title

Zach Kelling

Satschel, Inc.

March 2026

SATSCHEL INC.

ALTERNATIVE TRADING SYSTEM
SEC / FINRA REGULATED

Custody Architecture Audit Report

From Custodial SSS to Non-Custodial MPC
Threshold Signing with Passkey-Bound Recovery

CONFIDENTIAL — INTERNAL USE ONLY
March 31, 2026

CTO Office — Satschel Inc.
Access: @satschel.com accounts only

Satschel Inc.
Alternative Trading System
March 31, 2026

To: Eric Choi, CEO
From: Zach Kelling, CTO
Re: Custody Architecture — Non-Custodial Transition

Eric,

This report documents the complete custody architecture transition from the current Satschel production system to the new Liquidity.io platform.

Key finding: The current "MPC" implementation is Shamir Secret Sharing, not true MPC. The full private key is reconstructed in server memory on every signing operation. Shards 2 and 3 are stored in the same MongoDB database encrypted with a single environment variable. The company has de facto custody of all user funds.

The fix: The new architecture uses CGGMP21 threshold MPC where the private key is never reconstructed. Combined with passkey-encrypted backup shards, the company provably cannot sign transactions without user biometric approval.

To your question: "Can we do anything without their consent?"

No. With the passkey-bound architecture:

- The ATS holds 1 of 3 shards — insufficient to sign
- The backup shard is encrypted with the user's passkey — we cannot decrypt it
- Users can exit independently using their device shard + passkey
- Institutions can run their own MPC nodes or use their own Cloud HSM

The MPC infrastructure (3-node cluster, CGGMP21/FROST protocols, HSM integration, WebAuthn, trade approval) is built and running. The remaining work is the frontend integration: shard export to user devices, biometric trade approval UI, and passkey backup flow.

The detailed technical audit follows.

Zach Kelling
Chief Technology Officer
Satschel Inc.

1 Executive Summary

The Satschel platform operates a fully custodial model where the company holds all cryptographic key material. The implementation labeled "MPC" is actually Shamir Secret Sharing (SSS)—the full private key is reconstructed in server memory on every signing operation.

The Liquidity.io replacement uses **CGGMP21 threshold MPC** where the private key is **never reconstructed**. Each party computes a partial signature independently. Combined with passkey-encrypted backup shards, the company provably cannot sign transactions without user biometric approval.

Table 1: Architecture comparison summary

Property	**Satschel (Old)**	**Liquidity.io (New)**
Protocol	Shamir SSS	CGGMP21 MPC
Key reconstructed?	Yes (every sign)	Never
Company can sign alone?	Yes	No
User can exit alone?	No	Yes
Backup shard	Company-decryptable	Passkey-encrypted
"HSM"	MongoDB + AES	GCP Cloud HSM
Custody status	Custodian	Co-signer

2 Satschel Production Architecture (from code)

Three services handle custody in the current production system:

2.1 fortress.api — BitGo Wrapper + Fiat Rails

`fortress.api` is not a signing service. It is a wrapper around third-party providers:

- **BitGo**: Crypto wallets, deposits, withdrawals, trading (70+ service directories)
- **Priority Passport**: Fiat transfers (ACH, wire)
- **Finix**: Card processing, KYC
- Calls `multisig.api` for on-chain fee handling

BitGo holds the actual crypto custody keys. Satschel calls BitGo's API with `BITGO_ACCESS_TOKEN` but never sees BitGo's private keys. The "mpcHot" and "mpcCustody" fields in the code are BitGo's own product flags, not Satschel's implementation.

This path is compliant — BitGo is a qualified custodian.

2.2 multisig.api — On-Chain Wallet + Shard Storage

`multisig.api` manages the on-chain Polkadot/Substrate wallets for private securities:

- **shards service**: Stores encrypted key shards via `kmsEncryptData()` (Google Cloud KMS)
- **sign service**: Uses `TransactionManager` (Polkadot signer)
- **multisig service**: Creates multisig wallets (`owners + requiredSigners`)
- Mint, burn, transfer, swap services for on-chain tokens

Source: `satschel/multisig.api/src/services/shards/index.ts`

```
const encryptData = await kmsEncryptData(shard); // GCP Cloud KMS
Services.get('shard-db').create({
  shard: encryptData, // Encrypted shard stored in MongoDB
});
```

Table 2: Shard storage — GCP KMS encrypts, but both server shards in same database

Shard	Holder	Encryption	Risk
Shard 1	User device	Stored client-side	User-controlled
Shard 2	MongoDB (multisig.api)	GCP Cloud KMS	Same DB as shard 3
Shard 3	MongoDB (multisig.api)	GCP Cloud KMS	Company has 2-of-3

Note: Primary encryption uses GCP Cloud KMS (`kmsEncryptData`), not a plain environment variable. `SHARD_ENCRYPTION_KEY` is a local fallback only. However, the fundamental issue remains: `multisig.api` holds both shards 2 and 3 in the same database, and anyone with GCP KMS access can decrypt both.

2.3 backend.token — Key Reconstruction + Signing

Source: `satschel/backend.token/src/libs/blockchain/polkadot.ts`

```
// GF(256) lookup tables for Shamir Secret Sharing (line 48)
// ...
// Reconstruct full ed25519 key from any 2 shards (line 124)
const reconstructedSecretKey = this.reconstructShamirSecret(
  share1, share2
);
// Full private key exists in server RAM at this point
const account = keyring.addFromSeed(secretKeyBytes);
```

The Shamir reconstruction logic is inline in `backend.token` (not imported from `mpc.crypto.wallet`). The gateway service receives two shards, reconstructs the complete ed25519 private key in server memory, signs the Polkadot/Substrate transaction, then discards the key.

This is Shamir Secret Sharing, not MPC. In true MPC (CGGMP21), the key is never reconstructed — each party computes a partial signature independently.

3 New Architecture: Liquidity.io MPC

3.1 True Threshold Signing

- **Protocol**: CGGMP21 (ECDSA/secp256k1) + FROST (EdDSA/Ed25519)
- **Cluster**: 3 MPC nodes in consensus mode (ZAP wire protocol)
- **Storage**: ZapDB (ChaCha20-Poly1305 encrypted per-node)
- **Key property**: Private key is never reconstructed — partial signatures only

Source: `lux/mpc/pkg/mpc/keygen_session.go`, `signing_session.go`

3.2 Shard Distribution

Table 3: Liquidity.io shard distribution — company holds only shard 2

Shard	Holder	Protection
Shard 1	User device	Secure Enclave (biometric-gated)
Shard 2	ATS / Transfer Agent	Cloud HSM (GCP/AWS)
Shard 3	HSM storage	Encrypted with user's passkey

3.3 Passkey-Encrypted Backup (Non-Custodial Guarantee)

The backup shard (shard 3) is encrypted with the user's WebAuthn passkey public key via P-256 ECDH key agreement. The HSM stores the ciphertext but cannot decrypt it — only the user's biometric (via their passkey's private key in Secure Enclave) can.

```
// HSM stores: E(shard3, ECDH(user_passkey_pubkey, ephemeral))
// Company has: shard2 + encrypted_blob
// Company can decrypt: shard2 only (1 of 3 = INSUFFICIENT)
// User decrypts: passkey -> ECDH shared secret -> decrypt shard3
```

Result: The company provably cannot sign any transaction without the user's biometric approval.

3.4 Signing Scenarios

Table 4: Signing scenarios — company alone is always insufficient

Scenario	Shards Used	Outcome
Normal trade	User(1) + ATS(2)	Biometric approval + co-sign
User lost device	Passkey(3) + ATS(2)	Passkey synced via iCloud
User exits platform	User(1) + Passkey(3)	No company involvement
Company compromised	Attacker has shard 2	Cannot sign (1 of 3)

4 Regulatory Impact

4.1 Satschel: Custodian

- SEC Rule 15c3-3 (Customer Protection Rule) applies
- Annual PCAOB-registered audit required
- Net capital requirements for customer funds
- Full fiduciary responsibility

4.2 Liquidity.io: Co-Signer (Non-Custodial)

- ATS holds 1 of 3 shards — insufficient for unilateral control
- User maintains self-custody via shard 1 + passkey-encrypted shard 3
- Potential qualified custodian exemption
- Reduced net capital and insurance requirements

5 Implementation Status

6 Conclusion

The Satschel custody architecture provides a false sense of security. The "MPC" implementation is Shamir SSS with full key reconstruction, all shards in a single database, and a single encryption key. The company has de facto custody of all user funds.

The Liquidity.io architecture fixes this at the protocol level: CGGMP21 ensures the key is never reconstructed, passkey-encrypted backup shards ensure the company cannot access shard 3, and users can independently exit with their device shard and passkey.

The remaining work (user shard export, frontend integration) builds on infrastructure that is already deployed and running in production.

Table 5: Implementation status across repositories

Component	Location	Status
CGGMP21 keygen + signing	`lux/mpc/pkg/mpc/`	Built
FROST EdDSA (Ed25519)	`lux/mpc/pkg/mpc/`	Built
HSM co-signing	`lux/mpc/pkg/hsm/`	Built
WebAuthn registration	`lux/mpc/pkg/api/handlers_webauthn.go`	Built
Trade approval (biometric)	`lux/mpc/pkg/api/handlers_trade_approval.go`	Built
Passkey-encrypted backup	`lux/mpc/pkg/api/handlers_settlement.go`	Built
MPC cluster (3 nodes)	K8s chain namespace	Live
IAM passkey storage	`hanzo/iam/object/user_webauthn.go`	Built
Per-user account mapping	`liquidity/ats/exchange.go`	Live
User shard export endpoint	`lux/mpc/pkg/api/`	Next
Frontend shard storage	`liquidity/app` (IndexedDB/Keychain)	TODO
Frontend trade approval	`liquidity/app` (push notification)	TODO

7 Addendum: Verified Production Flow

Important correction: The Satschel production system has two separate custody paths, not one. The Shamir SSS finding applies only to Path 2.

7.1 Path 1: Equities + Fiat (Compliant)

```
fortress.api -> BitGo API (crypto) / Alpaca Securities (equities)
```

BitGo holds the actual crypto custody keys — Satschel calls BitGo's API with an access token but never sees the private keys. For equities, Alpaca holds custody in their SIPC-insured omnibus account. Satschel is the broker, not the custodian. This path is compliant.

7.2 Path 2: On-Chain Private Securities (Problem)

```
exchange.frontend -> backend.token -> Polkadot/Substrate
```

The Shamir SSS implementation lives inline in `backend.token/src/libs/blockchain/polkadot.ts` (GF(256) tables at line 48, reconstruction at line 124). This path is feature-gated via Firebase (`isMPCEnabled = mpc_{config.name}`) and only enabled for specific tenants. Most production users use Path 1.

This is the path the passkey-bound MPC architecture replaces.

7.3 Architecture: Event-Driven Settlement (Not Cron)

The new Liquidity.io settlement is event-driven, not cron-based:

1. User approves trade via biometric (WebAuthn assertion on device)
2. ATS receives approval event with user's partial signature
3. ATS validates compliance (KYC, AML, offering checks)
4. ATS co-signs with shard 2 (partial signature)
5. Partial signatures combined → valid ECDSA signature
6. Transaction broadcast to Liquid EVM (chain ID 8675311)
7. Settlement confirmation event fires webhooks

The settlement cron (30s) exists only as a retry mechanism for failed broadcasts — it does not initiate signing. Signing is always initiated by user approval events.

```
User biometric -> WebAuthn assertion -> MPC partial sign
  -> ATS co-sign -> combine partials -> broadcast on-chain
```

```
  -> settlement confirmed -> webhook fired

Retry cron (30s): only retries ALREADY-SIGNED transactions
  that failed to broadcast (network issues, gas, etc.)
  Does NOT sign new transactions.
```

7.4 Deployment Flow

Stage	Old (Satschel)	New (Liquidity.io)
Development	dev branch	dev branch
Staging	release → *.stage.satschel.com	test → *.test.satschel.com
Beta	beta → beta.satschel.com	(no separate beta)
Production	master → apps-gke cluster	main → *.main.satschel.com

7.5 Summary

- **Path 1** (equities/fiat): Alpaca/BitGo custody — no change needed
- **Path 2** (on-chain): Shamir SSS → CGGMP21 MPC + passkey — fix in progress
- Settlement is event-driven (user approval), not cron-initiated
- Passkey-encrypted backup shard code committed to `lux/mpc`
- MPC cluster (3 nodes) live in production K8s
- Remaining: user shard export endpoint + frontend integration

---

This document is generated from code audit of the Satschel and Liquidity.io repositories. All code references verified against source as of March 31, 2026.