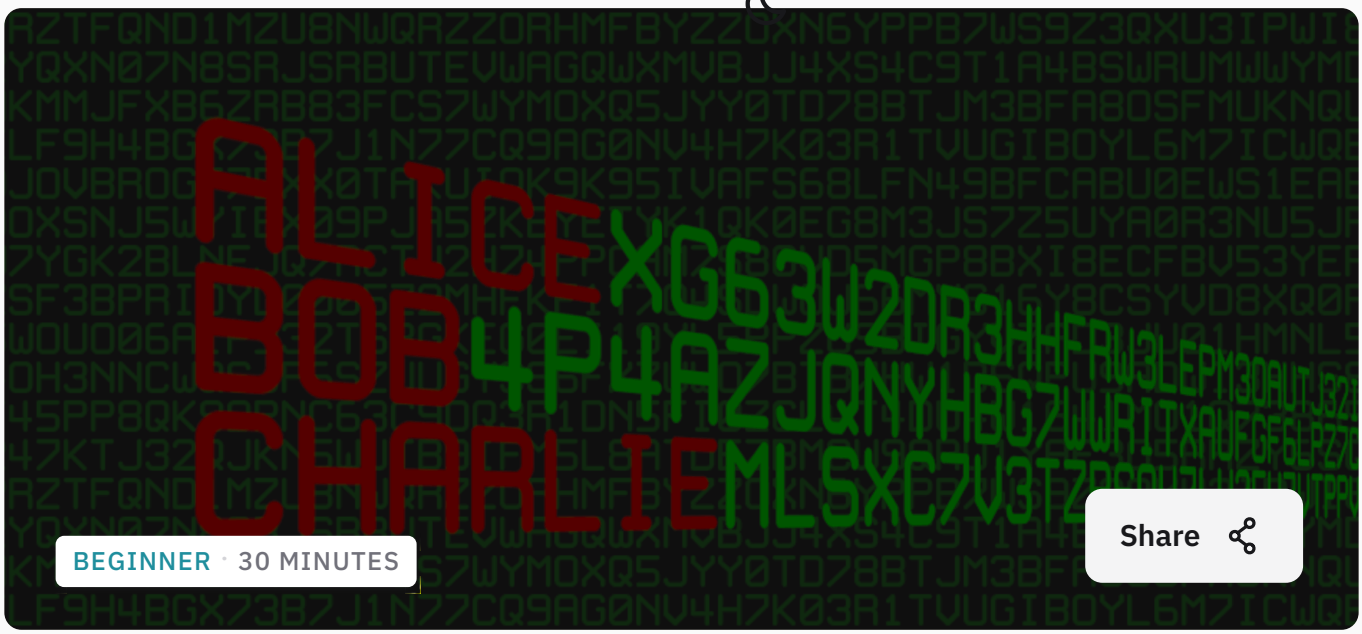

Generating And Securing A Vanity Address Using Rekeying

This tutorial will show you how to use the Python package py-algorand-sdk to generate a new custom vanity Algorand address and how to subsequently secure it by rekeying it to an existing or new Algorand private key.

Steve Ferrigno
Developer Ambassador

March 18, 2021

python addresses SDK rekeying

Python

Requirements

- Python environment with ability to install packages via PIP
- Access to an Algorand node for sending transactions

This is a relatively simple tutorial and intended as a fun way to get started with the Python Algorand SDK. If you have ever done any basic programming you should be able to complete this tutorial.

Background

Vanity addresses are public address which start with a user-define prefix (e.g. **DODO**QWK...5RZK3PU) and are used by both individuals and organisations alike. Other than suggesting the owner of the address has a technical personality or background they do not hold any additional value or security to any other address. In fact by generating an address, discarding it, then generate another address you'll actually be reducing the entropy and technically reducing the security of the address.
However one of the great features of Algorand is the ability to rekey your public Algorand addresses! This means you can have your own name at the start of a new public Algorand address and secure it to an existing or new private key you generated securely! I personally find these useful for when I'm testing. I have a set of test accounts with the prefixes ALICE, BOB, and CHRIS which I frequently use due to being easily identifiable.

> **Info**
>
> Vanity addresses are not unique to Algorand and have been used with other cryptocurrencies for many years.

It should be stressed that this can become incredibly computationally difficult depending on the length of the prefix you're after. Stick to a maximum of 5 characters and you should be able to calculate an address within a few minutes to hours depending on your computers hardware. Without going into the technical details, no this isn't a viable way to figure out the private key of someones public Algorand address!

This tutorial has been written so that you can follow along inside the Python IDLE, but if you'd prefer to create a `vanity_generator.py` script with some additional features please check the script example at the end of this tutorial.

Steps

- Prerequisites
- Step 1 - The Basics
- Step 2 - Find Your Vanity Address
- Step 3 - Vanity Address Mnemonic
- Step 4 - Configure Authoritative Account
- Step 5 - Connecting To A Node

Step 6 - Rekeying

Step 7 - Testing

Conclusion

Script Example

Prerequisites

You will need to have access to an Algorand node for the later part of this tutorial, as rekeying and testing the address will require sending transactions to a network. This can either be your own go-algorand instance, a sandbox environment, or access to the PureStake Developer API.

- How do I obtain an algod address and token?

It is assumed you already have a working Python environment configured. Next we need to make sure py-algorand-sdk is installed. This is the official Python library from Algorand that makes interacting with the Algorand network from Python a lot simpler.

From a terminal type the following PIP command.

```
user@computer:~/vanity_rekey$ pip install py-algorand-sdk
```

Step 1 - The Basics

Now we have the py-algorand-sdk library installed, enter into a Python3 IDLE environment and follow along.

Before we can use any of the helpful methods, we must import the account class from algosdk.

```
>>> from algosdk import account
```

Then to generate a single private key and address and print them to the screen we can call `generate_account()` from the `account` class we just imported.

```
>>> vanity_private_key, vanity_address =
account.generate_account()
>>> print("Private key:", vanity_private_key)
Private key:
P2V8bh3RM54rNsS9dQC5DTAcgalvjZ73fNR9O/WcB3K/XCx+n
T0LTWi5amaK0m4hWsANSMp0v7VN6c6Zk1vhNg==
>>> print("Address:", vanity_address)
Address:
X5OCY7U5HUFU22FZNJTIVUTOEFNMADKIZJ2L7NKN5HHJTE234
E3IHIZT3M
```

Step 2 - Find Your Vanity Address

Chances are the public Algorand address you generated above doesn't include the prefix you're after, so let's define it now and start the search to find the one you want. It's worth mentioning that there are also certain character which won't ever be found at the start of an address, for example the numbers 0, 1, or 8. It's worth testing with a shorter 1 to 3 character prefix e.g. "AA" or "ABC" first before wasting hours.

!!!! warning
This process can take a very long time depending on your hardware. It's not recommended for anything over 4 to 5 characters in length.

```
>>> prefix = "DODO"
>>> while (not
vanity_address.startswith(prefix)):
...     vanity_private_key, vanity_address =
account.generate_account()
...
```

This may take some time to return. Your computer is now generating tens to hundreds of thousands of accounts until it finds one that starts with your desired prefix. Remember that this is entirely luck based, there's always a chance you won't find what you're after in a reasonable amount of time.

Once the `>>>` prompt is visible again you can see the details of the account by printing out the `vanity_private_key` and `vanity_address` variables like before.

```
>>> print("Private key:", vanity_private_key)
Private key:
O533OJJz7+QkbkrIBnzbX8omAFQlRBgwzu7M+qbqreQbhuhZQ
UuboAHvdFofMsGM+dIPZU90RXJSKRF/EWXFnw==
```

```
>>> print("Address:", vanity_address)
Address:
DODOQWKBJON2AAPPORNB6MWBRT45ED3FJ52EK4SSFEIX6ELFY
WP5RZK3PU
```

Step 3 - Vanity Address Mnemonic

From here you can import and use the `mnemonic` class from algosdk and call the `from_private_key()` method to convert the private key string into a 25-word mnemonic you're familiar with.

```
>>> from algosdk import mnemonic
>>> print("Mnemonic:",
mnemonic.from_private_key(vanity_private_key))
Mnemonic: polar taxi broccoli decrease ten
decrease illness engine suit useless unit planet
eternal abandon click during adapt decide jazz
proud evil kingdom century abstract empty
```

Step 4 - Configure Authoritative Account

Finally you'll want to secure your new vanity address to your existing or a new private key. Algorand has an excellent feature called rekeying, which allows an Algorand account holder to replace the authoritative private key for their public address.

We will generate a new account using the Python IDLE and use that as our "existing account", but if you already have an existing account you could use those details. We will then rekey the vanity address to make our existing account the authorative key.

```
>>> from algosdk import account, mnemonic
>>> existing_private_key, existing_address = account.generate_account()
>>> # NOTE: If you were to import an existing account you could use the following commands instead.
>>> existing_private_key = mnemonic.to_private_key('enter your twenty five word mnemonic here')
>>> existing_address = account.address_from_private_key(existing_private_key)
```

Step 5 - Connecting To A Node

The next step will depend on how you are accessing the Algorand node. If you have a Sandbox environment running locally you should be able to use the provided code without changes. If you have setup an go-algorand algod node or are using the PureStake Developer API you will have to

adjust the `algod_token` and `algod_address` appropriately.

- How do I obtain an algod address and token?

In this code we're setting up `algod_client` to connect to an Algorand node, which we will use to retrieve suggested parameters from the network and for when we send a transaction to the network.

```
>>> from algosdk import algod
>>> algod_token =
'aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa'
>>> algod_address = 'http://localhost:4001'
>>> algod_client = algod.AlgodClient(algod_token,
algod_address)
```

Step 6 - Rekeying

Now let's create the rekey transactions (A zero Algo payment transaction with the `rekey_to` parameter set), sign it with the vanity address private key, and send it to the network. The transaction data will require some relatively up-to-date data related to the network, since the transaction needs to submit a "first" and "last" round that the transaction will be valid for and the genesis hash, so we'll use `algod_client` to provide this using the `suggested_params()` method first.

Note

In order to complete the rekey transaction a minimum of 1000 microAlgo for the fee is required, and a further 1000 microAlgo to demonstrate the use of our rekeyed account. If you would like to follow along using the TestNet and don't currently have any Algo, use the faucet to send some to your vanity address. If you're using a Sandbox environment, send some Algo from one of the accounts which come setup with Algod.

```
>>> from algosdk import transaction
>>> params = algod_client.suggested_params()
>>> txn_rekey =
transaction.PaymentTxn(vanity_address,
params['minFee'], params['lastRound'],
params['lastRound']+1000,
params['genesishashb64'], vanity_address, 0,
rekey_to=existing_address)
>>> stxn_rekey =
txn_rekey.sign(vanity_private_key)
>>> algod_client.send_transaction(stxn_rekey)
'BE6VSX5LHMSVRMLCT3BMS4H5TLJXJGDNEENAUOVU5UYZ6KHO
OP6A'
```

Step 7 - Testing

And there you have it, the vanity account is now set to have the existing account as an authoritative address. Let's prove this by making a transaction sending 1 Algo to our existing address from the vanity

address, but signing it with our existing_private_key.

```
>>> params = algod_client.suggested_params()
>>> txn_test =
transaction.Payment(vanity_address,
params['minFee'], params['lastRound'],
params['lastRound']+1000,
params['genesishashb64'], existing_address,
1000000)
>>> stxn_test =
txn_test.sign(existing_private_key)
>>> algod_client.send_transaction(stxn_test)
'IXK7YGZU4GNYIZEKNYJGV5N6ERD4WCUH7GQK4HQAKGFIPM76
GPVA'
```

Conclusion

Whilst this tutorial should have helped you learn some basics of the algosdk, it must be stressed that you test and understand the full implications of rekeying before doing any of this to your real accounts. At the time of writing this the popular AlgoSigner browser extension doesn't support rekeyed addresses, so you won't be able to use it for signing transactions. So be prepared to do it manually for the time being.

You have now hopefully found a new vanity address that you can be proud of and further secured it by using the awesome rekeying feature! Now you can be the envy of your friends when you give them your custom public Algorand address the next time they need to send you some Algo. :sunglasses:

Script Example

The following code example contains a simple while-loop executing `account.generate_account()` until the desired address prefix is found. Additionally it'll give you a running total of attempts and a rough attempts-per-second value.

```
#!/usr/bin/env python3

from algosdk import account, mnemonic
import time

# Desired prefix for address.
PREFIX = "DODO"

# Variables containing Private Key and Address
PRIVATE_KEY = ""
ADDRESS = ""

# Keep track of attempts and start time.
ATTEMPT = 0
TIME_START = time.time()

# Keep looping until desired address is found.
while (not ADDRESS.startswith(PREFIX)):
    TIME_DIFF = time.time() - TIME_START
    print(f" {ATTEMPT} ({ATTEMPT /
TIME_DIFF:.2f}/sec) ", end="\r")
    PRIVATE_KEY, ADDRESS =
account.generate_account()
    ATTEMPT += 1
```

```
# Display your new address, private key, and
mnemonic
print()
print(f"Found after {ATTEMPT} attempts and
{TIME_DIFF:.2f} seconds")
print(ADDRESS)
print(PRIVATE_KEY)
print(mnemonic.from_private_key(PRIVATE_KEY))
print()
```

This will produce an output similar to this:

```
user@computer:~/vanity_rekey$
./address_generator.py
 2413655 (7913.30/sec)
Found after 2413656 attempts and 305.01 seconds
DODOQWKBJON2AAPPORNB6MWBRT45ED3FJ52EK4SSFEIX6ELFY
WP5RZK3PU
O533OJJz7+QkbkrIBnzbX8omAFQlRBgwzu7M+qbqreQbhuhZQ
UuboAHvdFofMsGM+dIPZU90RXJSKRF/EWXFnw==
```

Algorand™

DISCORD FORUM

Stay up to date

Your email address

Discover

Documentation

Developer Blog

Ecosystem Projects

MainNet Metrics

Partner Sites

Algorand

Algorand Foundation

Algorand Wallet

Support Contact Us Terms of Use Privacy Policy

English