# Bitcoin Transactions – Verification

## Overview

Before an output can be used as an input to another transaction, the spender must verify that they are the owner of the output. Each transaction contains a script which follows certain steps to check that the spender of the output is the rightful owner, and different types of scripts can follow different steps.

## Basics of Asymmetric Encryption

If two parties want to exchange data without it being visible to any third party, one way to do it is asymmetric encryption. Bitcoin’s verification system relies heavily on this kind of encryption, so a brief explanation will be provided here.

In any system using asymmetric encryption, every user owns 2 separate keys: a public key, and a private key. A user’s public key is visible to any other user, but their private key is only visible to themselves. Anything encrypted using the private key (i.e. “locked”) can be decrypted (“unlocked”) using their public key, and vice-versa.

If a user wants to send data (e.g. Sally) to a recipient (e.g. Robert), they can use the following steps. First, Sally encrypts the data using Robert’s public key. This means that no-one can read the data without decrypting it. Sally can then send the encrypted (i.e. hashed) data to Robert. Robert can use his private key to decrypt the data. This means that only Robert can decrypt it, as only Robert has access to the private key corresponding to his public key.

## Types of Scripts: P2PKH

P2PKH (Pay-To-Public-Key-Hash) scripts are the default script used for verifying bitcoin transactions. The steps for a P2PKH transaction are as follows:

First, the recipient of the transaction (Robert) generates a private key, and stores it. This is used to create a public key, and this public key is hashed (referred to as the pubkey hash). These steps are reproducible, so Robert doesn’t need to store the public key or pubkey hash.

Next, Robert gives a copy of the pubkey hash to the spender (Sally). Sally includes this in any outputs being given to Robert. The outputs will also contain a default pubkey script, which can verify the owner of the output when it is being spent.

With this, the initial transaction can continue as normal. The pubkey script is not used for verification until the output is actually spent. The transaction will be broadcast and added to the block chain (assuming it is valid).

In the transaction which spends the P2PKH output, Robert must verify that he owns the output. The pubkey script will return true or false, if false is returned then the second transaction will be invalidated. Robert must provide a signature and his public key to the pubkey script. The pubkey script executes the following steps:

* Robert’s signature is pushed to the execution stack
* Robert’s public key is pushed to the execution stack
* The public key is copied and pushed again to the stack
* The copy of the public key is hashed
* The original pubkey hash that Robert gave to Sally is then pushed to the stack
* The new pubkey hash is compared to the original. If they are different, the script returns false and evaluation halts
* Assuming they match, the signature is checked (decrypted with the now authenticated public key) – if it decrypts correctly and contains the necessary data, the transaction is valid

After this process (i.e. after the pubkey script is executed), if true is returned then the output is used in the second transaction as an input successfully (assuming that the rest of the transaction is valid).

## Types of Scripts: P2SH

P2PKH scripts have no conditions, but some recipients/spenders would want the script to only evaluate true for certain conditions (e.g. multiple people must provide signatures to allow the output to be spent). This can be done using P2SH (Pay-To-Script-Hash) verification, where instead of hashing the public key, the recipient hashes a redeem script. This script must be fulfilled to spend the output.

The steps for verifying a P2SH payment are the same as a P2PKH transaction, except instead of providing a public key when running the script, the recipient of the payment (Robert from the first example) must provide the redeem script (so that the two hashes match).

## Types of Scripts: Multisig

Multisig is a specialised type of verification script which allows the recipient to request that multiple correct signatures must be provided to validate a transaction. This is also possible using P2SH verification (and usually P2SH would be used for this), but the specialised script type still exists.

Multisig scripts are of the form “m-of-n”, which means that of the n provided signatures/hashes, at least m must match. m and n can be between 1 and 16. The script verification works the same way as a P2SH verification, but evaluation occurs for all “m” signatures provided, and the number verified is compared with “n”. If m<n, the verification fails.

## Types of Scripts: Pubkey

Pubkey verification is very similar to P2PKH, but it is less secure as pubkey hash comparison doesn’t occur, and so P2PKH is generally used instead.

## Types of Scripts: Null Data

Null data transactions are only used to add data to the block chain. They are unable to be spent as the pubkey script attached cannot be fulfilled (this must be provable for the transaction to be classified as a null data transaction). The “spender” of a null data transaction must also pay a transaction fee as usual.

## Locktime and Sequence Numbers

All transactions have a locktime. The locktime (included in a signature) indicates the earliest time that a transaction can be added to the block chain. This allows transactions to be created which are time-locked, allowing the signers time to change their mind. If the signers change their minds, they can create a new transaction which uses one of the inputs from the time-locked transaction, invalidating the time-locked transaction.

The locktime of a transaction is stored as a 4-byte unsigned integer. If it’s less than 500 million, it is read as block height (so the transaction can be added to any block with the height of the locktime or more). If it’s greater than 500 million, the locktime is interpreted as UNIX epoch time (i.e. the number or seconds passed since 00:00:00 on 01-01-1970.

Each transaction also contains four sequence numbers, which allow the locktime of a transaction to be disabled. This can be done if all four sequence numbers are set to their maximum value (0xffffffff).

## Standard Transactions

The default version of bitcoin code (Bitcoin Core) checks that all transactions are valid before they are broadcast by nodes in the network. This means that they must fulfil a set of rules, or else the transactions will be invalidated. Non-standard transactions can still be broadcast by nodes using different settings to the default Bitcoin Core settings, allowing them to be added to the block chain, but the chances of this occurring are low.

For a transaction to be standard, it must use one of the above types of verification (P2PKH, P2SH, Multisig, Pubkey, or Null Data). Standard transactions must also:

* Be finalised (i.e. the locktime has passed, or all of the sequence numbers are set to the maximum value)
* Be smaller than 100000 bytes
* Have signature scripts which are smaller than 1650 bytes (this allows multisig transactions of up to 15-of-15)
* Have less than 3 required public keys for a multisig transaction (non-P2SH)
* Have a signature script which only pushes data to the script evaluation stack
* Include only outputs of larger than 1/3 as many satoshis as it would take to spend in a typical input (currently 546 satoshis), except for null data outputs which must receive 0 satoshis

## Signature Hash Types

As mentioned above, certain transaction data is sent in a signature. This prevents modification of any signed data. The data which is included in the signature can be selected by the creator of the transaction. There are three types of signature hashes, which can each be modified with a flag: SIGHASH\_ANYONECANPAY.

SIGHASH\_ALL signs all of the inputs and outputs of a transaction, preventing anyone from modifying the transaction (by adding/changing any inputs or outputs) except the signer.

SIGHASH\_ALL|SIGHASH\_ANYONECANPAY signs only one input and all outputs, allowing anyone to add/remove other inputs as long as the total number of satoshis input is equal to those output.

SIGHASH\_NONE signs all of the inputs and none of the outputs, allowing anyone to change the outputs (as long as there are no inputs from other users which are signed with a different signature hash type).

SIGHASH\_NONE|SIGHASH\_ANYONECANPAY signs only one input but allows anyone to add/remove other inputs/outputs, allowing any user to spend the transaction in any way.

SIGHASH\_SINGLE signs only the output with the same index as this input. This means that no-one can modify the user’s part of the transaction, but other inputs/outputs can be added so long as the total number of satoshis input still equals the outputs (and transaction fees).

SIGHASH\_SINGLE|SIGHASH\_ANYONECANPAY signs one input and output, allowing any other users to add/remove other inputs.

## Key Reuse

If a user uses the same key multiple times, their transactions can be tracked (as the public key is publicly broadcast). There can also be other issues, such as the potential reconstruction of a user’s private key from their public key, or from comparison of multiple signatures.

Not reusing keys protects against any reconstruction of the private key. This is because the key would only be used for one transaction, and so anyone hoping to reconstruct the private key and modify said transaction would only be able to do so in the time between the reception of the transaction and the addition of the transaction to the block chain. This is a small time window, and so reconstruction of the private key in this period of time is effectively impossible under normal conditions. If a user reuses their key however, then an attacker has any time in between the first and last transaction made to reconstruct the private key, which is much more realistic.

Reusing keys also means that the user creates multiple signatures. These signatures can be compared to reconstruct the private key, allowing an attacker to modify transactions. If a key is only used once then only one signature is made, and so there are no other signatures to compare it with. This makes reconstruction of the private key through signature comparisons impossible (and also useless).

## Transaction Malleability

The signature script is unprotected in a transaction, and it can be modified. This can’t allow modification of a transaction, but it will affect the hash of the transaction. This hash is used to identify the transaction. This means that the actual transaction identifier (TXID) of a transaction will be different to the expected TXID.

This can cause problems. If a user wants to spend an output from a transaction which hasn’t been added to the block chain, they will use the expected TXID of the transaction to specify the output. Unexpected changes to the TXID due to transaction malleability will cause their new transaction to be invalid.

Transaction malleability can also cause issues with tracking payments. If a payment is tracked by its TXID, but the TXID changes, then it can appear to disappear. This can be avoided by tracking payments based on their inputs, not by their TXIDs. Payments can still be reissued if they seem to disappear, by using the same inputs in a new transaction (which would mean that one of the two payments are invalidated and one is valid, so one transaction should always be added to the block chain).