Programming Bitcoin Transaction Scripts

A gentle introduction

roland.kofler@gmail.com

Table of Contents

- Why to Deal with Transaction Scripts?
- Thinking in Transaction Scripts
  - Provably Unspendable Transaction
  - Anyone-Can-Spend Transaction
  - Generation Transaction
  - Simple Transaction
  - Reference for this Chapter
- Anatomy of a Transaction
  - Lock Time
  - Transaction Inputs
  - Transaction Outputs
- Build your First Transaction
- Appendix I - Useful Resources

Why to Deal with Transaction Scripts?

We are at the verge of a technological disruption. Modern market society is coordinated by trade and price, by contracts and possession. The Bitcoin technology promises to reorder all this. Increasing the efficiency of transactions, tearing down old power monopolies in finance and law. While the future remains uncertain, Satoshi Nakamoto and the earlier Bitcoin community kindled an idea that can not be reversed.
New distributed and transparent products and services will emerge on the block chain. The Bitcoin community invented valid concepts for contracts, escrows and insurance, inheritance and property rights based on cryptography and the blockchain.
Central to all this innovations are the transaction scripts[1].

[1] https://en.bitcoin.it/wiki/Contracts

Thinking in Transaction Scripts

A new transaction is valid if the transaction scripts of its input field and the transaction script of its predecessing transaction validates to true.

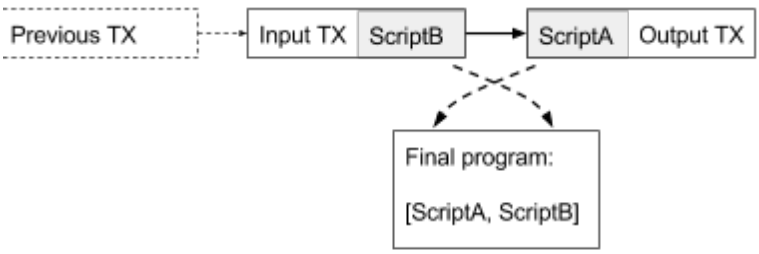


Figure: A output tx is valid only if a script program results in a boolean `TRUE`. The script is composed of two scripts blocks, the execution order of the scripts is: first ScriptA from OutputTX is executed, then ScriptB from InputTX.

The scripting language is stack-based, this means that each data, input or output is put on a stack of other data.
The script of OutputTX is executed first: i.e. the redeemer's code. But it is not possible to stop the script before it is executed entirely and marking the transaction as valid. Therefore this is no security issue. Finally the InputTX scriptB is executed and when the program terminates its return value determines if the OutputTX is considered valid.

Provably Unspendable Transaction

The null transaction, probably the simplest one.

This transaction will always be invalid. Whatever code is in ScriptA, when ScriptB is executed, the `OP_RETURN` op-code stops the execution of the transaction script and validates to `FALSE`.
This pattern is often used to encode data in the blockchain. After the `OP_RETURN` you can insert arbitrary data. The advantage is that the simple bitcoin nodes can prune the transaction saving memory, while full nodes will hold it. This is considered good behavior when 'misusing' the blockchain for storing data.

Anyone-Can-Spend Transaction

If the outputscript of the first transaction is empty, a redeeming second transaction can simply put `TRUE` on the stack so that the transaction is valid. Arguably anyone can do this if he is lucky to spot such an 'empty' transaction.
Anyone-Can-Spend are currently non-standard, and not broadcasted in the network. But they can play an important role in future. For example with Fidelity Bonds[2].

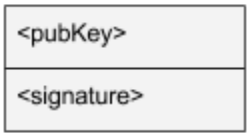
Generation Transaction

Normally a Bitcoin transaction is validated against the previous transaction (the input transaction). When a miner wins in the hashing competition and redeems his price, there is no previous transaction. He then simply creates an Input TX with the publicly known mining fee. Redeeming such a transaction is allowed to anyone who can provide a valid signature of the public key in the Input TX, i.e. the miner himself.

Now it is time to execute the script step by step. Remember that a transaction script is executed on a stack. At the beginning the stack is empty and the program is:

```
<signature> <pubKey> OP_CHECKSIG
```

1. First the program reads the first token `<signature>` and since it is data, it puts it on the stack. `<signature>` should be a piece of data encrypted with the private key of the authorized redeemer
2. Now the second token is also data, so we put `<pubKey>` on the stack. `<pubKey>` is the public key (the unhashed bitcoin address) of the redeemer. At the end of this two operations the stack looks like this:



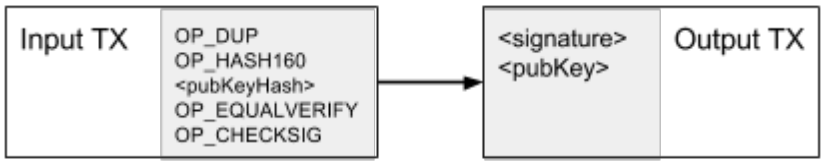
3. The next token is an operation. `OP_CHECKSIG` takes the first argument `<pubKey>` from the stack and validates the second argument `<signature>`. Basically it tries to open `<signature>` with the public key `<pubKey>`. If it succeeds it returns true, thus making the transaction Output TX valid.

We have seen: only the owner of the private key can redeem the Generation Transaction, he is the lucky miner.

Simple Transaction

[2] "This mechanism may be used in the future for fidelity bonds to sacrifice funds in a provable way." https://en.bitcoin.it/wiki/Script#Anyone-Can-Spend_Outputs

The standard transaction script looks like this:

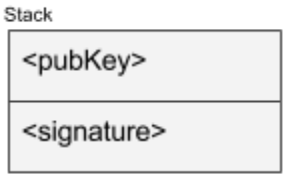


Therefore the full script looks like this:

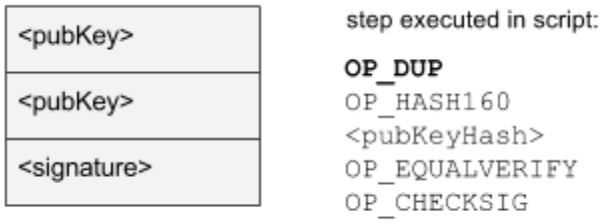
```
<signature> <pubKey> OP_DUP OP_HASH160 <pubKeyHash>
OP_EQUALVERIFY OP_CHECKSIG
```

Lets step through the execution of the program:

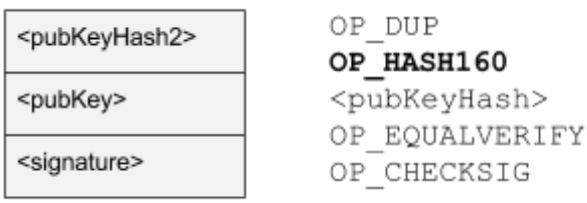
1. the two data tokens are put on the stack in the first two steps of the program



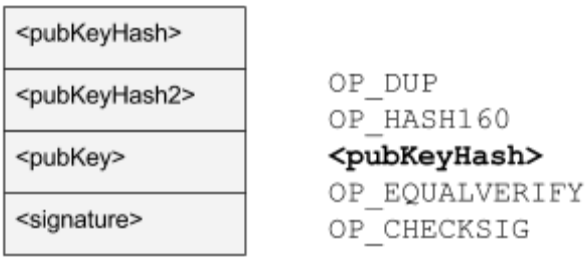
2. The operation `OP_DUP` duplicates the first element on the stack, so that we get:



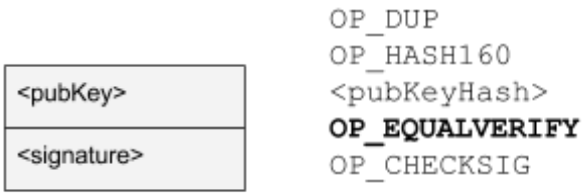
3. The operation `OP_HASH160` hashes the first element on the stack, so that we get:



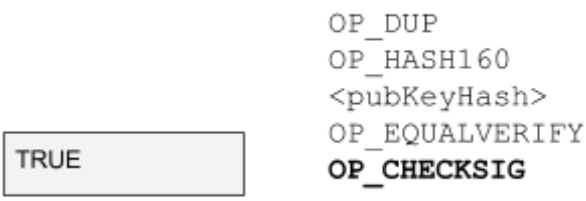
4. `<pubKeyHash>` is put on the stack:



5. The operation `OP_EQUALVERIFY` compares the first two elements of the stack, in reality this is a composed operation: `OP_EQUAL` and `OP_VERIFY` are executed. `OP_EQUAL` puts `TRUE` on the stack if the two elements are the same. `OP_VERIFY` marks a transaction valid, if the top stack element is true. And removes the top stack element if its `TRUE`, if its false it leaves it there.
   Generally a command executed on the stack takes its parameters from the stack, and puts its result at the stack. So `OP_VERIFY` behaves normally if its `FALSE`, because it leaves the result on the stack, but it behaves abnormally when its `TRUE`. It removes the result `TRUE` from the stack to continue with the next parameters.
   The result of a positive validation is therefore:



6. Finally the signature on the stack is verified with `OP_CHECKSIG` in the same way as in the Generation Transaction. `TRUE` is returned if the check succeeds.



The simple transaction therefore is not so simple at all. First we prove that the public key that the redeemer states is the same as we had in the Input TX, than we verify if the redeemer has the right secret key by verifying the signature of the transaction.

Reference for this Chapter

Description of opcodes and details of the scripts https://en.bitcoin.it/wiki/Script

I'm writing this to learn, not for profit. If you feel like to honor my work, I'd take a hat tip of 1 Euro or 1 Dollar and say thank you: 1MxzAKcsTxie4aJn2ncL8FR8ZYvQHABtBH
4 tips in 6 days/ somebody thinks this has value, I'm happy.

Anatomy of a Transaction

What we called [scriptA, scriptB] in the Bitcoin protocol is refered as [scriptSig, scriptPubKey], probably because scriptSig provides most often a proof that scriptPubKey has to verify. In the same way a public key (pubKey) would verify a signature (sig).

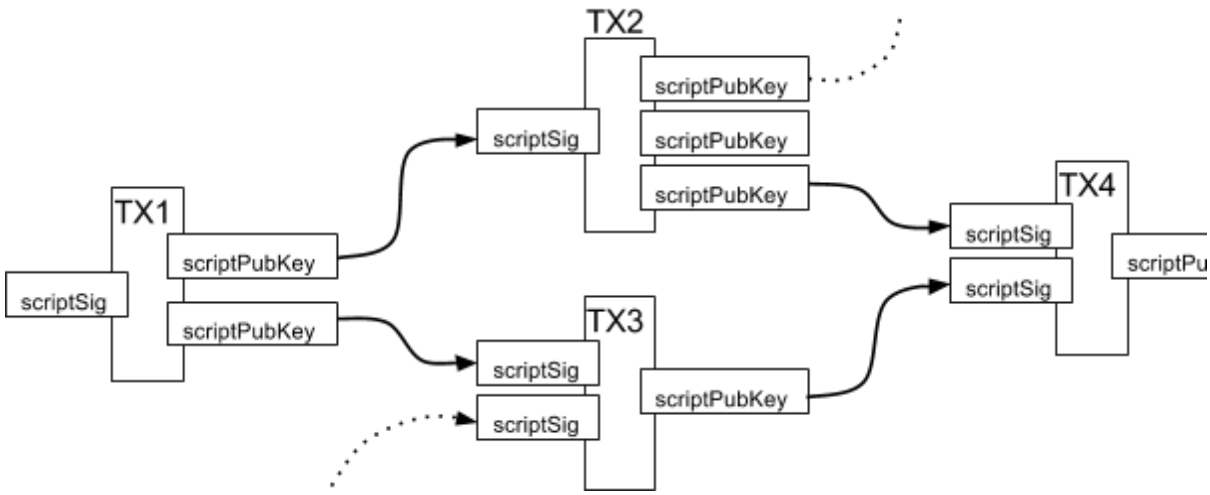


Figure: A chain of transactions with different numbers of inputs and outputs. The first transaction TX1 has no input, it is a *generation transaction* (a reward for your mining effort), it then has two outputs, which is unusual for a mined transaction, but not impossible, One goes to TX2 which has 3 outputs, the uppermost goes to a transaction not depicted, the second is unspent, and the third goes to transaction four. Try to explain the rest for yourself.

Lets take a look at a real transaction on the block chain. For example the transaction with the following transaction hash:
00144cd602ef6ed674f64ec0f229d9fb2a195e35c7dc05a49ce7d74903aef93e.
Here is a JSON format version of the binary code of the transaction:

```
{
  "hash":"00144cd602ef6e...",   The hashcode serves as an ID for the transaction
  "ver":1,                      A versioning for future enhancements, currently always 1
  "vin_sz":1,                   Number of incoming transactions
  "vout_sz":2,                  Number of outgoing transactions
  "lock_time":0,                Block height or timestamp when transaction is final
  "size":226,                   Transaction size in bytes
  "in":[                        Incoming transactions
    {
      "prev_out":{              First incoming transactions output
        "hash":"26be31...",     The "hash" value of the previous transaction
```

```
        "n":0                          The index of the output
      },
      "scriptSig":"304502..."          what we have called scriptA, a proof of ownership
    }
  ],
  "out":[                              Outgoing transactions
    {                                  First transaction output
      "value":"12.99990000",           The value of bitcoins going out. (1,299,990,000 Satoshis)
      "scriptPubKey":"OP_DUP OP_HASH160 703b... OP_EQUALVERIFY OP_CHECKSIG"
                                       What we called scriptB, a verification of ownership
    },
    {                                  Second transaction output
      "value":"2.00000000",            Bitcoin value of the second output (200,000,000 Satoshis)
      "scriptPubKey":"OP_DUP OP_HASH160 5853... OP_EQUALVERIFY OP_CHECKSIG"
    }
  ]
}
```

Stripping of some boilerplate code: hashcode, version, size and number of incoming and outcoming transaction are not really interesting.

Lock Time

```
  "lock_time":0,                       Block height or timestamp when transaction is final
```

The lock time denotes the absolute time or block number when a transaction can enter the blockchain. For example, if the current block has the sequence number (block height) null, i.e. the Genesis Block[3], and `lock_time:10`, then the transaction can enter the blockchain only from block ten[4] on. Because we know that each block is mined roughly every 10 minutes, this results in a 100 minutes quarantine before the `lock_time`-ed transaction can be verified. Worse: currently (Dec 2013) the transaction can't be relayed in the p2p network at all.

The application for `lock_time` is to sign a transaction and give it to another party off-line, so that they can issue it after `lock_time` expires. For example: *if I don't return from my zombie hunt at block height 30,000, you can have my bitcoin wealth.* We will discuss more serious applications in the Writing Contracts chapter.

[3] The first block of the blockchain, https://blockchain.info/block/000000000019d6689c085ae165831e934ff763ae46a2a6c172b3f1b60a8ce26f?currency=MBC

[4] See block ten at: https://blockchain.info/block/000000002c05cc2e78923c34df87fd108b22221ac6076c18f3ade378a4d915e9

Transaction Inputs

```
1.    "in":[                                  Incoming transactions
2.       {                                    First transaction is included in {...}
3.          "prev_out":{                      First incoming transactions output
4.             "hash":"26be31...",            The “hash” value of the previous transaction
5.             "n":0                          The index of the output
6.          },
7.          "scriptSig":"304502..."           What we have called scriptA, a proof of ownership
8.       }                                    End of first transaction
9.    ],                                      End of input section
```

It is not enough though to refer the previous output transaction by the “hash” value. Since this transaction can have multiple outputs, we also need to tell what outputs we want to spend. This is done by the index `n`, that starts from 0 for the first output of the previous transaction refered by the `hash`.

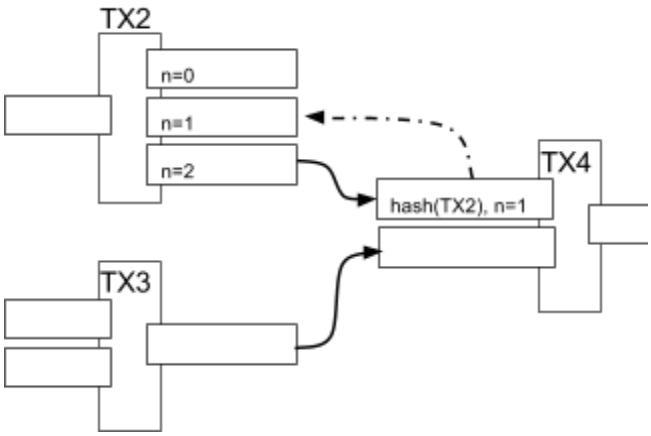


Figure: how the previous output is referred. The `hash` of the previous transaction identifies it, and the exact output is found by the index `n`.

Transaction Outputs

Transaction Outputs summed `values` must be smaller or equal the sum of the `values` found in inputs `prev_out`. Only then the transaction can be valid and spent. What happens with the difference if the output sum is smaller than the input? Its the miner fee, that incentivise the miners for verifying the transaction quicker than others.

```
1.   "out":[                                  Outgoing transactions
2.      {                                     First transaction output
3.         "value":"12.99990000",             The value of bitcoins going out.
4.         "scriptPubKey":"OP_DUP OP_HASH160 ..OP_EQUALVERIFY OP_CHECKSIG"
5.      },
6.      {                                     Second transaction output
7.         "value":"2.00000000",              Bitcoin value of the second output
8.         "scriptPubKey":"OP_DUP OP_HASH160 ..OP_EQUALVERIFY OP_CHECKSIG"
9.      }
10.  ]
```

Build your First Transaction

Appendix I - Useful Resources

Technical Introduction to Bitcoin youtube.com/watch?v=Lx9zgZCMqXE
How the bitcoin procol actually works
michaelnielsen.org/ddi/how-the-bitcoin-protocol-actually-works
A m-of-n transaction in the wild
http://blockexplorer.com/rawtx/60a20bd93aa49ab4b28d514ec10b06e1829ce6818ec06cd3aabd013ebcdc4bb1