Taproot Key-Path Signing in bitcoinjs-lib

November 6, 2026 · Document #420 · bip39-phrase.com

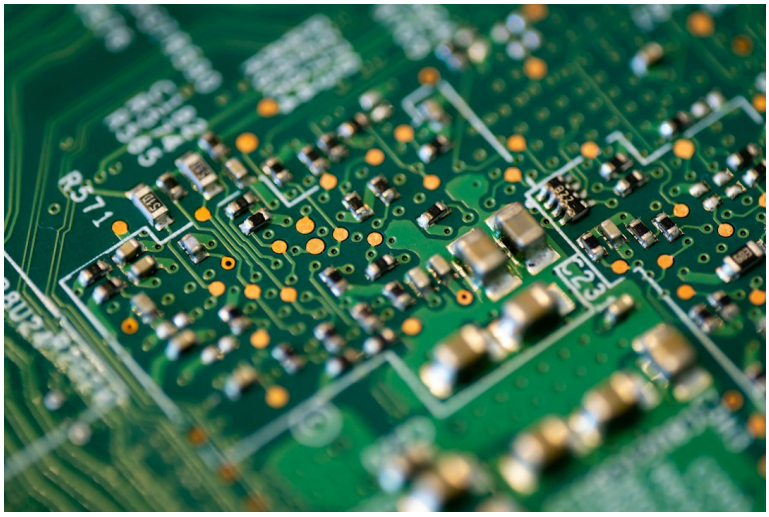


Photo by Magnus Engø on Unsplash

Build a BIP86 Taproot payment, normalize and tweak its signing key, configure the PSBT input, and verify the output against official vectors.

Contents

Initialize ECC and derive the internal key 1

Build the payment and tweak the signer 2

Add the Taproot input and sign 3

Confirm the BIP86 address 3

FAQ 3

Sources 4

Sample sequences of BIP39 words 4

Reading a BIP39 phrase 5

See also 5

A Taproot key-path signature must verify against the tweaked output key. Passing the original internal-key signer to a bitcoinjs-lib PSBT does not perform that tweak automatically. The payment, signing key, and PSBT metadata must describe the same Taproot output.

Initialize ECC and derive the internal key

Install compatible versions of `bitcoinjs-lib`, `tiny-secp256k1`, `ecpair`, and, for BIP86 derivation, `bip32` and `bip39`. Pin dependency versions and retain the lockfile. The examples below use bitcoinjs-lib's version 7 API, where transaction amounts are `bigint` values.

Initialize the elliptic-curve backend before constructing Taproot payments: `import * as bitcoin from 'bitcoinjs-lib'; import * as ecc from 'tiny-secp256k1'; import { ECPairFactory } from 'ecpair'; bitcoin.initEccLib(ecc); const ECPair = ECPairFactory(ecc);`. Initialization supplies the point operations bitcoinjs-lib needs to validate and tweak Taproot public keys.

For BIP32 derivation, initialize `BIP32Factory(ecc)` from the `bip32` package. BIP86 specifies the path `m / 86' / coin_type' / account' / change / address_index`. Its first mainnet receiving key in account zero uses `m/86'/0'/0'/0/0`.

Derive that child from the seed produced by the wallet's BIP39 mnemonic and passphrase, which may be empty. A compressed child public key has 33 bytes: a parity prefix followed by a 32-byte x-coordinate. Set `const internalPubkey = child.publicKey.slice(1, 33);`. This **x-only internal key** is the input to the Taproot payment constructor. Preserve the full compressed key because its prefix is needed when normalizing the signing secret.

Build the payment and tweak the signer

Construct the mainnet output with `const payment = bitcoin.payments.p2tr({ internalPubkey, network: bitcoin.networks.bitcoin });`. Check that `payment.address` and `payment.output` exist. The constructor calculates the tweaked output key and creates its witness-version-1 scriptPubKey and Bech32m address.

For a BIP86 output without a script tree, the tweak is `bitcoin.crypto.taggedHash('TapTweak', internalPubkey)`. The tagged-hash construction used by BIP341 is defined in BIP340: hash the tag with SHA-256, place that hash twice before the message, and apply SHA-256 again. Use the library function instead of substituting an ordinary SHA-256 hash.

The corresponding private-key calculation requires parity normalization:

1. Obtain the child's 32-byte private key as `let secret = child.privateKey;` and reject a missing key before continuing.
2. If `child.publicKey[0] === 3`, set `secret = ecc.privateNegate(secret);`. BIP340 interprets an x-only public key using the curve point with even y-coordinate.
3. Calculate `const tweak = bitcoin.crypto.taggedHash('TapTweak', internalPubkey);`.
4. Calculate `const tweakedSecret = ecc.privateAdd(secret, tweak);`. Stop if the library throws for an invalid tweak or returns null because the resulting secret is zero; do not silently substitute another key or reduce an invalid tweak yourself.
5. Create `const signer = ECPair.fromPrivateKey(tweakedSecret, { network: bitcoin.networks.bitcoin });`.

Check the result before signing: `Buffer.from(signer.publicKey.slice(1, 33)).equals(Buffer.from(payment.pubkey))` must return `true`, after checking that `payment.pubkey` exists. This compares the signer's x-coordinate with the payment's output key. The signer's Schnorr implementation handles the output key's parity when signing. Do not pass the tweaked key back as `internalPubkey`, which would apply another tweak.

Add the Taproot input and sign

Create `const psbt = new bitcoin.Psbt({ network: bitcoin.networks.bitcoin });`. Add the actual funding outpoint with `hash` set to its transaction ID and `index` set to its output index. Use verified transaction data for the referenced output, and confirm that its script equals `payment.output`.

The input for this workflow should include `witnessUtxo: { script: payment.output, value: fundingValue }` and `tapInternalKey: internalPubkey`. In the version 7 API, `fundingValue` is a `bigint` containing the output value in satoshis. The `tapInternalKey` field holds the original 32-byte internal public key; the tweaked output key belongs in the funding script. This metadata does not itself tweak the signer.

Add recipient and change outputs with their exact satoshi values. The fee is total input value minus total output value. With the default Taproot sighash, signing commits to every input's previous-output amount and script, so provide the required previous-output information for all inputs before signing.

Call `psbt.signInput(0, signer);`, then validate the signature using `psbt.validateSignaturesOfInput(0, (pubkey, hash, signature) => ecc.verifySchnorr(hash, pubkey, signature));`. Require successful validation before finalizing. If the transaction has additional inputs, sign and validate those too before calling `psbt.finalizeAllInputs();` and `psbt.extractTransaction();`.

With the default Taproot sighash, the key-path witness contains a 64-byte Schnorr signature. An explicitly selected non-default Taproot sighash adds a sighash byte. A script-free BIP86 spend needs neither a tapscript nor a control block.

Confirm the BIP86 address

Use the official BIP86 test mnemonic `abandon abandon abandon abandon abandon abandon abandon abandon abandon abandon abandon about` with an empty BIP39 passphrase. This public test phrase must never hold funds.

Derive `m/86'/0'/0'/0/0` and build the mainnet payment above. Copy the expected internal key, output key, scriptPubKey, and address directly from that entry in the official BIP86 vectors. Assert equality for all four, comparing byte values with the vectors' hexadecimal encodings and checking `payment.address === expectedAddress`. Checking only the address prefix cannot establish that the derivation or tweak is correct.

This vector tests key derivation and output construction. Signature validation and spending tests separately check the PSBT signing flow; signature validation alone does not establish that the funding output is unspent or that the transaction satisfies every consensus rule.

FAQ

Why negate an odd-parity private key?

The x-only internal key represents the even-y point. Negation aligns the private key with that point before adding the tweak.

Does tapInternalKey tweak the signer automatically?

No. In this bitcoinjs-lib workflow, supply a signer whose private key already includes the correct TapTweak adjustment.

Does this calculation support a script tree?

A script tree changes the tweak message to include its 32-byte Merkle root after the internal key. Payment construction and key-path signing must use the same root. The script-free calculation above must be adjusted accordingly.

Sources

- BIP86: Key Derivation for Single Key P2TR Outputs (github.com)
- BIP341: Taproot SegWit Version 1 Spending Rules (github.com)
- BIP340: Schnorr Signatures for secp256k1 (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

Sample sequences of BIP39 words

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 ice	2 wild	3 polar	4 december
5 garden	6 close	7 execute	8 fatal
9 raven	10 entry	11 robust	12 robust

Example 2			
1 aerobic	2 blossom	3 bunker	4 reveal
5 help	6 security	7 clever	8 often
9 taxi	10 west	11 cousin	12 fancy

Example 3			
1 mammal	2 swim	3 pill	4 pencil
5 possible	6 double	7 mammal	8 actor
9 pelican	10 large	11 rescue	12 royal

Example 4			
1 lucky	2 yellow	3 erode	4 sock
5 crawl	6 pear	7 minimum	8 conduct
9 transfer	10 glow	11 walk	12 tool

Example 5			
1 pulse	2 trial	3 element	4 knife
5 drift	6 term	7 bracket	8 claw
9 nut	10 sister	11 century	12 fatal

Example 6			
1 head	2 invite	3 journey	4 green
5 rally	6 scan	7 faith	8 leader
9 shy	10 wine	11 powder	12 wear

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	ice	897	898	01110000001
2	wild	2008	2009	11111011000
3	polar	1339	1340	10100111011
4	december	453	454	00111000101
5	garden	765	766	01011111101
6	close	349	350	00101011101
7	execute	633	634	01001111001
8	fatal	668	669	01010011100
9	raven	1427	1428	10110010011
10	entry	605	606	01001011101
11	robust	1498	1499	10111011010
12	robust	1498	1499	10111011010

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): 703f629d9c55fa5753ca9cb26976ed5d
```

This sequence ends in the checksum bits 1010, but SHA-256 of its entropy starts with 0110, 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

- Recovering Coins Across Multiple Address Types · October 22, 2026
- Air-Gapped Signing with PSBT and QR Codes · June 2, 2026
- Parse and Sign Bitcoin PSBTs with embit in Python · October 25, 2026

- Collaborative Custody: The Service Key Model · May 7, 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/