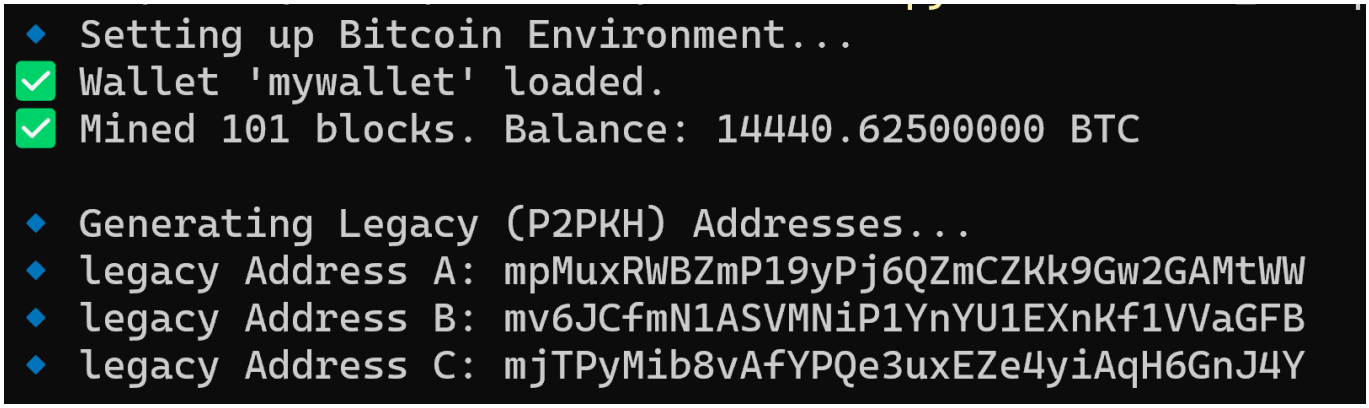
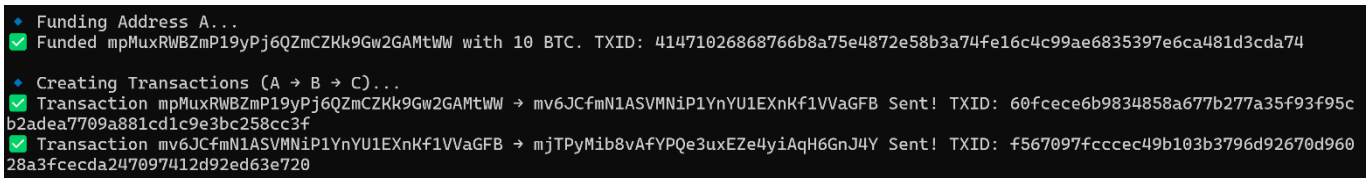
Report on Legacy (P2PKH) Transactions

1. Workflow of Transactions (A → B → C)





Step 1: Funding Address A

Before initiating the transactions, Address A (mpMuxRWBZmP19yPj6QZmCZKk9Gw2GAMtWW) was funded with **10 BTC**. The transaction that deposited these funds into Address A is identified by the following **TXID**:

TXID: 41471026868766b8a75e4872e58b3a74fe16c4c99ae6835397e6ca481d3cda74

This means Address A now has sufficient balance to transfer funds to another address.

Step 2: Transaction from Address A to Address B

The first transaction was created to transfer BTC from **Address A** (mpMuxRWBZmP19yPj6QZmCZKk9Gw2GAMtWW) to **Address B** (mv6JCfmN1ASVMNiP1YnYU1EXnKf1VVaGFB).

- The transaction ID (TXID) of this transfer:

TXID: 60fcece6b9834858a677b277a35f93f95cb2adea7709a881cd1c9e3bc258cc3f

- **Explanation**:
    - This transaction used the **10 BTC** in Address A as input.
    - The unlocking script (ScriptSig) in this transaction proves that Address A owned the UTXO and authorized the transfer.

This means that the new UTXO is locked to Address B, and Address B can spend the BTC by providing a valid unlocking script (ScriptSig) in a future transaction.

Step 3: Transaction from Address B to Address C

The second transaction was created to transfer BTC from **Address B** (mv6JCfmN1ASVMNiP1YnYU1EXnKf1VVaGFB) to **Address C** (mjTPyMib8vAfYPQe3uxEZe4yiAqH6GnJ4Y).

- The transaction ID (TXID) of this transfer:

TXID: f567097fcccec49b103b3796d92670d96028a3fcecda247097412d92ed63e720

- **Explanation**:
    - This transaction used the output of the previous transaction (60fcece6b9834858a677b277a35f93f95cb2adea7709a881cd1c9e3bc258cc3f) as its input.
    - The unlocking script (ScriptSig) in this transaction proves that Address B owned the UTXO and authorized the transfer.

This means that the new UTXO is locked to Address C, and Address C can spend the BTC by providing a valid unlocking script (ScriptSig) in a future transaction.

Step	*Sender Address*	*Receiver Address*	*TXID*
1	-	A (mpMuxRWBZmP19yPj6QZmCZKk9Gw2GAMtWW)	41471026868766b8a75e4872e58b3a74fe16c4c99ae6835397e6ca481d3cda74
2	A (mpMuxRWBZmP19yPj6QZmCZKk9Gw2GAMtWW)	B (mv6JCfmN1ASVMNiP1YnYU1EXnKf1VVaGFB)	60fcece6b9834858a677b277a35f93f95cb2adea7709a881cd1c9e3bc258cc3f
3	B (mv6JCfmN1ASVMNiP1YnYU1EXnKf1VVaGFB)	C (mjTPyMib8vAfYPQe3uxEZe4yiAqH6GnJ4Y)	f567097fcccec49b103b3796d92670d96028a3fcecda247097412d92ed63e720

2. Decoded Scripts for Legacy (P2PKH) Transactions

1. Transaction: Address A → Address B

```
🔹 Decoding Legacy Transactions...
🔎 Decoded Transaction 60fcece6b9834858a677b277a35f93f95cb2adea7709a881cd1c9e3bc258cc3f:
  - Locking Script (ScriptPubKey): OP_DUP OP_HASH160 9fe01af98ec070469c50e32969d856570ecc7e40 OP_EQUALVERIFY OP_CHECKSIG
  - Unlocking Script (ScriptSig): 3044022045ae8ca974c0bcc80caae1ab816cf25197a24c92272d6c9e693ad6265025a3f0022019295c23f8a6ec935291aa6f0b54c086b429363461efc315d764d8d56951f635[ALL] 0238fa8eec0f2b7ea2eafcb7ef3263282e42dbd9b1abfbe5888c19056ee63f3ee3
🔎 Decoded Transaction f567097fcccec49b103b3796d92670d96028a3fcecda247097412d92ed63e720:
  - Locking Script (ScriptPubKey): OP_DUP OP_HASH160 2b34461f489bd07473db3fa94bbcdbbf8b537dd7 OP_EQUALVERIFY OP_CHECKSIG
  - Unlocking Script (ScriptSig): 3044022045e9147b5d5de05b5c71bb86dc7e52cd1ae5814b250bca8ea2e3423ab46131c30220690db32bccda7c528a2c01663718fc8bc103cf0ae785991fcaaff1a16a8fa72c[ALL] 0311cbadc98500879f31279bebcd1db67ab75e364bd3d72713bb6f2b49f5297074
```

Transaction ID (TXID):

60fcece6b9834858a677b277a35f93f95cb2adea7709a881cd1c9e3bc258cc3f

Decoded Scripts

- **Locking Script (ScriptPubKey) of Output (B's address)**

OP_DUP OP_HASH160 9fe01af98ec070469c50e32969d856570ecc7e40 OP_EQUALVERIFY OP_CHECKSIG

Explanation:

- OP_DUP: Duplicates the top item on the stack (the public key).
- OP_HASH160: Hashes the public key using **SHA-256** followed by **RIPEMD-160**.
- 9fe01af98ec070469c50e32969d856570ecc7e40: The hashed public key of **Address B**.
- OP_EQUALVERIFY: Verifies that the provided public key matches the expected hash.
- OP_CHECKSIG: Checks the signature to validate the transaction.

This script ensures that only the owner of **Address B** can spend this UTXO by providing a valid signature and public key.

- **Unlocking Script (ScriptSig) of Input (A's address)**

```
3044022045ae8ca974c0bcc80caae1ab816cf25197a24c92272d6c9e693ad6265025a3f0022019295c23f8a6ec935291aa6f0b54c086b429363461efc315d764d8d56951f635[ALL] 0238fa8eec0f2b7ea2eafcb7ef3263282e42dbd9b1abfbe5888c19056ee63f3ee3
```

Explanation:

- The first part (3044...f635) is the **digital signature** of Address A's private key, signing the transaction.
- [ALL] is the SIGHASH flag, indicating the signature is for all inputs and outputs. This is the most common SIGHASH flag.
- The second part (0238fa8e...3ee3) is the **public key** corresponding to Address A.

This script proves that the sender (Address A) controls the private key needed to spend the UTXO.

2. Transaction: Address B → Address C

Transaction ID (TXID):

f567097fcccec49b103b3796d92670d96028a3fcecda247097412d92ed63e720

Decoded Scripts

- **Locking Script (ScriptPubKey) of Output (C's address)**

```
OP_DUP OP_HASH160 2b34461f489bd07473db3fa94bbcdbbf8b537dd7 OP_EQUALVERIFY OP_CHECKSIG
```

Explanation:

- OP_DUP: Duplicates the public key.
- OP_HASH160: Applies **SHA-256** then **RIPEMD-160** hashing.
- 2b34461f489bd07473db3fa94bbcdbbf8b537dd7: The hashed public key of **Address C**.

- OP_EQUALVERIFY: Ensures the public key matches the expected hash.
- OP_CHECKSIG: Verifies the signature to authorize the transaction.

This ensures that only the owner of **Address C** can spend this UTXO.

- **Unlocking Script (ScriptSig) of Input (B's address)**

```
3044022045e9147b5d5de05b5c71bb86dc7e52cd1ae5814b250bca8ea2e3423ab46131c30220690db3
2bccda7c528a2c01663718fc8bc103cf0ae785991fcaaff1a16a8fa72c[ALL]
0311cbadc98500879f31279bebcd1db67ab75e364bd3d72713bb6f2b49f5297074
```

Explanation:

- The first part (3044...a72c) is the **digital signature** of Address B's private key, signing the transaction.
- `[ALL]` is the SIGHASH flag, indicating the signature is for all inputs and outputs. This is the most common SIGHASH flag.
- The second part (0311cbad...97074) is the **public key** corresponding to Address B.

This proves that the sender (Address B) controls the private key needed to spend the UTXO.

3. Structure of Challenge and Response Scripts in Legacy (P2PKH) Transactions

1. Structure of Challenge Script (ScriptPubKey)

The **challenge script**, also called the **locking script (ScriptPubKey)** is included in the **output of a transaction**. It specifies the **conditions** required to unlock and spend the funds in a future transaction.

Structure of ScriptPubKey (Challenge Script)

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

Explanation:

Component	*Description*
OP_DUP	Duplicates the public key on the stack.
OP_HASH160	Hashes the duplicated public key using SHA-256 followed by RIPEMD-160.
<Public Key Hash>	The **hashed** public key of the recipient (the Bitcoin address).
OP_EQUALVERIFY	Ensures that the provided public key hash matches the expected hash.
OP_CHECKSIG	Verifies the signature using the provided public key.

How it Works

1. **Locks the UTXO**: The sender locks the transaction output so that only the recipient can unlock it.
2. **Requires a valid public key and signature**: The unlocking script (ScriptSig) must provide a valid **public key** and **digital signature** to match the challenge.

2. Structure of Response Script (ScriptSig)

The **response script**, also called the **unlocking script (ScriptSig)**, is included in **the input of a new transaction when spending the UTXO.**

Structure of ScriptSig (Response Script)

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

Explanation:

Component	*Description*
<Signature>	The **digital signature** generated by the sender's **private key**, signing the transaction.
<Public Key>	The **public key** corresponding to the private key that created the signature.

How it Works

- The spender provides their **public key** and **signature**.
- The **public key** proves ownership of the Bitcoin address.
- The **signature** is checked against the transaction data to verify authenticity.

3. Validation Process: How Bitcoin Executes the Scripts

Bitcoin uses a **stack-based execution model** to validate transactions.

Step-by-Step Execution:

1. Unlocking Script (ScriptSig) is pushed onto the stack

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

- First, the **digital signature** is pushed onto the stack.

  Stack: [Signature]

- Then, the **public key** is pushed onto the stack.

  Stack: [Signature, Public Key]

2. ScriptPubKey (Locking Script) is executed

OP_DUP OP_HASH160 <Public Key Hash> OP_EQUALVERIFY OP_CHECKSIG

- OP_DUP: Duplicates the **public key** on the stack.
  Stack: [Signature, Public Key, Public Key]
- OP_HASH160: Hashes the **top public key**.
  Stack: [Signature, Public Key, Hashed Public Key]
- <Public Key Hash>: Pushes the **expected public key hash** (stored in ScriptPubKey).
  Stack: [Signature, Public Key, Hashed Public Key, Expected Public Key Hash]
- OP_EQUALVERIFY: Compares the two hashes.
  - If they match, the **public key is verified**.
  - If not, the transaction is invalid.
    Stack: [Signature, Public Key]
- OP_CHECKSIG:
  - Uses the public key to check if the **signature is valid** for the transaction.
  - If valid, the transaction is **accepted**.
  - If invalid, the transaction is **rejected**.
    Final Stack: [] (Empty stack means successful validation).

Operation	*Stack Contents*
Push <Signature>	[Signature]
Push <Public Key>	[Signature, Public Key]
OP_DUP	[Signature, Public Key, Public Key]
OP_HASH160	[Signature, Public Key, Hashed Public Key]
Push <Public Key Hash>	[Signature, Public Key, Hashed Public Key, Expected Hash]
OP_EQUALVERIFY	[Signature, Public Key] (if match)
OP_CHECKSIG	[] (if signature is valid)

If the stack is empty after execution, the transaction is valid

4. Steps of Bitcoin Debugger Executing Challenge and Response Script

Debugging Steps:

Step 1: (Passing Script to BTC Debugger)

```
cse@cse-HP-ProDesk-400-G7-Microtower-PC:~/btcdeb$ btcdeb -v '[3044022045e9147b5d5de05b5c71bb86dc7e52cd1ae5814b250bca8ea2e3423ab46131c30220690db32bccda7c528a2c01663718fc8bc103cf0ae785991fcaaff1a16a8fa72c
 0311cbadc98500879f31279bebcd1db67ab75e364bd3d72713bb6f2b49f5297074 OP_DUP OP_HASH160 9fe01af98ec070469c50e32969d856570ecc7e40 OP_EQUALVERIFY OP_CHECKSIG]'
btcdeb 5.0.24 -- type `btcdeb -h` for start up options
LOG: signing segwit taproot
notice: btcdeb has gotten quieter; use --verbose if necessary (this message is temporary)
valid script
7 op script loaded. type `help` for usage information
script                                                              |  stack
--------------------------------------------------------------------+--------
3044022045e9147b5d5de05b5c71bb86dc7e52cd1ae5814b250bca8ea2e3423... |
0311cbadc98500879f31279bebcd1db67ab75e364bd3d72713bb6f2b49f5297074 |
OP_DUP                                                              |
OP_HASH160                                                          |
9fe01af98ec070469c50e32969d856570ecc7e40                            |
OP_EQUALVERIFY                                                      |
OP_CHECKSIG                                                         |
#0000 3044022045e9147b5d5de05b5c71bb86dc7e52cd1ae5814b250bca8ea2e3423ab46131c30220690db32bccda7c528a2c01663718fc8bc103cf0ae785991fcaaff1a16a8fa72c
```

Step 2: (ScriptSig Execution)

```
btcdeb> step
                <> PUSH stack 3044022045e9147b5d5de05b5c71bb86dc7e52cd1ae5814b250bca8ea2e3423ab46131c30220690db32bccda7c528a2c01663718fc8bc103cf0ae785991fcaaff1a16a8fa72c
script                                                              |                                                                stack
--------------------------------------------------------------------+----------------------------------------------------------------------
0311cbadc98500879f31279bebcd1db67ab75e364bd3d72713bb6f2b49f5297074 | 3044022045e9147b5d5de05b5c71bb86dc7e52cd1ae5814b250bca8ea2e3423...
OP_DUP                                                              |
OP_HASH160                                                          |
9fe01af98ec070469c50e32969d856570ecc7e40                            |
OP_EQUALVERIFY                                                      |
OP_CHECKSIG                                                         |
#0001 0311cbadc98500879f31279bebcd1db67ab75e364bd3d72713bb6f2b49f5297074
```

```
btcdeb> step
                <> PUSH stack 0311cbadc98500879f31279bebcd1db67ab75e364bd3d72713bb6f2b49f5297074
script                                                              |                                                                stack
--------------------------------------------------------------------+----------------------------------------------------------------------
OP_DUP                                                              | 0311cbadc98500879f31279bebcd1db67ab75e364bd3d72713bb6f2b49f5297074
OP_HASH160                                                          | 3044022045e9147b5d5de05b5c71bb86dc7e52cd1ae5814b250bca8ea2e3423...
9fe01af98ec070469c50e32969d856570ecc7e40                            |
OP_EQUALVERIFY                                                      |
OP_CHECKSIG                                                         |
#0002 OP_DUP
```

The stack shows:

```
[ <Signature> <PublicKey> ]
```

This is the first step where the **ScriptSig (Unlocking Script)** is pushed onto the stack.

- The **Signature** is added to the stack.
- The **Public Key** is added next.

Step 3: (OP_DUP Execution)

```
btcdeb> step
                <> PUSH stack 0311cbadc98500879f31279bebcd1db67ab75e364bd3d72713bb6f2b49f5297074
script                                                              |                                                                stack
--------------------------------------------------------------------+----------------------------------------------------------------------
OP_HASH160                                                          | 0311cbadc98500879f31279bebcd1db67ab75e364bd3d72713bb6f2b49f5297074
9fe01af98ec070469c50e32969d856570ecc7e40                            | 0311cbadc98500879f31279bebcd1db67ab75e364bd3d72713bb6f2b49f5297074
OP_EQUALVERIFY                                                      | 3044022045e9147b5d5de05b5c71bb86dc7e52cd1ae5814b250bca8ea2e3423...
OP_CHECKSIG                                                         |
#0003 OP_HASH160
```

- **OP_DUP** is applied to the top item (**PublicKey**).
- It pushes a copy of **PublicKey** into the stack.

🔗 Stack After OP_DUP:

```
[ <Signature> <PublicKey> <PublicKey> ]
```

Purpose: Prepares a copy of the public key for hashing while keeping the original intact for signature verification later.

Step 4: (OP_HASH160 Execution)

```
btcdeb> step
                <> POP  stack
                <> PUSH stack 9fe01af98ec070469c50e32969d856570ecc7e40
script                                                              |                                                                stack
--------------------------------------------------------------------+----------------------------------------------------------------------
9fe01af98ec070469c50e32969d856570ecc7e40                            |                                                9fe01af98ec070469c50e32969d856570ecc7e40
OP_EQUALVERIFY                                                      | 0311cbadc98500879f31279bebcd1db67ab75e364bd3d72713bb6f2b49f5297074
OP_CHECKSIG                                                         | 3044022045e9147b5d5de05b5c71bb86dc7e52cd1ae5814b250bca8ea2e3423...
#0004 9fe01af98ec070469c50e32969d856570ecc7e40
```

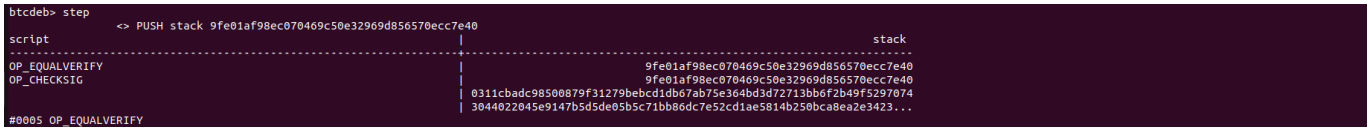
- **OP_HASH160** is applied to the top item (**PublicKey**).
- It performs **SHA256** followed by **RIPEMD160** hashing.

🔗 Stack After OP_HASH160:

[<Signature> <PublicKey> <PublicKeyHash>]

Where <PublicKeyHash> is the hashed version of the duplicated public key.

Step 5: (Pushing Expected PublicKeyHash - from ScriptPubKey)



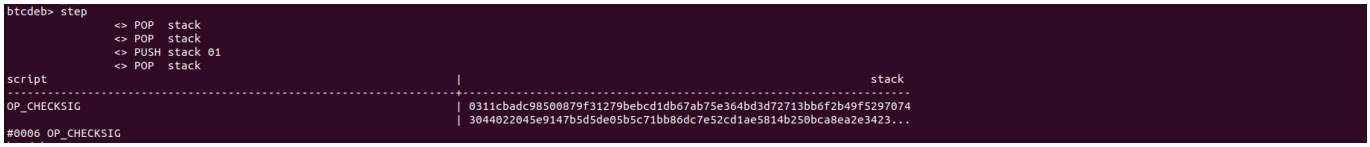
- The expected **Public Key Hash** (from the address locking the funds) is now pushed onto the stack.

🔗 Stack After Pushing Expected Public Hash:

[<Signature> <PublicKey> <PublicKeyHash> <ExpectedPubKeyHash>]

This sets the stage for comparison.

Step 6: (OP_EQUALVERIFY Execution)



OP_EQUALVERIFY compares the top two stack items:

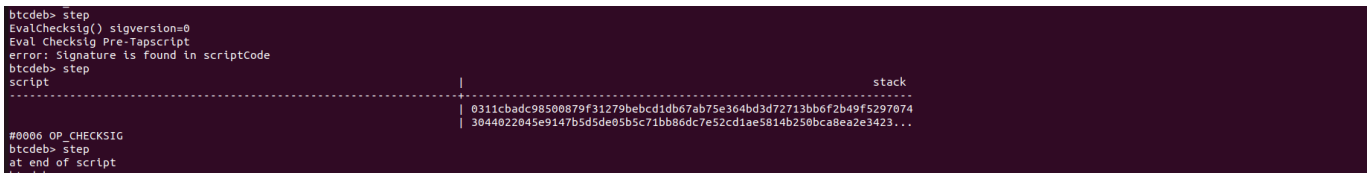
- <PublicKeyHash> (calculated)
- <ExpectedPublicKeyHash> (from ScriptPubKey)

- If equal: removes both and continues
- If not: validation **fails**
- Here, they **match** ✅, so:

🔗 Stack After OP_EQUALVERIFY:

[<Signature> <PublicKey>]

Step 7: (OP_CHECKSIG Success)



Explanation:

Now, **OP_CHECKSIG** is ready to execute:

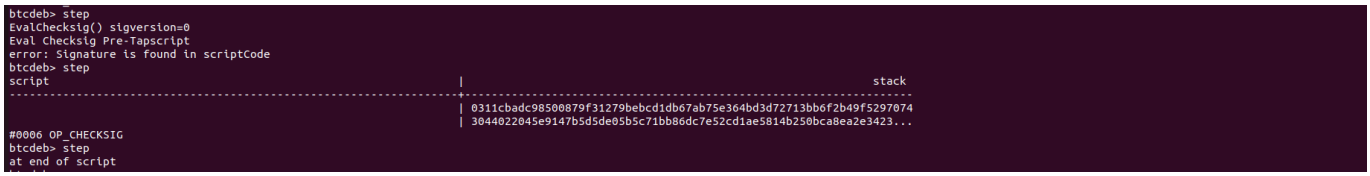
- It takes both the **Signature** and **PublicKey**

- Verifies that the signature is valid for the transaction **(using the provided PublicKey)**
- **OP_CHECKSIG** runs successfully
- If valid, it pushes 1 (True) onto the stack
- 0 would mean invalid signature
- Here, it **succeeds** ✅

🔗 Stack After OP_CHECKSIG:

[1]

Step 8: (Final Stack Cleared)



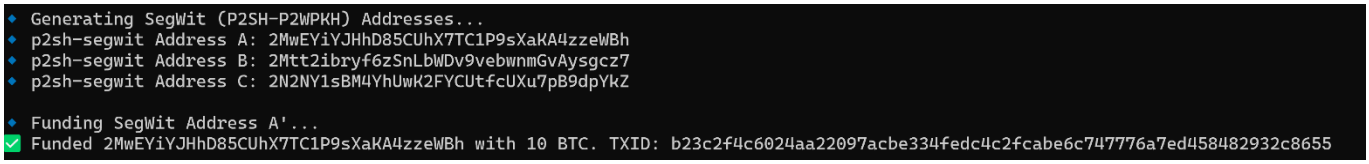
🔗 Stack After Validation:

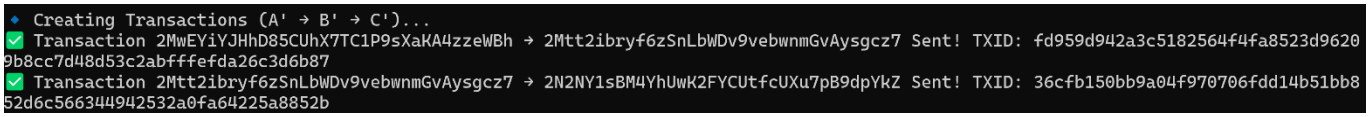
[ ]

- After evaluation, Bitcoin expects the **stack to be empty or have a true (1) value**.
- Typically, during final validation, 1 is consumed, leaving the stack **empty**.
- **Empty Stack = Transaction Validated**

Report on SegWit (P2SH-P2WPKH) Transactions

1. Workflow of Transactions (A → B → C)





Step 1: Funding Address A

Before initiating the transactions, Address A (2MwEYiYJHhD85CUhX7TC1P9sXaKA4zzeWBh) was funded with 10 BTC. The transaction that deposited these funds into Address A is identified by the following TXID:

TXID: b23c2f4c6024aa22097acbe334fedc4c2fcabe6c747776a7ed458482932c8655

This means Address A now has sufficient balance to transfer funds to another address.

Step 2: Transaction from Address A to Address B

The first transaction was created to transfer BTC from Address A (2MwEYiYJHhD85CUhX7TC1P9sXaKA4zzeWBh) to Address B (2Mtt2ibryf6zSnLbWDv9vebwnmGvAysgcz7).

- **Transaction ID (TXID) of this transfer:**

TXID: fd959d942a3c5182564f4fa8523d96209b8cc7d48d53c2abfffefda26c3d6b87

- **Explanation:**
    - This transaction used the 10 BTC in Address A as input.
    - The unlocking script (ScriptSig) in this transaction proves that Address A owned the UTXO and authorized the transfer.

This means that the new UTXO is locked to Address B, and Address B can spend the BTC by providing a valid unlocking script (ScriptSig) in a future transaction.

Step 3: Transaction from Address B to Address C

The second transaction was created to transfer BTC from Address B (2Mtt2ibryf6zSnLbWDv9vebwnmGvAysgcz7) to Address C (2N2NY1sBM4YhUwK2FYCUtfcUXu7pB9dpYkZ).

- **Transaction ID (TXID) of this transfer:**

TXID: 36cfb150bb9a04f970706fdd14b51bb852d6c566344942532a0fa64225a8852b

- **Explanation:**

- This transaction used the output of the previous transaction (fd959d942a3c5182564f4fa8523d96209b8cc7d48d53c2abfffefda26c3d6b87) as its input.
- The unlocking script (ScriptSig) in this transaction proves that Address B owned the UTXO and authorized the transfer.

This means that the new UTXO is locked to Address C, and Address C can spend the BTC by providing a valid unlocking script (ScriptSig) in a future transaction.

Step	*Sender Address*	*Receiver Address*	*TXID*
1	-	A (2MwEYiYJHhD85CUhX7TC1P9sXaKA4zzeWBh)	b23c2f4c6024aa22097acbe334fedc4c2fcabe6c747776a7ed458482932c8655
2	A (2MwEYiYJHhD85CUhX7TC1P9sXaKA4zzeWBh)	B (2Mtt2ibryf6zSnLbWDv9vebwnmGvAysgcz7)	fd959d942a3c5182564f4fa8523d96209b8cc7d48d53c2abfffefda26c3d6b87
3	B (2Mtt2ibryf6zSnLbWDv9vebwnmGvAysgcz7)	C (2N2NY1sBM4YhUwK2FYCUtfcUXu7pB9dpYkZ)	36cfb150bb9a04f970706fdd14b51bb852d6c566344942532a0fa64225a8852b

2. Decoded Scripts for SegWit (P2SH-P2WPKH) Transactions

1. Transaction: Address A → Address B

```
🔹 Decoding SegWit Transactions...
🔎 Decoded Transaction fd959d942a3c5182564f4fa8523d96209b8cc7d48d53c2abfffefda26c3d6b87:
  - Locking Script (ScriptPubKey): OP_HASH160 11ee1bbb8516976c82c18813be63af7a831f08ac OP_EQUAL
  - Unlocking Script (ScriptSig): 0014fa59dec58b0d53a56fb4967f783577c37dd98300
🔎 Decoded Transaction 36cfb150bb9a04f970706fdd14b51bb852d6c566344942532a0fa64225a8852b:
  - Locking Script (ScriptPubKey): OP_HASH160 641b179a559f20c27d6e6ed60c0319b8bc7c25c3 OP_EQUAL
  - Unlocking Script (ScriptSig): 0014125723f73943feddb54027c4dc20b79fed34a169
```

Transaction ID (TXID):

fd959d942a3c5182564f4fa8523d96209b8cc7d48d53c2abfffefda26c3d6b87

Decoded Scripts

🔹 **Locking Script (ScriptPubKey) of Output (B's address)**

```
OP_HASH160 11ee1bbb8516976c82c18813be63af7a831f08ac OP_EQUAL
```

Explanation:

- OP_HASH160: Hashes the redeem script using SHA-256 followed by RIPEMD-160.
- 11ee1bbb8516976c82c18813be63af7a831f08ac: This is the hashed redeem script corresponding to Address B.

- OP_EQUAL: Ensures that the provided redeem script, when spending, matches the expected hash.

- **Unlocking Script (ScriptSig) of Input (A's address)**

```
0014fa59dec58b0d53a56fb4967f783577c37dd98300
```

Explanation:

- 00: SegWit version 0 (indicating a native SegWit transaction).
- 14: Specifies that the following data is a 20-byte hash.
- fa59dec58b0d53a56fb4967f783577c37dd98300: This is the RIPEMD-160 hash of the sender's public key (Address A).

- **Witness Data (stored separately in SegWit structure):**
  - Signature: The ECDSA signature proving ownership of the private key.
  - Public Key: The uncompressed public key of the sender (Address A).

2. Transaction: Address B → Address C

Transaction ID (TXID):

36cfb150bb9a04f970706fdd14b51bb852d6c566344942532a0fa64225a8852b

Decoded Scripts

- **Locking Script (ScriptPubKey) of Output (C's address)**

```
OP_HASH160 641b179a559f20c27d6e6ed60c0319b8bc7c25c3 OP_EQUAL
```

Explanation:

- OP_HASH160: Hashes the redeem script using SHA-256 followed by RIPEMD-160.
- 641b179a559f20c27d6e6ed60c0319b8bc7c25c3: This is the hashed redeem script corresponding to Address C.
- OP_EQUAL: Ensures that the provided redeem script, when spending, matches the expected hash.

- **Unlocking Script (ScriptSig) of Input (B's address)**

```
0014125723f73943feddb54027c4dc20b79fed34a169
```

Explanation:

- 00: SegWit version 0 (indicating a native SegWit transaction).

- 14: Specifies that the following data is a 20-byte hash.
- 125723f73943feddb54027c4dc20b79fed34a169: This is the RIPEMD-160 hash of the sender's public key (Address B).

- **Witness Data (stored separately in SegWit structure):**
  - Signature: ECDSA signature proving ownership of the private key.
  - Public Key: The uncompressed public key of the sender (Address B).

3. Structure of Challenge and Response Scripts in SegWit (P2SH-P2WPKH) Transactions

1. Structure of Challenge Script (ScriptPubKey)

The locking script, or ScriptPubKey, specifies the conditions under which the UTXO can be spent. In SegWit P2SH-P2WPKH, the script is:

```
OP_HASH160 <Redeem Script Hash> OP_EQUAL
```

Explanation:

Component	*Description*
OP_HASH160	Hashes the provided Redeem Script using SHA-256 followed by RIPEMD-160.
<Redeem Script Hash>	A 20-byte hash of the Redeem Script.
OP_EQUAL	Ensures that the hash of the provided script matches the expected hash.

This script does **not** contain the actual Redeem Script but only its hash. This makes the transaction appear as a P2SH transaction while spending a SegWit output.

2. Structure of Response Script (ScriptSig)

The unlocking script (ScriptSig) in a SegWit P2SH-P2WPKH transaction does not contain a signature but only the Redeem Script:

```
<Serialized Redeem Script>
```

Explanation:

Component	*Description*
<Serialized Redeem Script>	This script matches the hash stored in ScriptPubKey and reveals the actual spending conditions.

For P2SH-P2WPKH, the Redeem Script is:

```
0 <Public Key Hash>
```

- **Unlike legacy transactions**, SegWit moves the actual unlocking information (signature + public key) to the **witness field**.
- The ScriptSig only needs to provide the Redeem Script so that the script hash in ScriptPubKey can be verified.

3. Witness Data (Actual Signature and Public Key)

The witness data contains:

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

Explanation:

Component	*Description*
<Signature>	A valid ECDSA signature proving ownership of the private key.
<Public Key>	The sender's full public key, which is hashed and checked against the Redeem Script.

This information is stored separately from the traditional ScriptSig, improving efficiency and security.

4. Validation Process

A Bitcoin node validates a SegWit transaction as follows:

Step 1: Check that ScriptSig contains the correct Redeem Script

- The node **extracts** the Redeem Script from ScriptSig:

```
0 <Public Key Hash>
```

- The node **hashes** this script using OP_HASH160.
- The node **compares** this hash with the <Redeem Script Hash> in ScriptPubKey.
- If they **match**, the script execution continues.

Step 2: Execute Witness Program

- The Redeem Script is structured as:

```
0 <Public Key Hash>
```

- The node recognizes that this is a **SegWit v0 script** (because of the leading 0).
- The node then verifies the witness stack.

Step 3: Verify Signature and Public Key

- The node extracts the <Signature> and <Public Key> from the witness field.
- The node hashes <Public Key> using SHA-256 and RIPEMD-160.
- The node compares the computed hash with <Public Key Hash> in the Redeem Script.
- If they **match**, the signature verification continues.

Step 4: Signature Verification

- The node uses the provided <Public Key> to verify the <Signature> against the transaction data.
- If the signature is valid, the transaction is considered **valid**.

4. Steps of Bitcoin Debugger Executing Challenge and Response Script

Debugging Steps:

Step 1: (Passing Script to BTC Debugger)

```
cse@cse-HP-ProDesk-400-G7-Microtower-PC:~/btcdeb$ btcdeb -v '[0014125723f73943feddb54027c4dc20b79fed34a169 OP_HASH160 11ee1bbb8516976c82c18813be63af7a831f08ac OP_EQUAL]'
btcdeb 5.0.24 -- type `btcdeb -h` for start up options
LOG: signing segwit taproot
notice: btcdeb has gotten quieter; use --verbose if necessary (this message is temporary)
valid script
4 op script loaded. type `help` for usage information
script                                      |  stack
--------------------------------------------+--------
0014125723f73943feddb54027c4dc20b79fed34a169 |
OP_HASH160                                  |
11ee1bbb8516976c82c18813be63af7a831f08ac     |
OP_EQUAL                                    |
#0000 0014125723f73943feddb54027c4dc20b79fed34a169
```

Step 2: (ScriptSig Execution)

```
btcdeb> step
                <> PUSH stack 0014125723f73943feddb54027c4dc20b79fed34a169
script                                      |                                        stack
--------------------------------------------+----------------------------------------
OP_HASH160                                  | 0014125723f73943feddb54027c4dc20b79fed34a169
11ee1bbb8516976c82c18813be63af7a831f08ac     |
OP_EQUAL                                    |
#0001 OP_HASH160
```

The ScriptSig contains a serialized Redeem Script:

<Serialized Redeem Script>

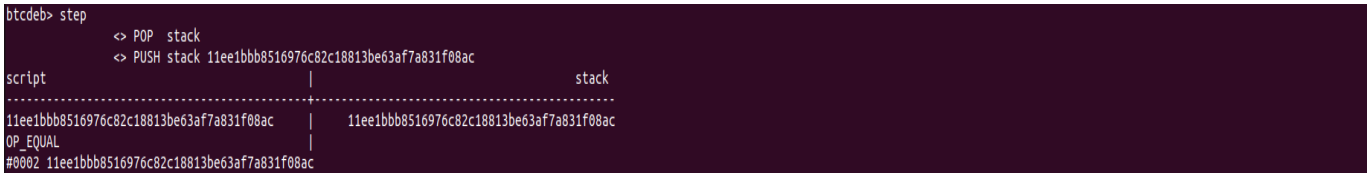
The Bitcoin node **extracts** the Redeem Script and decodes it.

The stack shows:

OP_0 <Public Key Hash>

This is the first step where the **ScriptSig (Unlocking Script)** is pushed onto the stack.

Step 3: (OP_HASH160 Execution)

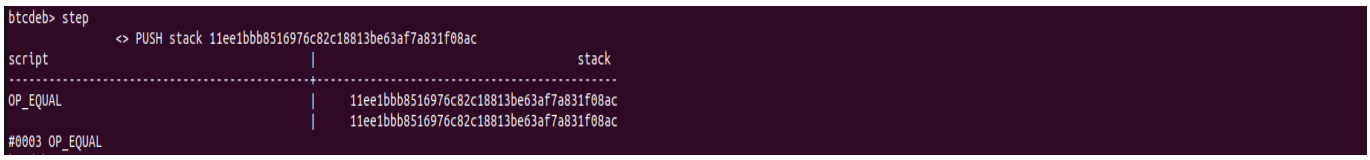


- **OP_HASH160** is applied to the top item (**Redeem Script**).
- It performs **SHA256** followed by **RIPEMD160** hashing.

Stack after OP_HASH160:

<Redeem Script Hash>

Step 4: (Pushing Expected Redeem Script Hash- From ScriptPubKey)

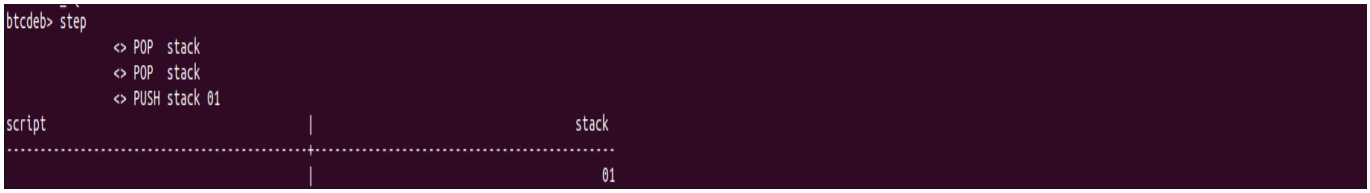


- The expected **Redeem Script Hash** is now pushed onto the stack.

Stack after Pushing Expected Redeem Script Hash:

<Redeem Script Hash> <Expected Redeem Script Hash>

Step 5: (OP_EQUAL Execution)



- The computed hash is then compared with the <Redeem Script Hash> stored in the ScriptPubKey.
- ✅ If they match, execution continues. Otherwise, validation fails.

Stack after OP_EQUAL:

[1]

5. Steps of Processing Witness Data

Step 1: Parse the Witness Data

The witness data contains:

- **<Signature>:** A valid ECDSA signature proving ownership of the private key.

- **<Public Key>:** The sender's full public key, which is hashed and checked against the Redeem Script.

This information is stored separately from the **ScriptSig**, improving efficiency and security.

📌 **Key Takeaway**: The witness separates critical transaction verification data, enhancing Bitcoin's security model

Step 2: Execute the Witness Program

The Redeem Script format:

```
0 <Public Key Hash>
```

- Since the leading 0 is present, the node **recognizes this as SegWit v0** (P2WPKH).
- The node now processes the witness stack.

Step 3: Validation Process of P2SH-SegWit Transaction

Step a: Validate the Redeem Script in ScriptSig

- The node extracts the Redeem Script from ScriptSig:

```
<Redeem Script>
```

- The node applies OP_HASH160 to the Redeem Script.
- The computed hash is compared with the **Redeem Script Hash** in scriptPubKey.
- If they match, execution continues.

📌 **Key Takeaway**: This step ensures the provided **Redeem Script** matches the expected hash.

Step b: Execute the Witness Program

- The Redeem Script structure:

```
0 <Public Key Hash>
```

- The node recognizes this as a **SegWit v0 script** due to the leading 0.
- The node then verifies the witness stack.

📌 **Key Takeaway**: This step confirms the SegWit script structure and prepares for witness verification.

Step c: Verify Signature and Public Key

- The node extracts:

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

- It hashes the **Public Key** using SHA-256 and RIPEMD-160.
- The computed **Public Key Hash** is compared to the one in the Redeem Script.

- If they match, signature verification proceeds.

📌 **Key Takeaway**: This step links the provided **Public Key** to the expected Redeem Script hash.

Step d: Signature Verification

- The node uses the **Public Key** to verify the **Signature** against transaction data.
- If valid, the transaction is confirmed.

📌 **Key Takeaway**: Signature verification is the final step ensuring only the rightful owner can spend the funds.

The node computes the **hash of the extracted Public Key**:

```
Hash = RIPEMD-160(SHA-256(<Public Key>))
```

This computed hash is compared with <Public Key Hash> in the Redeem Script.

Step 4: Signature Verification and Final Validation

- The node uses the extracted <Public Key> to verify the <Signature> against the **transaction data**.
- The **ECDSA algorithm** checks if the signature is valid for the given transaction.
  - ✅ If the signature is valid, the transaction is fully validated.
  - ❌ If invalid, the transaction is rejected.

Analysis and Explanation

1. Size of the P2PKH transactions and the P2SH-P2WPKH transactions

```
🔹 Comparing Transaction Sizes...
📊 Legacy Transaction Size: 225 vbytes
📊 SegWit Transaction Size: 166 vbytes
📊 Legacy Transaction Size: 225 vbytes
📊 SegWit Transaction Size: 166 vbytes
```

The comparison of transaction sizes reveals that **SegWit (P2SH-P2WPKH) transactions are significantly smaller than Legacy (P2PKH) transactions**.

- **Legacy P2PKH Transaction Size**: **225 vbytes**
- **SegWit P2SH-P2WPKH Transaction Size**: **166 vbytes** (≈ 26% smaller)

This reduction in size leads to **lower fees, better block efficiency, and improved security**. Bitcoin has widely adopted SegWit to enhance the network's scalability and prevent malleability attacks while ensuring compatibility with older systems.

2. Comparison of the script structures of P2PKH (Legacy) and P2SH-P2WPKH (SegWit) transactions in terms of challenge-response script and size of the script (weight, vbyte)

Aspect	*P2PKH (Legacy)*	*P2SH-P2WPKH (SegWit)*
Challenge Script (Locking Script / ScriptPubKey)	OP_DUP OP_HASH160 <Public Key Hash> OP_EQUALVERIFY OP_CHECKSIG	OP_HASH160 <Redeem Script Hash> OP_EQUAL
Response Script (Unlocking Script / ScriptSig)	<Signature> <Public Key>	<Redeem Script> , which is 0 <Public Key Hash>
Witness Data (New in SegWit)	Not present	<Signature> <Public Key> (Stored in Witness field)
Script Length	Longer, as all verification steps are in ScriptPubKey	Shorter, as the actual validation is moved to the witness
Weight (WU)	4 times the vbyte count (since all parts are in legacy format)	Lower, because witness data is only 1 WU per byte, while non-witness data is 4 WU per byte.
Virtual Size (vbyte)	Relatively high (directly proportional to script size)	Lower, due to SegWit witness discount
TXID Malleability	Affected by changes in ScriptSig	Not affected (signature moved to witness)
Efficiency	Less efficient due to larger scripts and no witness discount	More efficient as most data is in the witness section

Compatibility	Fully compatible with older Bitcoin versions	Compatible via P2SH but requires SegWit-supporting nodes for witness benefits

3. Why are SegWit Transactions Smaller and Benefits of SegWit Transactions

Why Are SegWit Transactions Smaller?

Segregated Witness (SegWit) is an upgrade to Bitcoin that improves transaction efficiency by separating the witness (signature) data from the main transaction structure. This change effectively reduces the size of Bitcoin transactions, making them more compact and allowing more transactions to fit within a block.

How SegWit Reduces Transaction Size

1. Separation of Witness Data

In traditional Bitcoin transactions, all data—including inputs, outputs, and signatures—is stored together within the same structure. This means that every transaction contains a significant amount of cryptographic signature data.

SegWit modifies this structure by **moving witness (signature) data to a separate section**. The transaction is now divided into two parts:

- **Main transaction data:** Contains inputs (without signatures), outputs, and other transaction details.
- **Witness data:** Holds the cryptographic signatures and other verification data.

Since signature data is no longer included in the main transaction, the effective size of the transaction decreases.

2. Weight-Based Calculation

Before SegWit, Bitcoin blocks had a strict **1 MB size limit**. This limited the number of transactions that could be included in a block, leading to congestion and high transaction fees.

With SegWit, a new metric called **weight units (WU)** was introduced, replacing the strict 1 MB block size limit with a **4 million weight unit limit**.

How weight is calculated:

- **Non-witness data (inputs & outputs)** → 1 byte = 4 weight units
- **Witness data (signatures)** → 1 byte = 1 weight unit (discounted)

This means that the witness data is given a **75% discount** when calculating block weight. As a result, the effective block size can exceed 1 MB while still adhering to the weight limit.

For example:

- A standard Bitcoin transaction might be **250 bytes** in size.
- If 100 bytes of this transaction are witness data, the weight calculation is:
    - **150 bytes (non-witness) × 4 WU/byte = 600 WU**
    - **100 bytes (witness) × 1 WU/byte = 100 WU**

- **Total weight = 700 WU (instead of 1000 WU if there were no discount)**

This makes the transaction **effectively smaller** in terms of how much space it takes up in a block.

3. Higher Transaction Density

Since witness data is heavily discounted, it takes up less block space. This allows more transactions to fit within the same 4 million WU limit, increasing Bitcoin's transaction throughput.

Benefits of SegWit Transactions

1. Lower Fees

Transaction fees in Bitcoin are based on the size of the transaction in bytes. Since SegWit transactions have a lower effective size due to witness separation and discounting, they incur **lower transaction fees** compared to legacy transactions.

2. More Transactions per Block

By reducing the effective size of each transaction, SegWit increases the number of transactions that can fit into a block. This helps:

- Reduce transaction backlog.
- Lower wait times for confirmation.
- Improve overall Bitcoin network efficiency.

3. Fixes Transaction Malleability

Transaction malleability refers to a flaw in traditional Bitcoin transactions where the signature data could be modified slightly without changing the transaction's actual effect. This could lead to issues with multi-step transactions and smart contracts.

SegWit eliminates this problem because:

- The transaction ID (TXID) is now based only on **the main transaction data** (not the witness).
- Since the witness data is separate, modifying the signature does not change the TXID.

This fix was **crucial for the Lightning Network**, which relies on unalterable TXIDs to enable instant micropayments.

4. Better Scalability

Bitcoin's scalability is improved because:

- More transactions fit into each block.
- Fees are lower, making small transactions more feasible.
- It lays the groundwork for future optimizations like **Taproot** and **Schnorr signatures**, which further improve efficiency.