```
BIP: 79
Layer: Applications
Title: Bustapay :: a practical coinjoin protocol
Author: Ryan Havar <rhavar@protonmail.com>
Comments-Summary: No comments yet.
Comments-URI: https://github.com/bitcoin/bips/wiki/Comments:BIP-0079
Status: Replaced
Type: Informational
Created: 2018-10-05
License: CC0-1.0
Superseded-By: 78
```

Abstract

The way bitcoin transactions are normally created leaks more information than desirable, and as a result has been exploited by unreasonably effective blockchain analysis techniques to jeopardize important properties that are expected of a useful currency.

Bustapay is a simple and practical protocol for the sender and receiver of a payment to collaboratively sign a bitcoin transaction in such a way that busts some analysis assumptions to the immediate benefit of the sender and receiver. Furthermore it does so in such a way that gives a significant amount of control to the receiver to help manage their utxo set size, a constant problem for bitcoin merchants.

Copyright

Motivation

One of the most powerful blockchain analysis heuristics has been to assume all inputs of a transaction are controlled by a single party unless otherwise known (such as by the distinctive structure of a traditional coinjoin, or multisig spends that are validated onchain). Combined with other techniques (notably change-output guessing) this has lead to unexpectedly accurate tracking that has exposed bitcoin participants to unacceptable personal, business and financial risks -- undermining bitcoin's utility and fungibility -- and ultimately jeopardizing its ability to function as useful money.

We however can bust these assumptions with a sender-receiver coinjoin. To prevent costless spy/DoS attacks, we require the sending party to provide a fully-valid ready-to-propagate transaction to initiate the process, that the receiver can broadcast if the sender never completes the coinjoin thus tying the cost to that of spending a utxo. Most promisingly, bustapay transactions do not have an identifiable structure so any network analysis will be not able to tell if a given transaction is a bustapay transaction or not which erodes the confidence of their entire models, providing positive externalities for the entire bitcoin ecosystem.

Bustapay transactions also do not grow the receiver's count of unspent transaction outputs, and in fact gives the receiver an opportunity to better manage their utxo set, something normally only done when sending payments. Large utxo sets are often problematic and expensive, and frequently requiring privacy-destroying consolidation. Besides busting clustering assumptions, bustapay also provides a layer of obfuscation of send amounts.

It is worth noting that this specification has eschewed complexity and potentially useful extensions on the assumption that simplicity is of the most important to encourage adoption.

Overview

A bustapay payment is made from a sender to a receiver.

Step 1. Sender creates a bitcoin transaction paying the receiver This transaction must use segwit for all inputs, and be fully valid and signed. The transaction must be eligible for propagation on the network (but not done so at this stage)

Step 2. Sender gives the "template transaction" to the receiver This is done via an HTTP POST request, sent to a "bustapay url"

Step 3. Receiver processes the transaction and returns a partially signed coinjoin The receiver validates the transaction, and pays himself. The receiver then adds one or more of his own inputs (known as the *contributed inputs*) and (optionally) increases the output that pays himself (generally by the sum of the *contributed inputs*). Doing so creates a *partial transaction*, which the receiver returns to the sender. It is called such as it requires the sender to re-sign his own inputs.

Step 4. Sender validates, re-signs, and propagates on the bitcoin network The sender MUST validate the *partial transaction* was changed correctly and non-maliciously (to allow using potentially untrusted communication channels), re-sign its original inputs and propagate the final transaction over the bitcoin network.

Step 5. Receiver observes the finalized transaction on the bitcoin network Once the receiver has seen the finalized transactions on the network (and has enough confirmations) it can process it like a normal payment for the sent amount (as opposed to the amount that it looks like on the network). If the receiver does not see the finalized transaction after a timeout, they will propagate the original "template transaction", which ensures the payment happens and functions a strong anti-DoS mechanism.

Specification

The standard way of letting a sender know where to send a bustapay transaction is done via a bip21 encoded address. The key value "bpu" (short for "BustaPayUrl") should be used. An example of such address would be bitcoin:2NABbUr9yeRCp1oUCtVmgJF8HGRCo3ifpTT?bpu=https://bp.bustabit.com/submit It is highly encouraged that urls are kept short.

When the sender is creating a "template transaction" it is done almost identically to creating a normal send, with the exception that *only* segwit inputs may be used. The sender is also encouraged to use a slightly more aggressive feerate than usual as well as BIP125 (Opt-in Full Replace-by-Fee Signaling), but neither is strictly required.

The template transaction should be sent to the receiver via an HTTP POST to the bustapay url, with a binary encoded body.

The receiver is then responsible for validating the template transaction. If there is a problem with the transaction, or the receiver is generally unhappy with the transaction (e.g. fees are too small) the HTTP response code of 422 should be used and a human-readable string containing information on why which can be directly given to the user.

Should the receiver reject a transaction, it should not attempt to propagate it on the network. However it is important for the sender to be aware that the receiver *could* at any time (regardless of which error was given) send this transaction. The client should therefore assume the receiver will, and act accordingly (either retry with adjustments or just propagate the transaction). It is imperative that the sender never finds themselves in a situation where two payments to the sender could be valid.

Contributed Input Choice

The receiver must add at least one input to the transaction (the "contributed inputs"). If the receiver has no inputs, it should use a 500 internal server error, so the client can send the transaction as per normal (or try again later). Its generally advised to only add a single contributed input, however they are circumstances where adding more than a single input can be useful.

To prevent an attack where a receiver is continually sent variations of the same transaction to enumerate the receivers utxo set, it is essential that the receiver always returns the same contributed inputs when it's seen the same inputs.

It is strongly preferable that the receiver makes an effort to pick a contributed input of the same type as the other transaction inputs if possible.

Output Adjustment

After adding inputs to the transaction, the receiver generally will want to adjust the output that pays himself by increasing it by the sum of the contributed input amounts (minus any fees he wants to contribute). However the only strict requirement is that the receiver *must never* remove inputs, and *must not* ever decrease any output amount.

Returning the partial transaction

The receiver must sign all contributed inputs in the partial transaction. The partial transaction should also remove all witnesses from the the original template transaction as they are no longer valid, and need to be recalculated by the sender. The receiver returns the partial transaction as a binary-encoded HTTP response with a status code of 200. To ensure compatibility with web-wallets and browser-based-tools, all responses (including errors) must contain the HTTP header "Access-Control-Allow-Origin: *"

Sender Validation

The sender *must* do important validation on the partial transaction. They *must* verify:

- All template transaction inputs are in the partial transaction (but perhaps different order) and have the same sequence numbers.
- The partial transaction contains at least one new (and signed) segwit input (owned by the receiver)
- All outputs from the template transaction exist in the partial transaction, except they are allowed to be reordered and have their amounts increased (but *never* decreased)

Creating Final Transaction

After validating the partial transaction, the sender signs all its inputs to create what is now the final transaction. It is important that the sender is careful to not be tricked by the receiver into signing other inputs it owns. The sender must only sign inputs that existed in the template transaction. If the sender is not careful the receiver may "contribute" inputs that are actually owned with by the sender, with the hope the sender blindly signs everything.

Transaction Publishing

Once the final transaction is created, the sender should publish it directly onto the bitcoin network. If the sender does not do this after a reasonable time (e.g. 1 minute), the receiver should publish the template transaction as an important anti-spy/anti-DoS tactic . The sender may also choose to publish the template transaction instead of the final transaction if they believe the receiver to have unreasonably lowered the feerate of the transaction (i.e. increased the size of the transaction, but not the feerate enough). And both parties can consider publishing the template transaction even after the finalized transaction is on the network (taking advantage of replace-by-fee) if the final transaction is not confirming and the template transaction has more fees.

Implementation Notes

For anyone wanting to implement bustapay payments, here are some notes for receivers:

- A transaction can easily be checked if it's suitable for the mempool with testmempoolaccept in bitcoin core 0.17+
- Tracking transactions by txid is precarious. To keep your sanity make sure all inputs are segwit. But remember segwit does not prevent txid malleability unless you validate the transaction. So really make sure you're using testmempoolaccept at the very least
- Bustapay could be abused by a malicious party to query if you own a deposit address or not. So never accept a bustapay transaction that pays an already used deposit address
- You will need to keep a mapping of which utxos people have showed you and which you revealed. So if you see them again, you can reveal the same one of your own
- Check if the transaction was already sorted according to BIP69, if so ensure the result stays that way. Otherwise probably just shuffle the inputs/outputs
- A reference implementation is maintained at https://github.com/rhavar/bustapay which functions as a wrapper around some RPC calls to bitcoin core's wallet.
- The sender must be careful of an attack where the receiver tries to add additional inputs that are controlled by the sender, with the hope that the sender blindly signs it.

Backwards Compatibility

Bustapay is an optional payment protocol and therefore has no backwards compatibility concerns. It in fact can only be supported in addition to normal transaction processing, as falling back to a normal bitcoin transaction is a required behavior.

Credits

The idea is obviously based upon Dr. Maxwell's seminal CoinJoin proposal, and reduced scope inspired by a simplification of the "pay 2 endpoint" blog post by blockstream.