How to Generate a Bitcoin Address — Step by Step

Jordan Baczuk Follow
Jun 6, 2018 · 8 min read ★

. . .

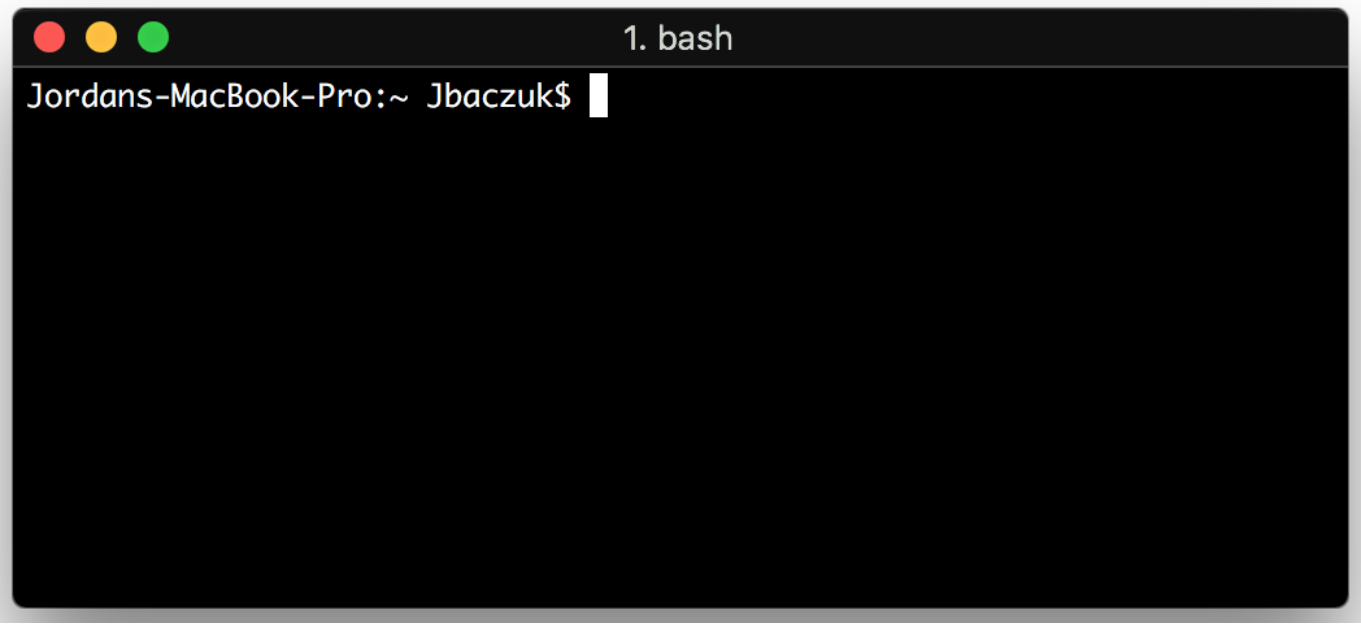
Here is a bash script that does what is outlined below: https://bit.ly/2MIgeOD

Introduction

This is a hands on, technical guide about the generation of Bitcoin addresses including private and public keys, and the cryptography involved.

> *Learn more and join people in 22 countries around the world in my course on how to Become a Bitcoin + Blockchain Programmer.*

This guide will walk you through all the steps to generate a Bitcoin address using the command line on a Mac. Similar steps should be possible on other operating systems using similar cryptographic tools. Lines starting with `$` denote terminal commands, which you can type and run (without the `$` of course).



Mac Terminal (iTerm 2)

Dependencies

- `brew` — Installation: https://brew.sh/
- `pip` — Installation: `sudo easy_install pip`
- `libressl` — Installation: `brew install libressl`
- `base58` — Installation: `pip install base58`

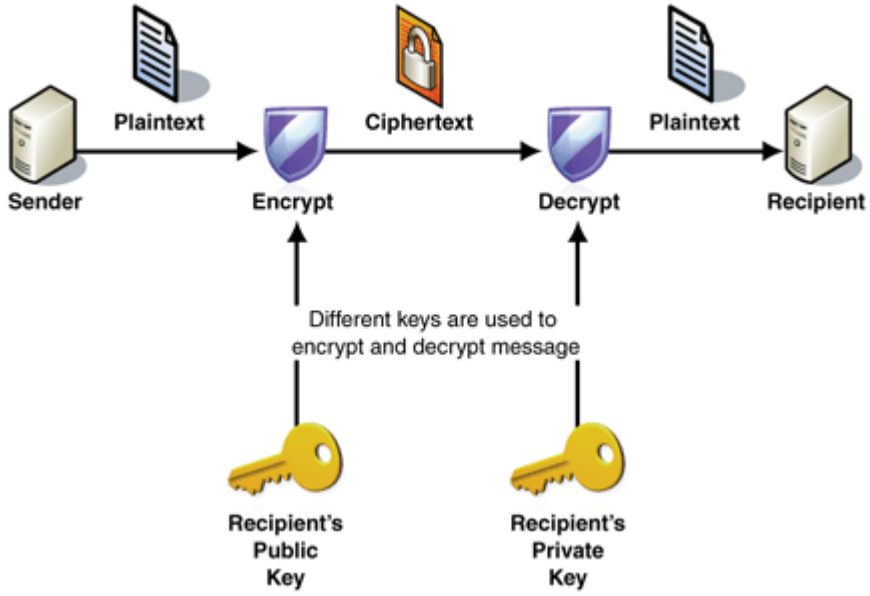
> *Note: To do the contained `openssl` cli commands, I installed `libressl` in order for some of the elliptic curve commands to work as the current version of openssl cli on mac has a bug.*

Cryptography Primer

Public Key Cryptography

Or asymmetric cryptography, is a type of cryptography that uses key pairs, each of which is unique. The pair of keys includes a public key and a private key. This is the type of

cryptography that Bitcoin uses to control funds. A public key can be generated from a private key, but not vice-versa (computationally too difficult). Also, something encrypted with a private key can be decrypted with the public key, and vice-versa, hence they are asymmetric.



Assymetric Encryption

Use Cases

- Encryption: When a user has a public key, a message can be encrypted using a public key, which can only be read by the person with the private key. This also works in reverse.
- Digital Signatures: A user can, with their private key and a hash of some data, use a digital signature algorithm such as ECDSA, to calculate a digital signature. Then, another user can use the algorithm to verify that signature using the public key and the hash of the same data. If it passes, this proves a user did in fact submit a specific message, which has not been tampered with.
- Digital Fingerprint: Is a way to represent an arbitrarily large data set by computing the hash of it to generate a fingerprint of a standard size. This fingerprint would be so difficult to replicate without the same exact data, which can be assumed to have not been tampered with.

Private keys are what prove you can send Bitcoin that has been sent to you. It is like the password to your bank account. If you lose it or someone else gets a hold of it, you're toast.

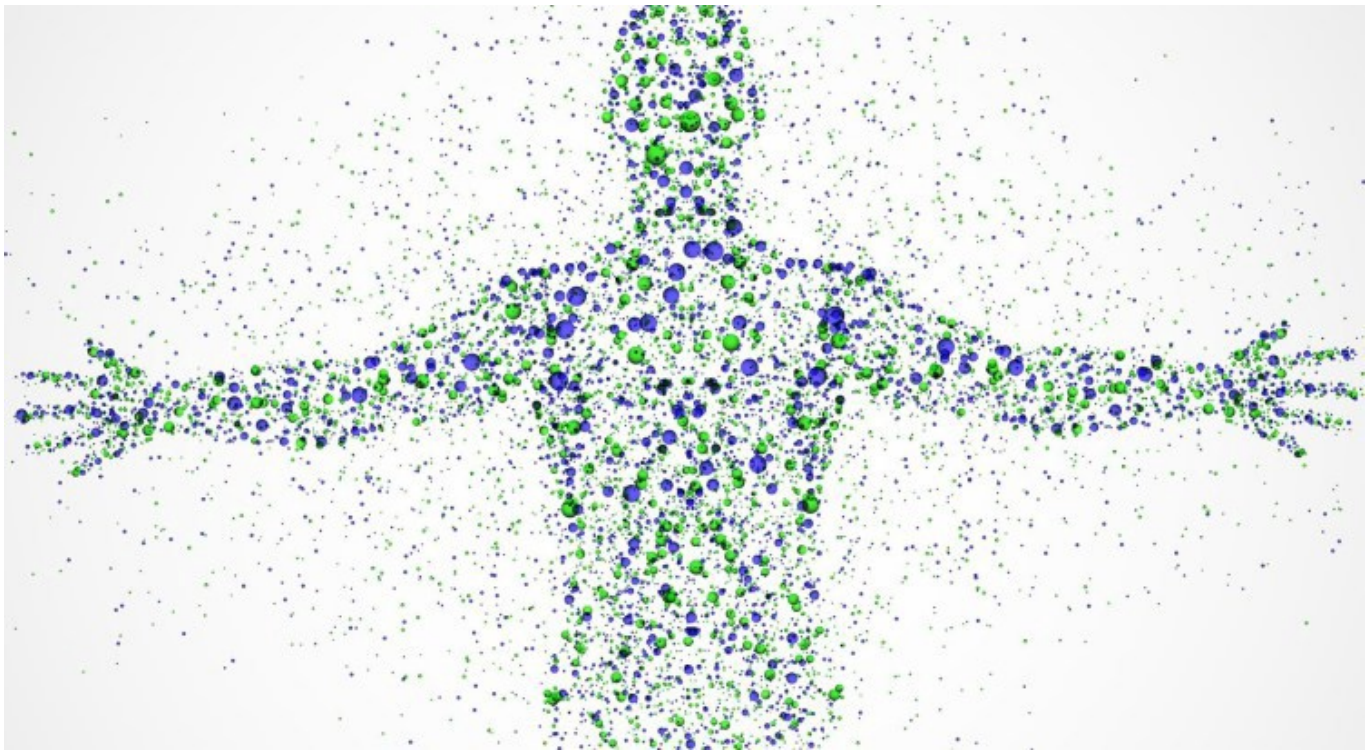
Public keys help people know how to send you Bitcoin.

Creating a Bitcoin Address

Private Key Generation

Private keys can be any 256 bit (32 byte) value from `0x1` to `0xFFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF BAAE DCE6 AF48 A03B BFD2 5E8C D036 4140` .[1]

The total possible number of private keys is therefore 2^{256} or 1.16×10^{77}. Imagine the total number of atoms in your body, then imagine that each of those atoms is an earth. The total number of atoms on all of those earths is about 7×10^{77}.[2] There is virtually no chance that your random private key will ever be generated randomly or found by someone else.



A common (but not the most secure) way of creating a private key is to start with a seed, such as a group of words or passphrases picked at random. This seed is then passed through the SHA256 algorithm, which will always conveniently generate a 256 bit

value. This is possible because every computer character is represented by an integer value (see ASCII and Unicode).

> *Note: SHA256 is a one-way, deterministic function meaning that it is easy to compute in one direction, but you cannot reverse it. In order to find a specific output, you have to try all the possible inputs until you get the desired output (brute forcing) and it will always produce the same output given the same input, respectively.*

The seed can be used to generate the same private key if the same hashing algorithm is used in the future, so it is only necessary to save the seed.

```
$ echo "this is a group of words that should not be considered
random anymore so never use this to generate a private key" |
openssl sha256

a966eb6058f8ec9f47074a2faadd3dab42e2c60ed05bc34d39d6c0e1d32b8bdf
```

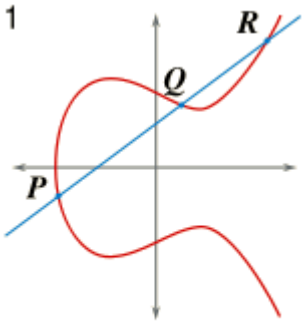
This private key is in hexadecimal or base 16. Every 2 digits represents 8 bits or 1 byte. So, with 64 characters, there are 256 bits total.

Public Key Generation

Public keys are generated from the private keys in Bitcoin using elliptic curve (`secp256k1`) multiplication using the formula $K = k * G$, where K is the public key, k is the private key, and G is a constant called the Generator Point[4], which for `secp256k1` is equal to:

```
04 79BE667E F9DCBBAC 55A06295 CE870B07 029BFCDB 2DCE28D9 59F2815B
16F81798 483ADA77 26A3C465 5DA4FBFC 0E1108A8 FD17B448 A6855419
9C47D08F FB10D4B8
```

It doesn't seem to be known how this point was chosen by they designers of the curve. Also, this algorithm is a one-way algorithm, or a "trap door" function so that a private key cannot be derived from the public key. It is important to note that elliptic curve multiplication is not the same as scalar multiplication, though it does share similar properties.



Elliptic Curve Example

To do this in the terminal from our private key earlier,

```
$ openssl ec -inform DER -text -noout -in <(cat <(echo -n
"302e0201010420") <(echo -n
"a966eb6058f8ec9f47074a2faadd3dab42e2c60ed05bc34d39d6c0e1d32b8bdf")
<(echo -n "a00706052b8104000a") | xxd -r -p) 2>/dev/null | tail -6 |
head -5 | sed 's/[ :]//g' | tr -d '\n' && echo

043cba1f4d12d1ce0bced725373769b2262c6daa97be6a0588cfec8ce1a5f0bd092f
56b5492adbfc570b15644c74cc8a4874ed20dfe47e5dce2e08601d6f11f5a4
```

This public key contains a prefix `0x04` and the x and y coordinates on the elliptic curve `secp256k1`, respectively.

Compressed Public Key

Most wallets and nodes implement compressed public key as a default format because it is half as big as an uncompressed key, saving blockchain space. To convert from an uncompressed public key to a compressed public key, you can omit the x value because the y value can be solved for using the equation of the elliptic curve: $y^2 = x^3 + 7$. Since the equation solves for y^2, the right side of the equation could be either positive or negative. So, `0x02` is prepended for positive y values, and `0x03` is prepended for negative ones. If the last binary digit of the y coordinate is 0, then the number is even, which corresponds to positive. If it is 1, then it is negative. The compressed version of the public key becomes:

```
023cba1f4d12d1ce0bced725373769b2262c6daa97be6a0588cfec8ce1a5f0bd09
```

The prefix is `0x02` because the y coordinate ends in `0xa4`, which is even, therefore positive.

Address Generation

There are multiple Bitcoin address types, currently `P2SH` or pay-to-script hash is the default for most wallets. `P2PKH` was the predecessor and stands for Pay to Public Key Hash. Scripts give you more functionality, which is one reason why they are more popular. We'll first generate a `P2PKH` original format address, followed by the now standard `P2SH`.

Hash

The public key from the previous output is hashed first using `sha256` and then hashed using `ripemd160`. This shortens the number of output bytes and ensures that in case there is some unforeseen relationship between elliptic curve and sha256, another unrelated hash function would significantly increase the difficulty of reversing the operation:

```
$ echo
023cba1f4d12d1ce0bced725373769b2262c6daa97be6a0588cfec8ce1a5f0bd09 |
xxd -r -p | openssl sha256
(stdin)=
8eb001a42122826648e66005a549fc4b4511a7ad3fc378221aa1c73c5efe77ef

$ echo
8eb001a42122826648e66005a549fc4b4511a7ad3fc378221aa1c73c5efe77ef |
xxd -r -p | openssl ripemd160
(stdin)= 3a38d44d6a0c8d0bb84e0232cc632b7e48c72e0e
```

> *Note that since the input is a string, the `xxd -r -p` will convert the hex string into binary and then output it in hexdump style (ascii), which is what the openssl hashing functions expect as input.*

Encoding

Now that we have hashed the public key, we now perform `base58check` encoding. Base58check allows the hash to be displayed in a more compact way (using more letters of the alphabet) while avoiding characters that could be confused with each other such as 0 and O where a typo could result in your losing your funds. A checksum is applied to make sure the address was transmitted correctly without any data corruption such as mistyping the address.

Value	Character	Value	Character	Value	Character	Value	Character
0	1	1	2	2	3	3	4
4	5	5	6	6	7	7	8
8	9	9	A	10	B	11	C
12	D	13	E	14	F	15	G
16	H	17	J	18	K	19	L
20	M	21	N	22	P	23	Q
24	R	25	S	26	T	27	U
28	V	29	W	30	X	31	Y
32	Z	33	a	34	b	35	c
36	d	37	e	38	f	39	g
40	h	41	i	42	j	43	k
44	m	45	n	46	o	47	p
48	q	49	r	50	s	51	t
52	u	53	v	54	w	55	x
56	y	57	z				

Base58 Encoding Table

Address format

Bitcoin `P2PKH` addresses begin with the version byte value `0x00` denoting the address type and end with a 4 byte checksum. First we prepend the version byte (prefix) to our

public key hash and calculate and append the checksum before we encode it using `base58` :

```
$ echo 003a38d44d6a0c8d0bb84e0232cc632b7e48c72e0e | xxd -p -r |
base58 -c && echo
16JrGhLx5bcBSA34kew9V6Mufa4aXhFe9X
```

> *Note: `-c` denotes a checksum is to be applied. The checksum is calculated as checksum = SHA256(SHA256(prefix+data)) and only the first 4 bytes of the hash are appended to the end of the data.*

The resulting value is a `P2PKH` address that can be used to receive Bitcoin:
16JrGhLx5bcBSA34kew9V6Mufa4aXhFe9X

Pay-to-Script Hash

The new default address type is a pay-to-script-hash, where instead of paying to a pubKey hash, it is a script hash. Bitcoin has a scripting language, you can read more about it here. Basically it allows for things like multiple signature requirements to send Bitcoin or a time delay before you are allowed to send funds, etc. A commonly used script is a P2WPKH (Pay to Witness Public Key Hash): `OP_0 0x14 <PubKey Hash>` where the PubKey Hash is the `RIPEMD160` of the `SHA256` of the public key, as before, and 0x14 is the number of bytes in the PubKey Hash. So, to turn this script into an address, you simply apply `BASE58CHECK` to the `RIPEMD160` of the `SHA256` of the script `OP_0 0x14 <PubKey Hash>` except you prepend `0x05` to the script hash instead of `0x00` to denote the address type is a `P2SH` address.

```
$ echo 00143a38d44d6a0c8d0bb84e0232cc632b7e48c72e0e | xxd -r -p |
openssl sha256
(stdin)=
1ae968057eaef06c3e13439695edd7a54982fc99f36c3aa41d8cc41340f30195

$ echo
1ae968057eaef06c3e13439695edd7a54982fc99f36c3aa41d8cc41340f30195 |
xxd -r -p | openssl ripemd160
(stdin)= 1d521dcf4983772b3c1e6ef937103ebdfaa1ad77

$ echo 051d521dcf4983772b3c1e6ef937103ebdfaa1ad77 | xxd -p -r |
base58 -c && echo
```

```
34N3tf5m5rdNhW5zpTXNEJucHviFEa8KEq
```

If you like the article, check out my course on how to Become a Bitcoin + Blockchain Programmer.

. . .

References

- [1] https://en.bitcoin.it/wiki/Private_key
- [2] https://education.jlab.org/qa/mathatom_05.html, https://education.jlab.org/qa/mathatom_04.html
- [3] https://crypto.stackexchange.com/questions/1145/how-much-would-it-cost-in-u-s-dollars-to-brute-force-a-256-bit-key-in-a-year
- [4] https://en.bitcoin.it/wiki/Secp256k1

)

Thanks to Mark Wardle.

Bitcoin Tutorial Programming Education Wallet