Parse and Sign Bitcoin PSBTs with embit in Python

October 25, 2026 · Document #408 · bip39-phrase.com

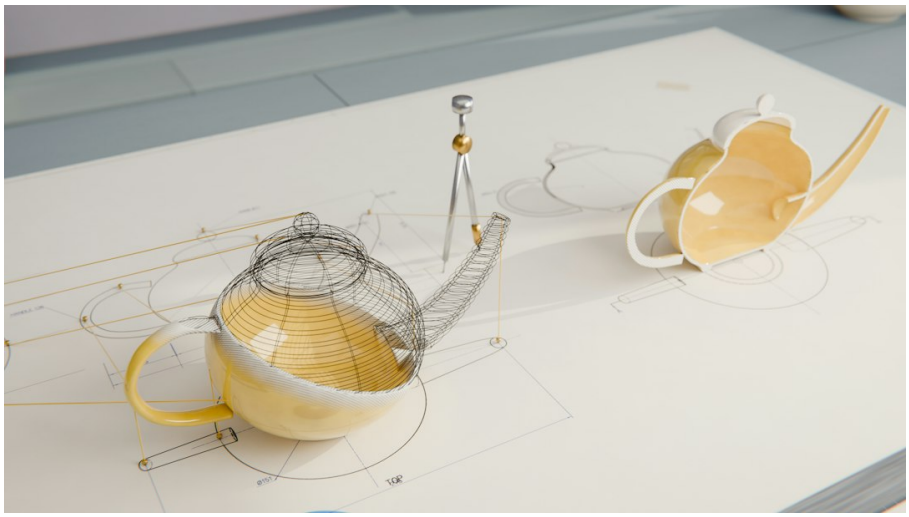


Photo by Brecht Corbeel on Unsplash

Load, inspect, and sign a base64 PSBT with embit, using public BIP32 test keys and descriptors while accounting for offline signing and validation limits.

Contents

Load the PSBT and inspect its contents	2
Derive a test key and check a descriptor	3
Review, sign, and serialize	3
Why embit fits an air-gapped signer	4
FAQ	4
Sources	4
Example BIP39 word sequences	5
Reading a BIP39 phrase	6
See also	6

A Python signer can read a base64 PSBT, inspect its transaction, and produce signatures without connecting to a Bitcoin node. The difficult part is deciding whether the transaction deserves a signature: a successfully parsed PSBT can still contain an unexpected payment, misleading change information, or incomplete previous-output data.

The example below uses embit's `psbt`, `bip32`, and `descriptor` modules with a public BIP32 test seed. It requires a matching test transaction supplied in `unsigned.psbt`; it does not create a spendable transaction by itself.

Load the PSBT and inspect its contents

Install embit with `python -m pip install embit`. For a reproducible signer, pin the reviewed package version and retain its dependency artifacts before moving the application offline.

A PSBT contains transaction data plus metadata needed by signers, such as previous transactions, spent outputs, scripts, and key derivation paths. BIP174 defines the original format, version 0, which embeds an unsigned transaction. BIP370 defines PSBT version 2, which distributes transaction fields among the PSBT maps instead. Confirm which versions your pinned library supports. Base64 is a transport encoding; decoding it provides no authentication.

The following Python statements load a base64 text file and list the transaction inputs and outputs. Execute them in order in a script; each backticked line is Python code.

```
from pathlib import Path

from embit import psbt, bip32, descriptor, networks

encoded = Path("unsigned.psbt").read_text(encoding="ascii").strip()

packet = psbt.PSBT.from_string(encoded)

tx = packet.tx

print("Inputs:", len(packet.inputs), "Outputs:", len(packet.outputs))

input_rows = [(i, vin.vout, vin.sequence, scope.utxo.value,
scope.utxo.script_pubkey.data.hex()) for i, (vin, scope) in enumerate(zip(tx.vin,
packet.inputs))]

output_rows = [(i, out.value, out.script_pubkey.data.hex()) for i, out in
enumerate(tx.vout)]

print("Inputs: index, previous-output index, sequence, satoshis, scriptPubKey hex")

print(*input_rows, sep="\n")

print("Outputs: index, satoshis, scriptPubKey hex")

print(*output_rows, sep="\n")

fee_sat = packet.fee()

print("Fee in satoshis:", fee_sat)
```

Each input scope exposes its available spent output through `utxo`. If the necessary previous-output information is missing, inspection or fee calculation can fail. Treat that failure as a reason to obtain a complete PSBT, rather than substituting a zero value.

The fee is the sum of input values minus the sum of output values. This calculation depends on the supplied input information. Validate previous-transaction hashes, referenced output indexes, and consistency between supplied UTXO records where applicable; arithmetic alone does not authenticate a witness UTXO value or establish that an output remains unspent. The final fee rate also depends on the finalized transaction’s virtual size.

Derive a test key and check a descriptor

Use the seed from BIP32 test vector 1. It is public and provides **no security**. Never fund keys derived from it, even if the code runs on an offline device.

These statements derive a BIP84 testnet account at `m/84h/1h/0h`, build its external-chain descriptor, and identify outputs matching its first receiving script:

```
seed = bytes.fromhex("000102030405060708090a0b0c0d0e0f")

root = bip32.HDKey.from_seed(seed)

account = root.derive("m/84h/1h/0h")

account_xpub =
account.to_public().to_base58(version=networks.NETWORKS["test"]["xpub"])

receive = descriptor.Descriptor.from_string(f"wpkh({account_xpub}/0/*)")

first_receive_script = receive.derive(0).script_pubkey()

matching_outputs = [i for i, out in enumerate(tx.vout) if out.script_pubkey.data ==
first_receive_script.data]

print("Outputs matching receiving index 0:", matching_outputs)
```

The descriptor specifies native SegWit single-key spending through `wpkh(...)`. The wildcard selects an address index, and `/0/` selects the external receiving branch. A conventional change branch uses `/1/`. The testnet extended-public-key encoding does not make the resulting scripts exclusive to testnet; keep this demonstration separate from real funds.

This comparison demonstrates descriptor-based script verification. It does not classify every wallet output. A real signer needs its approved account descriptors, relevant branches, and address indexes. Derivation metadata supplied by the coordinator is a lookup hint; derive the script locally and compare it with the actual output before labeling that output as change.

Review, sign, and serialize

Before signing, review the intended recipients, amounts, change scripts, absolute fee, and transaction policy. Display destinations in the intended network’s address format where possible, while retaining the underlying script comparison. Also enforce permitted sighash types: a signature’s commitment can differ from the complete transaction shown on screen. These inspection statements are not a complete transaction-policy validator.

For this example, the PSBT must spend an output controlled by the test root and include the scripts and key-origin information needed for embit to locate its key. Standard PSBT BIP32 derivation entries identify the root fingerprint and child path; those claims must be checked against locally derived keys and the spending script.

After approving the transaction, execute:

```
signature_count = packet.sign_with(root)
```

```
print("Signatures produced:", signature_count)

Path("signed.psbt").write_text(packet.to_string() + "\n", encoding="ascii")
```

The return value counts signing operations, not necessarily newly populated signature entries: an existing signature entry can be overwritten. A zero result means no signatures were produced by that call; investigate the keys, derivation paths, scripts, and permitted sighash types. Signing updates the PSBT. Finalization, extraction of the broadcast transaction, and broadcasting are separate operations. Another participant may still need to sign.

Why embit fits an air-gapped signer

Embit combines transaction handling, hierarchical keys, descriptors, and PSBT processing in a compact library designed for embedded use, including supported MicroPython environments. An offline project can assemble a focused signing application without including node synchronization or a network service.

A smaller application can reduce storage requirements and make dependency review more manageable. It still needs a trustworthy display, secure entropy for real wallets, protected key storage, and carefully bounded import handling. Transfer PSBTs through the project's chosen removable-media or QR workflow, and return only the necessary signing result. Air-gapping does not eliminate malicious-input or data-exfiltration risks.

Sources for this workflow are the embit project documentation and PSBT implementation, together with BIP174, BIP370, BIP32, and BIP84.

FAQ

Does the example need a BIP39 phrase?

No. It starts directly from an official BIP32 test seed. BIP39 would provide a separate mnemonic-and-optional-passphrase-to-seed step.

Why can fee calculation fail?

The PSBT may lack usable previous-output information. Obtain and validate the missing data before signing; a calculated fee alone does not prove that the supplied values are authentic.

Is an offline signer automatically safe?

No. It must verify transaction intent and handle untrusted PSBT data safely, even without a network connection.

Sources

- embit project documentation (github.com)
- embit PSBT implementation (github.com)
- BIP174: Partially Signed Bitcoin Transaction Format (github.com)

> Continue reading on bip39-phrase.com: BIP39 vs BIP32 vs BIP44: How the Seed Standards Fit Together
> More on this topic: BIP Standards

Example BIP39 word sequences

Randomly generated sequences of words from the BIP39 English wordlist. None of them passes the BIP39 checksum, so no wallet will accept them: they show the format only. Never store funds with a phrase you found in a document.

Example 1

1	guilt	2	person	3	iron	4	soccer	5	defense	6	rule
7	benefit	8	essay	9	kind	10	believe	11	jaguar	12	lyrics

Example 2

1	basket	2	wish	3	public	4	image	5	spirit	6	under
7	eager	8	garden	9	sound	10	flush	11	will	12	throw

Example 3

1	carry	2	void	3	unfold	4	current	5	relax	6	next
7	manage	8	inner	9	million	10	plate	11	dragon	12	total

Example 4

1	adult	2	custom	3	remember	4	exist	5	float	6	federal
7	verb	8	ivory	9	crater	10	sign	11	orchard	12	away

Example 5

1	twist	2	media	3	have	4	vacant	5	little	6	gather
7	blade	8	find	9	fluid	10	report	11	wait	12	energy

Example 6

1	alter	2	change	3	health	4	vibrant	5	surge	6	time
7	boil	8	coil	9	drill	10	welcome	11	patch	12	battle

Example 7

1	evoke	2	illegal	3	either	4	monster	5	butter	6	patch
7	oven	8	novel	9	boat	10	key	11	door	12	sorry

Example 8

1	limit	2	citizen	3	smooth	4	boss	5	fix	6	bounce
7	jump	8	youth	9	vacuum	10	sphere	11	advice	12	tunnel

Example 9

1	ride	2	blossom	3	earth	4	grape	5	attitude	6	talent
7	candy	8	narrow	9	once	10	moment	11	gap	12	across

Example 10

1	depth	2	grain	3	melt	4	profit	5	remove	6	tell
7	action	8	invite	9	harvest	10	captain	11	tool	12	season

Reading a BIP39 phrase

Every word stands for an 11-bit number: its position in the 2048-word list, counted from 0. Here is Example 1 decoded word by word. Each word links to its page on bip39-phrase.com, where the list is numbered from 1.

#	Word	Index (0-2047)	Word no.	11 bits
1	guilt	829	830	01100111101
2	person	1306	1307	10100011010
3	iron	946	947	01110110010
4	soccer	1646	1647	11001101110
5	defense	459	460	00111001011
6	rule	1513	1514	10111101001
7	benefit	169	170	00010101001
8	essay	617	618	01001101001
9	kind	980	981	01111010100
10	believe	165	166	00010100101
11	jaguar	953	954	01110111001
12	lyrics	1067	1068	10000101011

The 12 words carry 132 bits: 128 bits of entropy followed by a 4-bit checksum. The checksum must equal the first 4 bits of the SHA-256 hash of the entropy.

```
Entropy (hex): 67b469d966e3977a454a697a8295dcc2
```

This sequence ends in the checksum bits 1011, but SHA-256 of its entropy starts with 0000, so wallets reject it as an invalid phrase. That is the point of the checksum: about 15 of every 16 random or mistyped sequences fail it.

See also

- Air-Gapped Signing with PSBT and QR Codes · June 2, 2026
- Watch-Only Wallets: Tracking Funds Without Private Keys · October 18, 2026

Educational document. The example sequences are random and fail the BIP39 checksum: they are not wallets and must never hold funds. Security notes: bip39-phrase.com/security/