# Bitcoin Scripting Report

## Legacy (P2PKH) Transaction Details

### Legacy Addresses:

- Address A: moLCf35zVqdtFDtFgH6KwNGm5yokp399U5
- Address B: myho1AsNMWA1yuahCmDixJga8wrvDtNFoN
- Address C: mmZr5YUUi6NoCCfaAaLUpzwW84QwTBBk8Y

### Transaction IDs:

- A→B txid: f9a3b71dca1f48e5055cb0db0b2cec3a8ab176fbea88b5e41036f10310656939
- B→C txid: 9dc67446044ea7ac1b4178df73a6cd46a153bf44469e8d396096dd13c37c2135

## P2PKH Transaction Workflow

In a Pay-to-Public-Key-Hash (P2PKH) transaction workflow, the process involves multiple parties and transactions that build upon each other. This workflow demonstrates how Bitcoin transactions are linked through transaction identifiers (txids).

When A wants to send Bitcoin to B using P2PKH, the following process occurs:

1. B provides A with his public key hash, typically encoded as a Bitcoin address (starting with '1').
2. A creates a P2PKH transaction output containing instructions (scriptPubKey) that allow anyone to spend that output if they can

prove they control the private key corresponding to B's hashed public key.

3. A broadcasts this transaction to the Bitcoin network, and it gets added to the blockchain. The network categorizes it as an Unspent Transaction Output (UTXO).

4. This transaction has a unique identifier called a txid, which is created by double-hashing (SHA256 twice) the transaction data.

5. When B later wants to send these bitcoins to C, he must create a new transaction that references A's transaction by its txid and the specific output index number.

6. B creates a signature script (scriptSig) containing his signature and public key that satisfies the conditions A placed in the previous output's pubkey script.

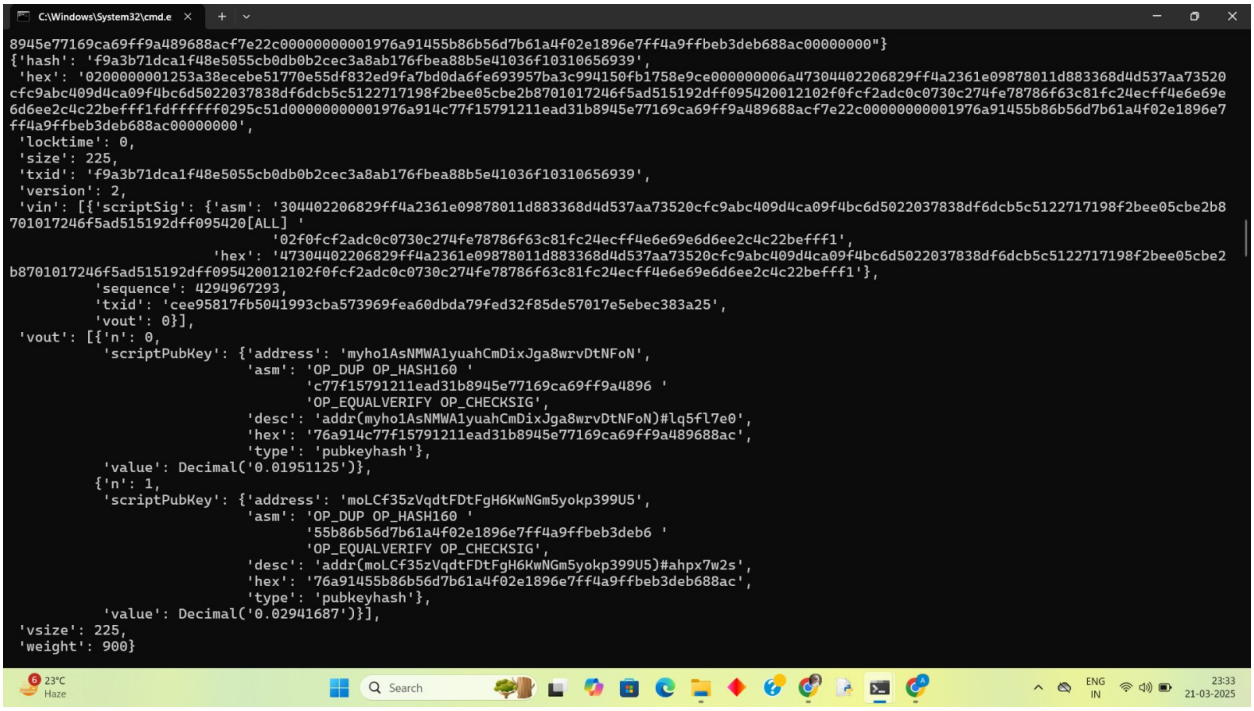
7. The validation process combines B's signature script with A's pubkey script to verify that B has the right to spend the output.

8. If validation succeeds, B's transaction to C is accepted by the network, creating a new UTXO that C can later spend using the same process.

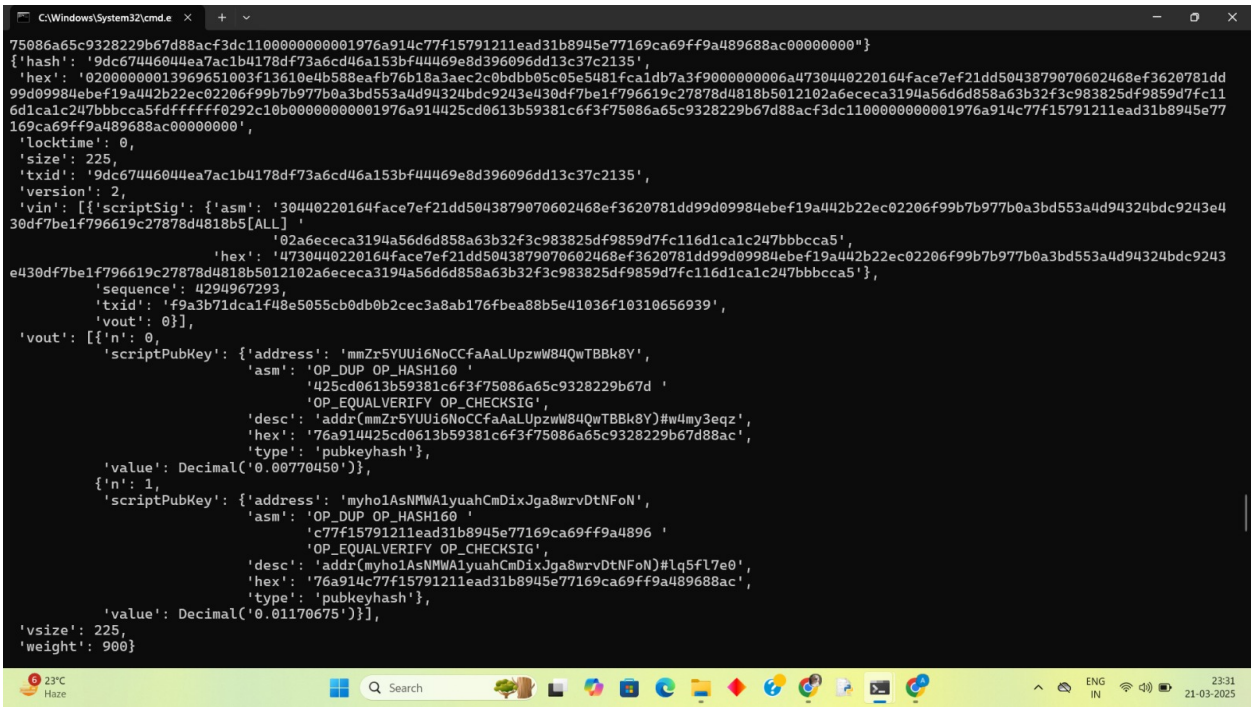
The txid serves as the critical link between these transactions, allowing each new transaction to reference and spend the outputs of previous transactions in a verifiable chain.

## P2PKH Script Structure and Validation

In Pay-to-Public-Key-Hash (P2PKH) transactions, the validation process relies on two key components: the challenge script (also called pubkey script) and the response script (also called signature script). These scripts work together to ensure that only the rightful owner of the bitcoins can spend them.




A->B




B->C

## Challenge Script Structure

The P2PKH challenge script has the following structure:

text

```
OP_DUP OP_HASH160 <PubkeyHash> OP_EQUALVERIFY
OP_CHECKSIG
```

This script contains:

- `OP_DUP`: Duplicates the top item on the stack
- `OP_HASH160`: Hashes the top item using HASH160 (SHA-256 followed by RIPEMD-160)
- `<PubkeyHash>`: The hash of the recipient's public key (essentially their Bitcoin address without Base58Check encoding)
- `OP_EQUALVERIFY`: Verifies that the two top items are equal and removes them
- `OP_CHECKSIG`: Verifies the signature against the public key

## Response Script Structure

When the owner wants to spend these bitcoins, they must provide a response script with:

text

```
<Sig> <PubKey>
```

This script contains:

- `<Sig>`: A digital signature created with the private key corresponding to the public key hash
- `<PubKey>`: The full public key that, when hashed, should match the pubkey hash in the challenge script

## Validation Process

The validation process combines both scripts and executes them in sequence:

text
```
<Sig> <PubKey> OP_DUP OP_HASH160 <PubkeyHash>
OP_EQUALVERIFY OP_CHECKSIG
```

The execution proceeds as follows:

1. `<Sig>` is pushed onto the initially empty stack
2. `<PubKey>` is pushed onto the stack
3. `OP_DUP` duplicates the public key on the stack
4. `OP_HASH160` hashes the duplicate public key
5. `<PubkeyHash>` (from the challenge script) is pushed onto the stack
6. `OP_EQUALVERIFY` checks if the calculated hash matches the stored hash
    - If they don't match, execution terminates and the transaction is invalid
    - If they match, both hashes are removed from the stack
7. `OP_CHECKSIG` verifies if the signature is valid for this transaction using the provided public key
    - If valid, it pushes TRUE onto the stack
    - If invalid, it pushes FALSE onto the stack

The transaction is considered valid if TRUE remains at the top of the stack after execution completes.

This validation ensures two critical security aspects:

1. The spender has the correct public key that matches the hash in the challenge script

2. The spender controls the private key corresponding to that public key, proven by providing a valid signature

The P2PKH script is designed to be stateless and non-Turing complete, ensuring that once a transaction is added to the blockchain, there's no condition that would render it permanently unspendable.

## SegWit Transaction Details

### SegWit Addresses:

- Address A': 2Mz4VTJroCth4D7Y2ei2mcZvYNAjEdnNHDU

- Address B': 2NBqEwthRj5SNCxXZn8VDr2QQCiCGG49iU1

- Address C': 2NBff7hcMk5DsMyVdM1nPYGFuQAawjVpoey

### Transaction IDs:

- A'→B' txid: 36bb8a1bf7fca6afdfbe3c4a47a97e8074042d6a83fa3837f5e5a94bc25c1da1

- B'→C' txid: fd157ecb073c8900e4f5ab95b0e75053a893ff3afbe28c384525e534115b9b30

## SegWit Transaction Workflow

In Segregated Witness (SegWit) transactions, the workflow and validation process differ from legacy P2PKH transactions, particularly in how transaction data is structured and how signatures are handled.

When A sends Bitcoin to B using SegWit, the following process occurs:

1. B provides A with his SegWit address (starting with "bc1" for native SegWit/Bech32 or "3" for P2SH-wrapped SegWit).

2. A creates a transaction sending Bitcoin to B's SegWit address. This transaction gets a unique transaction ID (txid) that is calculated by double-hashing (SHA256 twice) the transaction data with witness data removed.

3. The txid is created differently for SegWit transactions compared to legacy transactions. For SegWit, the transaction is first encoded without any witness data before hashing, ensuring backward compatibility with older nodes.

4. When B later wants to send these bitcoins to C, he creates a new transaction that references A's transaction by its txid and the specific output index.

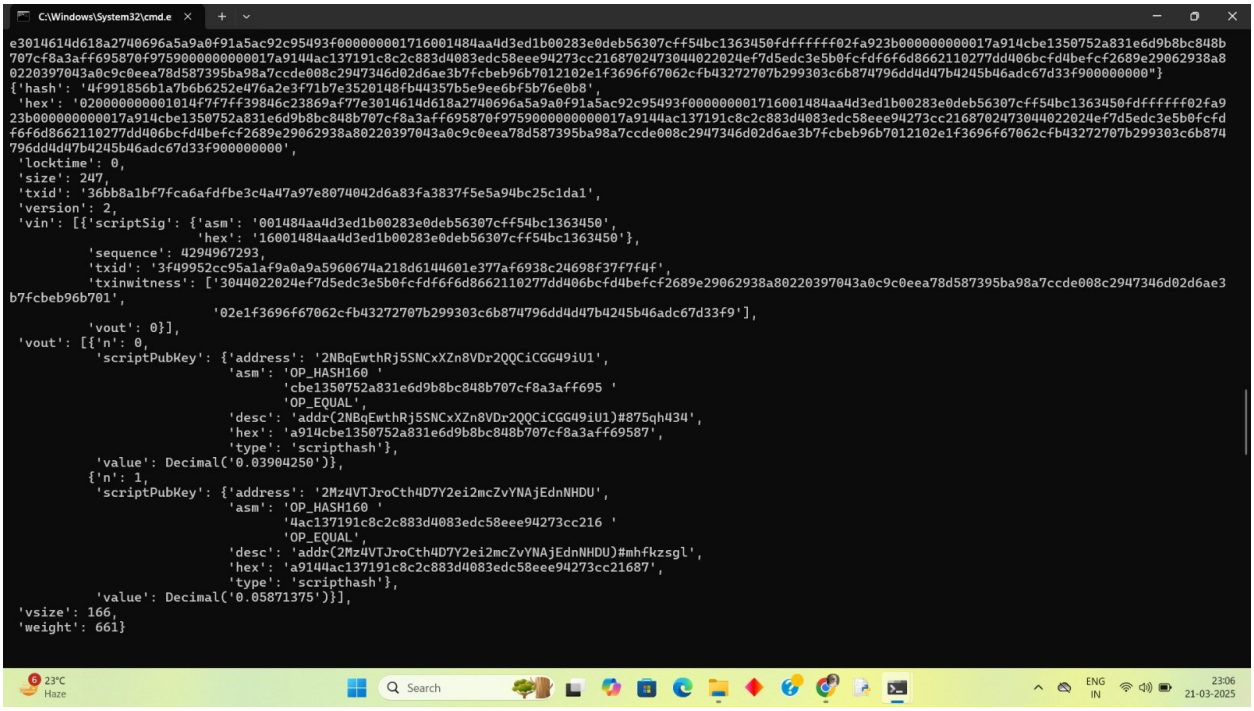
5. In this new transaction, B's input must include the previous txid (in reverse byte order) and the output index from A's transaction.

6. The witness data (signatures) is moved to a separate section at the end of the transaction rather than being included in the middle as with legacy transactions.

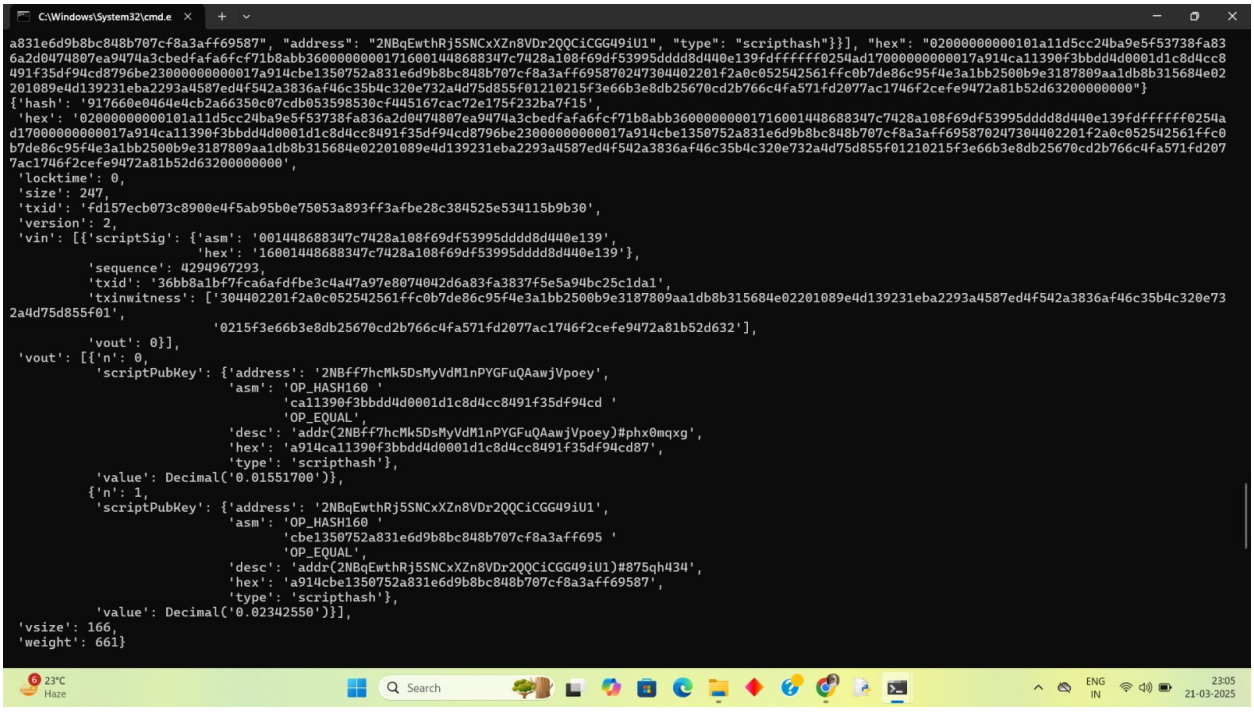
7. When B's transaction is validated and added to the blockchain, C can later spend it using the same process.

The key innovation with SegWit is that the witness data (signatures) is segregated from the transaction data used to calculate the txid, which resolves transaction malleability issues and allows for more transactions to fit in each block.

## SegWit Script Structure and Validation



Script for A->B



Script for B->C

# Challenge Script (scriptPubKey)

For native P2WPKH (Pay-to-Witness-Public-Key-Hash) SegWit transactions, the challenge script has the following structure:

text
```
OP_0 <20-byte-key-hash>
```

For P2SH-wrapped SegWit addresses (transitional format), the challenge script looks like:

text
```
OP_HASH160 <20-byte-script-hash> OP_EQUAL
```

Where the script hash is the hash of `OP_0 <20-byte-key-hash>`.

## Response Script (scriptSig and witness)

In SegWit transactions, the response is split between:

1. **scriptSig**: For native SegWit, this is empty. For P2SH-wrapped SegWit, it contains the redeem script (`OP_0 <20-byte-key-hash>`).

2. **witness**: Contains the actual signature and public key data needed to validate the transaction.

## Validation Process

The validation process for SegWit transactions works as follows:

1. For native SegWit (P2WPKH), when the transaction is validated:
    - The node identifies the input as SegWit by seeing the `OP_0 <20-byte-key-hash>` pattern
    - It then looks at the corresponding witness field that contains `<signature> <pubkey>`
    - The node verifies that the hash of the public key matches the key hash in the script

    - It then verifies the signature against the public key for this transaction

2. For P2SH-wrapped SegWit:
    - First, the P2SH validation occurs, checking that the hash of the redeem script matches the script hash
    - Then, the redeem script (`OP_0 <20-byte-key-hash>`) is executed, triggering the SegWit validation
    - The node looks at the witness data to complete the validation as with native SegWit
3. Legacy nodes that don't understand SegWit will:
    - For native SegWit: See a transaction with an unusual script they don't understand but accept as valid
    - For P2SH-wrapped SegWit: Process it as a normal P2SH transaction, stripping the witness data

The key advantage of this approach is that the witness data (which makes up about 60% of transaction data) is segregated from the main transaction data, allowing more transactions to fit in each block while maintaining backward compatibility with older nodes.

When requesting SegWit transaction data from other nodes on the network, you must specifically request the full transaction data including witness information using the appropriate message types (`MSG_WITNESS_TX` or `MSG_WITNESS_BLOCK`).

**Example ScriptSig from B→C transaction:**

```
30440220164face7ef21dd5043879070602468ef3620781dd99d099
84ebef19a442b22ec02206f99b7b977b0a3bd553a4d94324bdc9243
e430df7be1f796619c27878d4818b5[ALL]
02a6ececa3194a56d6d858a63b32f3c983825df9859d7fc116d1ca1
c247bbbcca5
```

Part 3

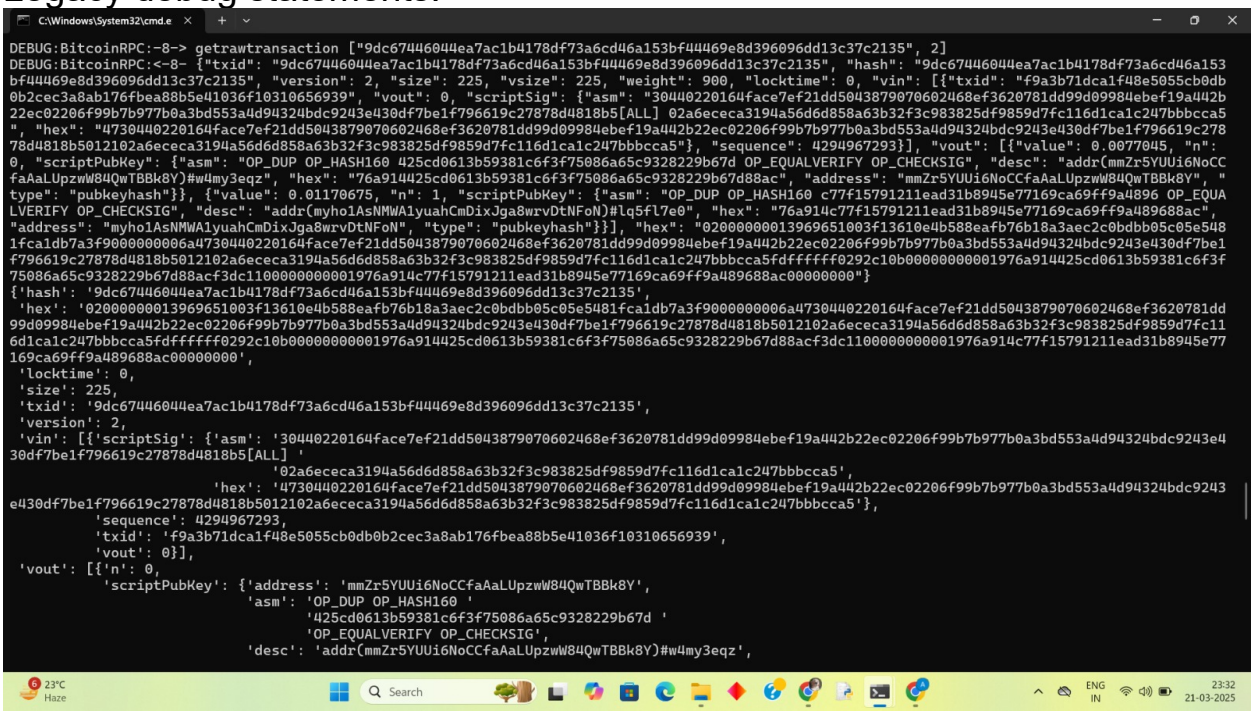
**P2PKH (Legacy)**

- The unlocking script (scriptSig) contains the **public key** and **signature**.
- The output script ensures that only the owner of the corresponding private key can spend the UTXO.
- Everything is contained within `scriptSig` and `scriptPubKey`.

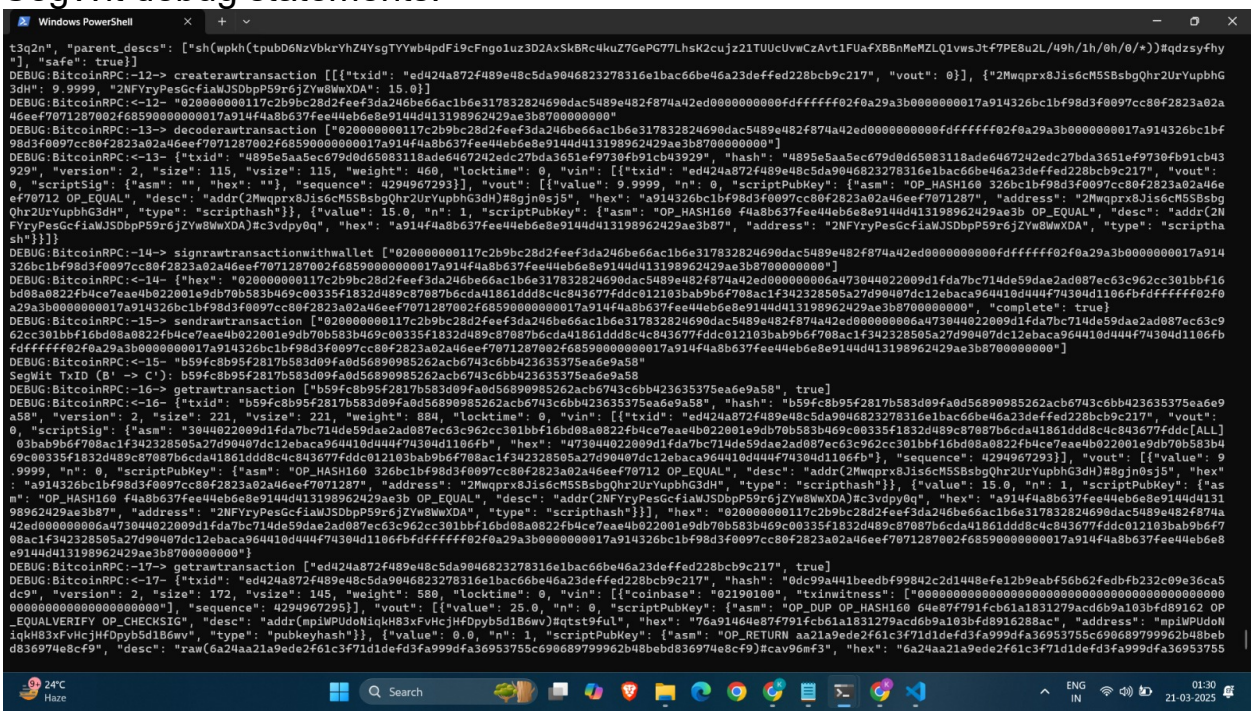
**P2SH-P2WPKH (SegWit)**

- The scriptSig is minimal, containing only the **redeemScript** (`0 <pubKeyHash>`).
- The **witness** stores the **signature** and **public key**, moving the unlocking logic out of `scriptSig` for efficiency.
- The scriptPubKey contains a hashed redeemScript, which is verified when spent.

## Legacy debug statements:




## SegWit debug statements:



## Why SegWit Transactions Are Smaller

SegWit transactions are smaller because they separate the transaction data from the witness data (which includes transaction signatures). In traditional Bitcoin transactions, both the transaction data (e.g., sender and receiver addresses) and the witness data (e.g., digital signatures) were stored together in the block. By moving the witness data outside of the main transaction block, SegWit reduces the size of each transaction, allowing more transactions to fit into a block of the same size .

## Benefits of Smaller SegWit Transactions

The smaller size of SegWit transactions offers several benefits:

1. Increased Block Capacity: With more transactions fitting into each block, the network's overall capacity to process transactions increases, improving scalability.
2. Reduced Transaction Fees: Smaller transactions require less space in a block, which means users can pay lower fees to have their transactions processed quickly. This is because the cost is typically calculated per byte of transaction data.
3. Improved Transaction Speed: While SegWit doesn't inherently speed up individual transaction confirmations, it increases the network's throughput. This means users often experience faster confirmations, especially during times of high network congestion.

4. Enhanced Security: SegWit solves the problem of transaction malleability by separating the signature data from the transaction data, making transactions more secure and less susceptible to malicious alterations.

Size for Legacy transaction is less than SeqWit transactions (screenshot):

```
 'txid': 'fd157ecb073c8900e4f5ab95b0e75053a893ff3afbe28c384525e534115b9b30',
 'version': 2,
 'vin': [{'scriptSig': {'asm': '001448688347c7428a108f69df53995dddd8d440e139',
                        'hex': '1600144868834 7c7428a108f69df53995dddd8d440e139'},
          'sequence': 4294967293,
          'txid': '36bb8a1bf7fca6afdfbe3c4a47a97e8074042d6a83fa3837f5e5a94bc25c1da1',
          'txinwitness': ['304402201f2a0c052542561ffc0b7de86c95f4e3a1bb2500b9e3187809aa1db8b315684e02201089e4d139231eba2293a4587ed4f542a3836af46c35b4c320e732a4d75d855f01',
                          '0215f3e66b3e8db25670cd2b766c4fa571fd2077ac1746f2cefe9472a81b52d632'],
          'vout': 0}],
 'vout': [{'n': 0,
           'scriptPubKey': {'address': '2NBff7hcMk5DsMyVdM1nPYGFuQAawjVpoey',
                            'asm': 'OP_HASH160 '
                                   'ca11390f3bbdd4d0001d1c8d4cc8491f35df94cd '
                                   'OP_EQUAL',
                            'desc': 'addr(2NBff7hcMk5DsMyVdM1nPYGFuQAawjVpoey)#phx0mqxg',
                            'hex': 'a914ca11390f3bbdd4d0001d1c8d4cc8491f35df94cd87',
                            'type': 'scripthash'},
           'value': Decimal('0.01551700')},
          {'n': 1,
           'scriptPubKey': {'address': '2NBqEwthRj5SNCxXZn8VDr2QQCiCGG49iU1',
                            'asm': 'OP_HASH160 '
                                   'cbe1350752a831e6d9b8bc848b707cf8a3aff695 '
                                   'OP_EQUAL',
                            'desc': 'addr(2NBqEwthRj5SNCxXZn8VDr2QQCiCGG49iU1)#875qh434',
                            'hex': 'a914cbe1350752a831e6d9b8bc848b707cf8a3aff69587',
                            'type': 'scripthash'},
           'value': Decimal('0.02342550')}],
 'vsize': 166,
 'weight': 661}
```


SegWit Size

```
 'txid': 'f9a3b71dca1f48e5055cb0db0b2cec3a8ab176fbea88b5e41036f10310656939',
 'version': 2,
 'vin': [{'scriptSig': {'asm': '304402206829ff4a2361e09878011d883368d4d537aa73520cfc9abc409d4ca09f4bc6d5022037838df6dcb5c5122717198f2bee05cbe2b8701017246f5ad515192dff095420[ALL] '
                               '02f0fcf2adc0c0730c274fe78786f63c81fc24ecff4e6e69e6d6ee2c4c22befff1',
                      'hex': '473044022068 29ff4a2361e09878011d883368d4d537aa73520cfc9abc409d4ca09f4bc6d5022037838df6dcb5c5122717198f2bee05cbe2b8701017246f5ad515192dff095420012102f0fcf2adc0c0730c274fe78786f63c81fc24ecff4e6e69e6d6ee2c4c22befff1'},
          'sequence': 4294967293,
          'txid': 'cee95817fb5041993cba573969fea60dbda79fed32f85de57017e5ebec383a25',
          'vout': 0}],
 'vout': [{'n': 0,
           'scriptPubKey': {'address': 'myho1AsNMWA1yuahCmDixJga8wrvDtNFoN',
                            'asm': 'OP_DUP OP_HASH160 '
                                   'c77f15791211ead31b8945e77169ca69ff9a4896 '
                                   'OP_EQUALVERIFY OP_CHECKSIG',
                            'desc': 'addr(myho1AsNMWA1yuahCmDixJga8wrvDtNFoN)#lq5fl7e0',
                            'hex': '76a914c77f15791211ead31b8945e77169ca69ff9a489688ac',
                            'type': 'pubkeyhash'},
           'value': Decimal('0.01951125')},
          {'n': 1,
           'scriptPubKey': {'address': 'moLCf35zVqdtFDtFgH6KwNGm5yokp399U5',
                            'asm': 'OP_DUP OP_HASH160 '
                                   '55b86b56d7b61a4f02e1896e7ff4a9ffbeb3deb6 '
                                   'OP_EQUALVERIFY OP_CHECKSIG',
                            'desc': 'addr(moLCf35zVqdtFDtFgH6KwNGm5yokp399U5)#ahpx7w2s',
                            'hex': '76a91455b86b56d7b61a4f02e1896e7ff4a9ffbeb3deb688ac',
                            'type': 'pubkeyhash'},
           'value': Decimal('0.02941687')}],
 'vsize': 225,
 'weight': 900}
```

Legacy Size

The vsize of a SegWit transaction is 166 vbytes.
The vsize of a Legacy transaction is 225 vbytes.