:

675a121e23d3e367bc6796a9fa5d18440061a47e39d0aa479a8d4854baf847d9

تراکنش خرج:

b396ecb37d955bb9ef858b04a6c38d599a129af2faa699daa6a5ca92005c4efe

سوال ۲) برای این قسمت از عملگر OP_CHECKMULTISIG در scriptPubKey استفاده می‌کنیم. این عملگر به ترتیب ورودی‌های زیر را دریافت می‌کند:

۱. تعدادی امضای دیجیتال که خرج کننده بر روی استک گذاشته
۲. تعداد امضاهای دیجیتال: این مقدار در این مورد برابر ۲ می‌باشد
۳. کلید عمومی همه افراد
۴. تعداد کلیدهای عمومی

در نهایت این عملگر چک می‌کند که امضاهای دیجیتال توسط همین کلیدهای عمومی انجام شده باشند. فرد خرج‌کننده نیز باید امضاهای دیجیتال را با scriptSig بر روی استک قرار دهد.

آدرس تراکنش‌های این سوال در ادامه مشاهده می‌شود:

تراکنش اول:

9bca58e73dd1970bb3fd8204483d4c10f7e3796e277c30eecbb6842588e4a68e

تراکنش خرج:

3a0d4439e936ba1ed0d477a8d77e117f1172ac809e65d77c9e3ecf740cabf94

سوال سوم) در این سوال نیز فرد خرج کننده در scriptSig مقادیر اعداد اول را بر روی استک قرار می‌دهد. سپس در scriptPubKey با استفاده از OP_ADD و OP_SUB اختلاف و جمع دو عدد محاسبه شده و با استفاده از OP_EQUALVERIFY و OP_EQUAL برابری این محاسبات را با جواب اصلی مقایسه می‌کنیم.

آدرس تراکنش‌های این سوال در ادامه مشاهده می‌شود:

تراکنش اول:

813ec0f9370e344f3ca04f3bbe35d7d05fa71a5ea555eefaafa8ba83689fe3dc

تراکنش خرج:

32160dcb55462c4967bc72baec4e12a1d0882c7a311df6caa5192d6cfad9b715

قسمت سوم: استخراج بلاک

برای استخراج بلاک در ابتدا نیاز داریم تا که تراکنش coinbase را ایجاد کنیم. ورودی این تراکنش یک تراکنش تمام 0 می‌باشد و اندیس UTXO آن برابر 0xFFFFFF می‌باشد. داده خواسته‌شده یعنی coinbase data که در اینجا برابر 810198528OmidPanakari می‌باشد را به صورت باینری انکود کرده و بر روی استک scriptSig مربوط به ورودی می‌گذاریم. مشخصات خروجی را نیز به صورت عادی برای آدرس خودمان می‌زنیم. کد ساخت این تراکنش در زیر قابل مشاهده است:

```
    def get_coinbase_transaction(self):
        SelectParams('mainnet')
        txid = '0' * 64
        index = int('0xFFFFFFFF', 16)
        scriptSig =
CScript([bytes.fromhex(self.data.encode('ascii').hex())])
        txin = create_txin(txid, index)
        txin.scriptSig = scriptSig
        txout = create_txout(self.reward,
self.get_P2PKH_scriptPubKey(self.address))
        coinbase_tx = CMutableTransaction([txin], [txout])
        return coinbase_tx
```

برای محاسبه merkle root از آنجا که تنها یک تراکنش دارد کافی است از آن تراکنش دو هش SHA-256 بگیریم. البته کدی که من در این قسمت نوشته‌ام برای چندین تراکنش نیز عمل می‌کند و تراکنش‌ها را دو به دو هش می‌کند تا به یک هش که همان merkle root می‌باشد برسد. این کد در ادامه مشاهده می‌شود:

```
def get_merkle_root(self, transactions):
        if (len(transactions) == 1):
            return transactions[0]
        new_transactions = []
        for i in range(0, len(transactions), 2):
            if i + 1 >= len(transactions):
                new_transactions.append(sha256(sha256(transactions[i] +
transactions[i]).digest()).digest())
            else:
                new_transactions.append(sha256(sha256(transactions[i] +
transactions[i+1]).digest()).digest())

        self.get_merkle_root(new_transactions)
```

در ادامه برای استخراج کافی است که تا زمانی که شرط سختی برقرار نشده(۲ بایت اول برابر هش بلاک صفر نشده). Nonce را یکی یکی افزایش دهیم. در هرگام با استفاده از struct.pack اطلاعات هدر را به مجموعه‌ای از بایت‌ها تبدیل می‌کنیم و از آن هش می‌گیریم. کد مربوط به این بخش نیز در ادامه مشاهده می‌شود:

```
def mine_block(self):
        coinbase_transaction = self.get_coinbase_transaction()
        merkle_root = self.get_merkle_root([coinbase_transaction.GetHash()])
        timestamp = int(time.time())
        nonce = 0
        while True:
            header = struct.pack("<L32s32sLLL", self.version,
bytes.fromhex(self.last_block_hash),
                                 merkle_root, timestamp, self.bits, nonce)
            block_hash = sha256(sha256(header).digest()).digest()
            if block_hash.startswith(b'\x00\x00'):
                print("Block mined successfully!")
                print("Block number:", self.num)
                print("Block hash:", block_hash.hex())
                print("Block header:", b2x(header))
                print("Version:", self.version)
                print("Merkle root:", merkle_root.hex())
                print("Nonce:", nonce)
                print("timestamp:", timestamp)
                return
            nonce += 1
```

بلاک به دست آمده :

```
Block mined successfully!
Block number: 8528
Block hash: 00006e5d4a5bf9bf30b79d17712cf7823d498af900f1ffca7e1746833defe23e
Block header: 0100000000000000e8ef29ff44603ced8534398e1b5d9ff169e5f21d0036bbda3542e24a9420dc414ebf924feff4ae595517c52c5e
db8be5e1246310474fa9bbc5f03ffc0edd71640000011f34a40000
Version: 1
Merkle root: 9420dc414ebf924feff4ae595517c52c5edb8be5e1246310474fa9bbc5f03ffc
Nonce: 42036
timestamp: 1685183758
```