(Bitcoin) Script Kiddies - Understanding Basic Transaction Scripts

Author	josh
Date	2019-02-16 17:34:35

Overview

In the Bitcoin world, money is not just digital - money is *programmable!* When transactions between users on the network are created and broadcast, miners and nodes independently verify that these transactions are valid. But this verification is not just checking some basic data points - it involves the execution of special scripts specified in the transaction parameters.

Script Basics

The Bitcoin Scripting Language

Before we can understand how basic Bitcoin scripts operate, we need to know a little bit about the scripting language itself. Unlike common scripting languages such as Python and Bash, the Bitcoin scripting language is quite limited and fairly simple in its execution. "Script" is stack based, meaning data is stored on an execution stack and script operators "push" and "pop" data from this stack. As well, Script is *not* Turing complete. There are no functions for looping or jumping around in the order of script execution. Operations are completely linear from the beginning of execution to the end. This keeps scripts secure, as it is not possible to tie up machines executing the scripts with an infinite loop.

Some operators are general, but most are specific to the cryptography of Bitcoin. Operators such as OP_ADD, OP_SUB, OP_DUP are pretty self explanatory - they make it possible to add, subtract, or duplicate data on the stack. Operators such as OP_HASH160 are more specific to the way Bitcoin operates - this operator takes the top item on the stack, hashes it using SHA-256 and then RIPEMD160, and finally pushes the result back on to the stack.

Common Bitcoin Script Mechanics

Pay to Public Key Hash (P2PKH) Script Basics

Finally, we need to understand a bit about how scripts are formed by discussing the basics of transactions. When a user "receives" Bitcoin in a transaction, they don't just have a "bank account" balance on the blockchain. Rather, the blockchain stores what are called *unspent transaction outputs* associated with a user's address. These outputs specify a *locking condition* that must be satisfied in a script when the user tries to spend that output in a future transaction. When the user creates a new transaction with that UTXO, they specify an unlocking script that satisfies that locking script

The most common form of script on the Bitcoin network is called *Pay to Public Key Hash*. With this type of script, the locking script requires that the user provide their public key and a digital signature formed with transaction data and their private key. This public key and digital signature will "satisfy" the locking script. When this unlocking script is combined with the UTXO locking script and executed, the final result on the Script stack should be *true*, meaning that the user can spend the Bitcoin.

P2PKH Formation

For a P2PKH script, the *locking script* specified by an unspent transaction output looks like this:

```
OP_DUP OP_HASH160 <Public Key Hash> OP_EQUALVERIFY OP_CHECKSIG
```

The unlocking script provides the user's signature and public key in order

```
<Signature> <Public Key>
```

In order to verify that the user owns the Bitcoin they wish to spend, a node verifying this transaction will append the *locking script* to the *unlocking script* and then execute it:

```
<Signature> <Public Key> >OP_DUP OP_HASH160 <Public Key Hash> OP_EQUALVERIFY
OP_CHECKSIG
```

Script Execution

Now let's walk through how this P2PKH script executes.

```
<Signature> <Public Key> OP_DUP OP_HASH160 <Public Key Hash> OP_EQUALVERIFY
OP_CHECKSIG
```

First, the signature and public key specified by the unlocking script are pushed on to the stack:

STACK: **<Signature>** **<Public Key>**

Next, OP_DUP pushes a copy of the top item on to the stack:

STACK: <**Signature>** **<Public Key>** **<Public Key>**

OP_HASH160 will pop the top stack item and hash it using SHA-256, and then RIPEMD160. Once the hashing operations are complete, the result is pushed on to the stack:

STACK: <**Signature>** **<Public Key>** **<Public Key Hash>**

The user's public key hash (a data item) specified by the locking script is pushed on to the stack:

STACK: <**Signature>** **<Public Key>** **<Public Key Hash>** **<Public Key Hash>**

OP_EQUALVERIFY now pops the top two stack items and checks that they are equal. If they are equal, execution continues. If the comparison fails, script execution exits with a failure.

STACK: <**Signature> <Public Key>**

OP_CHECKSIG now verifies that the signature is valid against the public key specified. An elliptic curve digital signature is created using a private key and a specific message, and any user with that message and public key can verify that the signature is valid without knowing the private key! Note that the message is not a part of the script, but is garnered from the overall transaction data. If the signature is valid, OP_CHECKSIG pushes true on to the stack.

STACK: **true**

Any Bitcoin script that ends with just *true* on the stack indicates a valid transaction. The user that created this transaction to spend some currency is in fact the rightful owner of the unspent output they want to use.

Bitcoin Scripts - Simple But Powerful

For someone with programming experience and some computer science background, Bitcoin scripts are generally straightforward to understand since the language is limited and Turing incomplete. Understanding P2PKH scripts requires just a working knowledge of stack data structures and commonly used cryptographic algorithms, but no higher level programming constructs. Now you know what goes on when you send your friend some money from your Bitcoin wallet!

However, the beauty of programmable money is the power to create transactions beyond the normal flow of "Alice sends Bob some cash". Script opens up the possibility of things like multi-signature transactions, time locked spending, and more!