Coldstar Documentation

Programmable cold-signing infrastructure for Solana. Air-gapped signing, Jupiter DEX integration, and native RWA support.

What is Coldstar?

Coldstar is programmable cold-signing infrastructure for Solana. Any USB drive becomes an air-gapped cold wallet. Air-gapped signing, Jupiter DEX integration, and native RWA support — built for DAOs, traders, and developers who need hardware-grade security without a $200 device.

The core idea

Hardware wallets keep your key alive in a secure chip **for years**. Coldstar takes the opposite approach: your key exists in memory **for microseconds**, only when needed, then it's gone. The rest of the time it sits encrypted on your USB drive, protected by a passphrase only you know.

It's a CLI tool with a Rich terminal UI, built for developers, traders, and operators who want to verify their own security — not trust a vendor's firmware.

Installation

You need Python 3.11+ and a USB drive. That's it.

Terminal

```
# 1. Clone the repo
git clone https://github.com/devsyrem/coldstar
cd coldstar-colosseum

# 2. Install Python dependencies
pip install -r local_requirements.txt

# 3. Launch — Coldstar guides you through setup
python main.py
```

CRYP2NOVA

Archived copy · cryp2nova.com

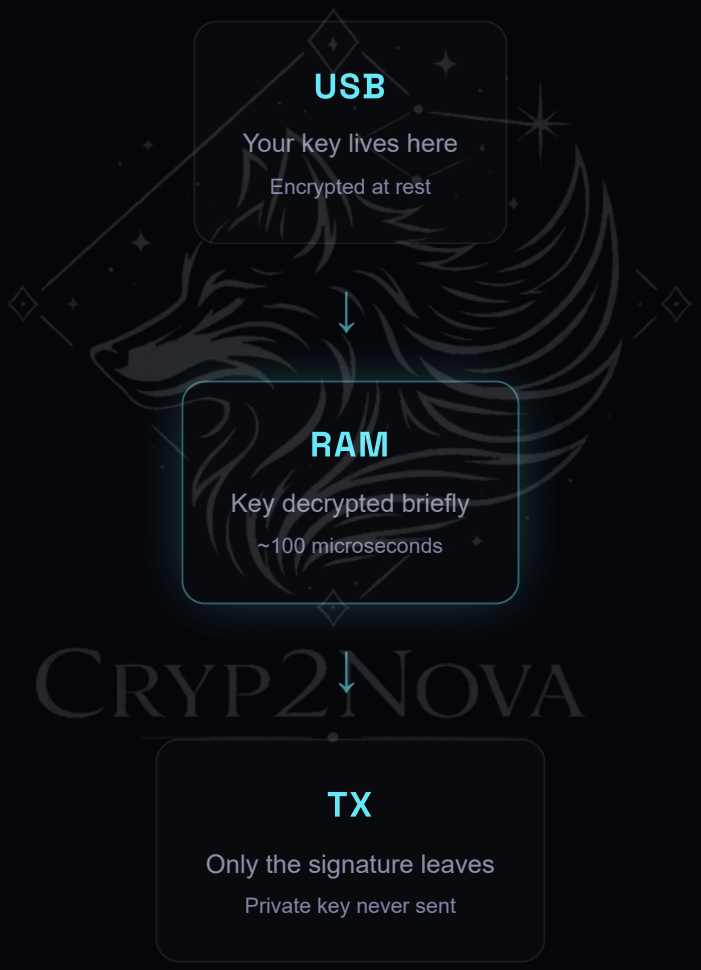
On first launch, Coldstar will detect your USB drive and walk you through creating or importing a wallet.

Want more control?

See the Build Guide for manual setup, Docker, Rust signer compilation, and air-gapped deployment.

How It Works

There are three stages: setup, signing, and broadcast. Here's what happens at each step.



1. **Setup:** You create or import a wallet. Coldstar encrypts your private key with a passphrase you choose and saves it to the USB drive. The plaintext key is immediately wiped from memory.
2. **Signing:** When you send a transaction, Coldstar reads the encrypted key from USB, decrypts it in locked RAM (memory that can't be swapped to disk), signs the transaction, and wipes the key. The whole process takes about 100 microseconds.
3. **Broadcast:** Only the transaction signature (public data) leaves the signing process. Your private key never touches the network, never touches disk, and never leaves the signing

Archived copy · cryp2nova.com

The USB is storage, not a signing device

Unlike hardware wallets, the USB drive doesn't run any code or sign anything. It's just an encrypted container for your key. All signing happens on your computer, in isolated memory.

Encryption & Key Storage

When you create a wallet, here's what happens under the hood:

1. **Key generation:** A 32-byte private key is created from your system's hardware random number generator (`/dev/urandom`).
2. **Passphrase protection:** Your passphrase is run through Argon2id — a memory-hard algorithm that uses 64 MB of RAM and 3 iterations. This makes brute-force attacks extremely slow, even with specialized hardware.
3. **Encryption:** The private key is encrypted with AES-256-GCM (the same standard used by governments and banks) and saved to your USB drive.
4. **Cleanup:** The plaintext key is overwritten with zeros in RAM. Nothing sensitive remains in memory.

COMPONENT	ALGORITHM	PURPOSE
Key encryption	`AES-256-GCM`	Encrypts private keys at rest on USB
Key derivation	`Argon2id`	Derives encryption key from passphrase (64 MB memory-hard)
Signing	`Ed25519`	Solana transaction signatures

Encrypted Container Format

keypair.json

```
{
  "version": 1,
  "salt": "base64_encoded_32_bytes",
  "nonce": "base64_encoded_12_bytes",
  "ciphertext": "base64_encoded_48_bytes",
```

Archived copy · cryp2nova.com

What if someone steals my USB?

They get a file full of encrypted noise. Without your passphrase, the data is computationally impossible to decrypt. You can destroy the USB and restore from a backup — your key is only as valuable as the passphrase protecting it.

RAM Signing Process

This is the most security-critical part of Coldstar. When you sign a transaction, here's exactly what happens:

1. **Read:** The encrypted key file is loaded from USB into Python. At this point, it's still encrypted — Python never sees the real key.
2. **Handoff:** The encrypted data is passed to the Rust signer (a compiled binary, not a script). This is where all sensitive work happens.
3. **Decrypt:** Inside Rust, your passphrase is used to re-derive the encryption key. This happens in *locked memory* — memory that the OS cannot swap to disk.
4. **Sign:** The private key is decrypted, used to sign the transaction (~100 microseconds), then immediately overwritten with zeros.
5. **Return:** Only the signature (public data) is returned to Python. The private key never leaves the Rust process.

CRYP2NOVA

Why Rust?

The plaintext private key **never enters Python memory**. Python can't guarantee memory cleanup — the garbage collector might leave key material in memory for seconds or longer. Rust's `Drop` trait guarantees zeroization, even if the program crashes.

Where your key exists at each stage

LOCATION	DATA STATE	DURATION
USB NAND Flash	Encrypted	Permanent

Archived copy · cryp2nova.com

Python RAM	Encrypted	Seconds
Rust mlock'd RAM	**Plaintext**	**~100 microseconds**
CPU registers	**Plaintext**	**~10 nanoseconds**
Network	Signature only	Never contains private key

Memory Protection

Three mechanisms work together to ensure your private key doesn't leak through memory:

mlock — Preventing Swap

Your OS can move idle memory to disk (swap). If your private key got swapped, it could be recovered later from the disk. Coldstar uses `mlock()` to tell the OS: *"keep this memory in physical RAM, never write it to disk."*

Zeroization — Cleaning Up

After signing, all buffers containing key material are overwritten with zeros. This uses Rust's `zeroize` crate, which guarantees the compiler won't optimize away the cleanup (a common problem in C/C++ where the compiler removes "unnecessary" memory writes).

Panic Safety — Crash Protection

Even if the signing process crashes mid-operation, Rust's `Drop` trait guarantees the cleanup code still runs. Your key is zeroized in all code paths — success, error, or crash.

Threat Model

No security tool protects against everything. Here's exactly what Coldstar handles and what it doesn't.

What Coldstar protects against

Archived copy · cryp2nova.com

- **Key theft from disk:** Your private key is never stored in plaintext. Even if your computer is compromised, attackers find only encrypted data.
- **Firmware backdoors:** There's no proprietary firmware to backdoor. Coldstar is open-source Python + Rust that you can audit yourself.
- **Supply-chain attacks:** Any $5 USB drive works. No specialized hardware means no supply chain to tamper with.
- **Physical seizure:** If someone takes your USB, they get encrypted noise. Destroy it and restore from backup.
- **Swap file forensics:** Memory locking (`mlock`) prevents your key from being written to swap space on disk.

What Coldstar does NOT protect against

- **Compromised OS:** If an attacker has root access to your machine while you're signing, they can read RAM directly. Mitigation: use an air-gapped machine.
- **Keyloggers:** A hardware keylogger can capture your passphrase as you type it. Mitigation: inspect your hardware, use a trusted keyboard.
- **Weak passphrases:** Argon2id makes brute-force slow but can't save a 4-character password. Use a strong, unique passphrase.
- **User error:** If you lose all copies of your USB and backup, your key is gone. Keep backups in separate physical locations.

> **Bottom line**
>
> Coldstar assumes you control your OS and can verify the code. If you don't trust your machine, run Coldstar on a dedicated air-gapped computer that has never connected to the internet.

USB Storage Structure

Here's what Coldstar creates on your USB drive:

```
USB Drive/
  wallet/   # Your encrypted wallet
    keypair.json   # Encrypted private key — this IS your wallet
    pubkey.txt   # Your Solana address (safe to share publicly)
  inbox/   # Drop unsigned transactions here (air-gapped mode)
```

Archived copy · cryp2nova.com

```
outbox/   # Signed transactions appear here, ready to broadcast
.coldstar/   # Internal system files
    last_boot_id   # Tracks which boot session last accessed the wallet
    backup/   # Automatic encrypted backups of your wallet
```

Every time you plug in your USB, Coldstar checks that your wallet file hasn't been corrupted. If it has, it automatically restores from the latest backup.

Coldstar vs Hardware Wallets

Different security models with different trade-offs. Neither is universally better — it depends on your threat model.

ASPECT	COLDSTAR	HARDWARE WALLETS
Key storage	Encrypted on USB, decrypted in RAM only when signing	Stored permanently in a secure chip
Key exposure time	~100 microseconds per transaction	Always present in the device (years)
Cost	Any USB drive ($5-10)	Proprietary device ($79-279)
Supply chain	No supply chain — commodity USB, open-source code	Must trust the vendor's hardware and firmware
OS compromise	Vulnerable during the ~100 microsecond signing window	Isolated by secure element even if OS is compromised
Scriptability	CLI-first — automate with cron, scripts, CI/CD	Requires manual button press for each transaction
Auditability	Fully open-source Python + Rust — read every line	Varies — some are open-source, most firmware is closed
Replaceable	Destroy and replace for $5, restore from backup	$79-279 to replace, seed phrase recovery

CRYP2NOVA
Archived copy · cryp2nova.com

Disclaimer

Coldstar is experimental software. Verify the code, understand the security model, and operate within its documented assumptions.

View on GitHub · Build Guide · Back to Home



Archived copy · cryp2nova.com