

Deriving a PGP Key from an Ed25519 Seed: A Vanilla Guide

Published under CC BY-SA 4.0 — fork it, remix it, host it.

Why this guide exists

Most PGP tutorials tell you to generate a new key and forget the old one. But if you already hold an Ed25519 seed — the kind used by NEAR, Solana, or any sovereign identity system — you may want that same root of trust to also produce a PGP key.

This is not a one-click operation. No online tool does it. There is no "paste your seed here" box that is safe to use. The only responsible way is to run a small script locally, on a machine you control, and never let the seed touch a network.

This guide documents exactly that process. It is the same path I walked to produce a working, importable OpenPGP key from a raw hex seed.

Prerequisites

- A Linux, macOS, or WSL environment (a VPS or local machine both work)
- Node.js v18 or newer
- A 64-character hex Ed25519 seed (32 bytes)
- Roughly five minutes

Check your Node version:

```
node -v
```

If you see v18.x or higher, you are ready.

Part 1: Understand what you actually have

An Ed25519 seed is a 32-byte value generated from Sovereign Burner. In hex, that is 64 characters. Example shape:

```
fd642221bb681a3ef6728eaa378f198db8ced24f4797c4d3a185bdfa9e1ea527
```

That value can:

- Derive an Ed25519 key pair
- Sign messages and transactions
- Authenticate to chains like NEAR or Solana

It cannot, on its own, become a PGP key. OpenPGP requires more than raw key material. A valid PGP key needs:

- A User ID (name, optional comment, email)
- Key usage flags (sign, certify, encrypt)
- Self-signatures binding the key to the identity
- Optionally, subkeys for encryption

So the process is not "convert seed to PGP." It is "use the seed as the signing key, then wrap it in a complete OpenPGP structure." That wrapping is what the tooling below handles.

Part 2: Set up a working directory

Keep this work isolated. Do not run it inside a shared project folder.

```
mkdir cryptography
cd cryptography
npm init -y
```

Then install the maintained library:

```
npm install micro-key-producer
```

A note on libraries: the older ed25519-keygen package still works but is deprecated by its author in favor of micro-key-producer. The successor also produces keys with better compatibility in standard OpenPGP software. Use the maintained one.

Part 3: Write the derivation script

Create a file named derive.mjs:

```
nano derive.mjs
```

Paste this in full:

```

import fs from 'fs';
import { getKeys } from 'micro-key-producer/pgp.js';

// 1. Paste your 64-character hex seed below.
//    Do not include "0x" or any spaces.
const hexSeed = 'PASTE_YOUR_64_CHAR_HEX_SEED_HERE';

// 2. Convert the hex string to a 32-byte Buffer.
const seed = Buffer.from(hexSeed, 'hex');

// 3. Fail loudly if the seed is the wrong size.
if (seed.length !== 32) {
  console.error(`\nError: expected 32 bytes, got ${seed.length}.`);
  console.error('The hex string must be exactly 64 characters long.\n');
  process.exit(1);
}

// 4. Your PGP identity. Format: Name (Comment) <email>
//    The email here is permanently bound to this key.
const userId = 'Juno Sterling (363d8379-f4f1-4696-ae8f-2b51956a1cf7)
<you@example.com>';

// 5. Derive the OpenPGP key pair.
const keys = getKeys(seed, userId);

// 6. Write armored output to disk.
fs.writeFileSync('private.asc', keys.privateKey);
fs.writeFileSync('public.asc', keys.publicKey);

// 7. Confirm.
console.log('\nKeys written:');
console.log('  private.asc – import this into your PGP client');
console.log('  public.asc   – share this with others');
console.log('\nDelete the private key file and this script when
finished.\n');

```

Save with Ctrl+O, Enter, then exit with Ctrl+X.

Why .mjs?

The .mjs extension tells Node to treat the file as an ES module, which lets us use import and top-level await without extra configuration.

Why Buffer.from(hexSeed, 'hex') and not a manual parse?

Buffer's hex decoder is native, strict, and length-checked. Manual approaches using parseInt on split strings will silently produce NaN for malformed input, and you will end up with a one-byte array and a confusing error from deep inside the crypto library. Let the runtime do the parsing.

Part 4: Run it

```
node derive.mjs
```

On success you will see:

```
Keys written:
  private.asc  - import this into your PGP client
  public.asc   - share this with others
```

If instead you see an error about byte length, your hex string is the wrong size. Count the characters. It must be 64. A common mistake is pasting a Base58 string, a public key, or a truncated copy.

Part 5: Import into your PGP client

OpenKeychain (Android)

1. Copy private.asc to your phone.
2. Open OpenKeychain → Import key from file.
3. Select private.asc.

If your client shows an alert like "PGP data decoding failed," you are most likely using a key generated by the deprecated ed25519-keygen library. Regenerate with micro-key-producer as shown above. Its output conforms more closely to what strict parsers expect.

Also verify the reported key size. A correct Ed25519 key reports as 256 bits. If you see 264 bits, the key material was assembled incorrectly and you should regenerate.

GnuPG (desktop)

```
gpg --import private.asc  
gpg --import public.asc  
gpg --list-secret-keys --keyid-format LONG
```

Thunderbird, ProtonMail, or any other client

Use the same private.asc file. The format is standard ASCII-armored OpenPGP.

Part 6: Clean up immediately

You have now written your seed-derived private key to disk in plaintext. That is necessary for the import, but it must not persist.

After the import succeeds:

```
rm private.asc derive.mjs
```

Verify the client holds the key. If it does, the file on disk is redundant and only serves as a liability.

If you needed to keep the public key for sharing:

```
mv public.asc ~/public-key-newidentity.asc
```

Troubleshooting

"Expected 32 bytes, got N"

Your hex string is not 64 characters. Recopy it. Check for a stray newline or a 0x prefix.

"PGP data decoding failed" in OpenKeychain

You used the deprecated library, or a malformed User ID. Regenerate with micro-key-producer. Ensure the User ID string is well-formed: Name (Comment) .

Key imports but shows 264 bits

Regenerate. Correct Ed25519 output is 256 bits.

GPG says "no valid data found"

You may be importing the public key into the secret keyring or vice versa. Both should import cleanly, but check which file you passed.

Security notes

Read these. They are not boilerplate.

Never paste a seed into a website.

Online converters are the single most common way people lose keys. The seed is the root of everything derived from it. If a site can read it, the site owns it.

Treat the seed and the derived PGP key as the same secret.

Compromising either compromises both. If you ever publish or leak the seed, the PGP key is compromised too, and you must revoke it and generate a new identity.

Do not reuse one seed across unrelated systems.

Using the same Ed25519 seed for SSH, a NEAR wallet, a Solana wallet, and PGP means a single leak destroys everything simultaneously. Derive separate seeds for separate purposes.

Back up the revocation certificate.

Most PGP clients generate one on import, or offer to. Store it offline. Without it, you cannot cleanly retire a compromised key.

Delete the script.

derive.mjs contains your seed in plaintext. It has no purpose after a successful run.

Plans For A Repo

```
ed25519-to-pgp/  
├── LICENSE           # MIT for code  
├── README.md         # This guide  
├── SECURITY.md        # Threat model and disclosure policy  
├── derive.example.mjs # Script with placeholder seed  
└── .gitignore        # *.asc, *.pgp, derive.mjs, node_modules
```

The .gitignore:

```
node_modules/  
*.asc  
*.pgp  
derive.mjs
```

Closing

The path from an Ed25519 seed to a working PGP key is short, but it requires care at three points: correct hex parsing, a well-formed User ID, and immediate cleanup of the plaintext private key. Do those three things correctly and the rest is mechanical.

The reason this guide exists is that no safe online tool performs this conversion. The only responsible method is local execution with a maintained library. Now you have that method written down.

Fork it, improve it, and keep your seed off the network.

This article is released under CC BY-SA 4.0. The accompanying script is released under MIT. Modify and redistribute freely with attribution.