Shield is a web-based embedded wallet that makes Zcash shielded transactions as simple as email login. Users authenticate with email/social, receive a non-custodial shielded Zcash wallet automatically, and can send/receive private transactions without ever seeing a seed phrase.

Core Value Proposition

Demonstrate that shielded Zcash transactions can be **as easy as Venmo** while maintaining **true privacy and self-custody**. This is currently impossible with existing Zcash wallets that require seed phrase management and manual setup.

User Flows

Onboarding (First-time user)

1. User visits app, clicks "Get Started"
2. Authenticates via email OTP or Google OAuth
3. System generates shielded Zcash wallet automatically (behind the scenes)
4. User sees their unified address (u-address) and 0 ZEC balance
5. Shows faucet link to get testnet ZEC

Receiving ZEC

1. User clicks "Receive"
2. Shows unified address as QR code + copyable text
3. When funds arrive, balance updates (may take 1-2 minutes for sync)
4. Shows transaction in activity feed with encrypted memo if present

Sending ZEC

1. User clicks "Send"
2. Enters recipient address (u- or z-address), amount, optional memo
3. Reviews transaction details (amount, fee, memo preview)
4. Clicks "Send" - triggers authentication check
5. Backend generates proof + signs transaction (7-10 seconds)
6. Shows pending status, then confirmed status
7. Balance updates

Technical Architecture (Simplified for Hackathon)

```
Frontend (React/Next.js)
    ↓
Auth Service (NextAuth or similar)
    ↓
Key Management Service
    - Generate seed from CSPRNG
    - Encrypt seed with AES-256-GCM using user-specific key derived from
auth
    - Store encrypted seed in database
    - For signing: decrypt seed → derive keys → sign → purge from memory
    ↓
Zcash Service
    - Connect to public lightwalletd (Zec Pages or Nighthawk)
    - Build transactions using librustzcash via WASM or backend Rust
    - Generate proofs (backend only, too heavy for browser)
    - Broadcast transactions
    ↓
Testnet Blockchain
```

Key Decision: Proof Generation Location

- **Browser proving**: Impossible (40+ MB memory, 7+ seconds, WASM limitations)
- **Backend proving**: Required, use Rust service
- **Architecture**: Frontend builds transaction parameters → backend generates proof in Rust → returns signed tx → frontend broadcasts

---

Zcash-Specific Technical Stack

Core Libraries (Must Use)

1. librustzcash (Backend - Rust) The only production-ready option for shielded transactions.

Required crates:

```toml
zcash_primitives = "0.15"      # Transaction building, key management
zcash_proofs = "0.15"          # Proof generation (Sapling + Orchard)
```

```
zcash_client_backend = "0.12"  # Wallet logic, scanning
zcash_address = "0.5"          # Address parsing/generation
zcash_keys = "0.2"             # Key derivation, unified keys
bip39 = "2.0"                  # Mnemonic generation
```

Why this stack:

- Orchard support (latest, most efficient shielded protocol)
- Unified Address generation (modern standard)
- Production-tested by ECC in Zashi wallet
- Active maintenance

2. Lightwalletd Connection

Use **public testnet servers** (don't run your own):

- Zec Pages: `testnet.lightwalletd.com:9067`
- Nighthawk: `testnet.lightwalletd.nighthawkapps.com:9067`

Protocol: gRPC with TLS

Client library options:

- Rust: `tonic` (gRPC client) + generated code from lightwalletd proto files
- Or use `zcash_client_backend` which includes light client abstractions

3. Frontend Integration

Since proving happens backend, frontend only needs:

- Address display/QR generation
- Transaction parameter collection (recipient, amount, memo)
- Balance/transaction display
- WebSocket or polling for transaction status

No Zcash-specific libraries needed in frontend - all complexity stays in Rust backend.

Critical Implementation Path

Phase 1: Key Management (Days 1-2)

Generate Unified Spending Key:

```
BIP39 mnemonic (24 words)
    → Seed (512 bits)
    → ZIP 32 master key (m/32'/133'/0')
    → Unified Spending Key (contains Orchard + Sapling keys)
    → Unified Full Viewing Key (for scanning)
    → Unified Address (u-address)
```

Critical Zcash specifics:

- Use `zcash_keys::keys::UnifiedSpendingKey`
- Derivation path: `m/32'/133'/0'` for first account
- Testnet network: `zcash_primitives::consensus::Network::TestNetwork`
- Must generate **Unified Address** not just Orchard address (for compatibility)

Storage:

- Encrypt the BIP39 mnemonic, NOT the derived keys
- Derive keys fresh for each operation
- Use AES-256-GCM with key derived from user's auth credentials + salt

Code location: Backend Rust service with `/generate-wallet` and `/get-address` endpoints

---

Phase 2: Balance & Transaction Scanning (Days 2-3)

Challenge: Zcash requires scanning blockchain to find your transactions (unlike account-model chains with instant balance queries).

Solution for Hackathon:

1. Connect to lightwalletd gRPC service
2. Use `zcash_client_backend::data_api::chain::scan_cached_blocks`
3. Store "wallet birthday" (block height when wallet created) - only scan from there
4. First sync takes 1-2 minutes even on testnet (show progress bar)

Critical implementation:

```
For each compact block from wallet birthday to chain tip:
    - Download compact block via lightwalletd
    - Trial-decrypt shielded outputs with Unified IVK
    - If successful decrypt: this is your transaction
    - Store note value, memo, nullifier
    - Update balance
```

Performance hack for hackathon:

- Cache compact blocks in local database (SQLite works)
- Only fetch new blocks after initial sync
- Use `zcash_client_backend::data_api::WalletWrite` trait for persistence

Data to track:

- Received notes (value, memo, spent/unspent status)
- Nullifiers (to detect when notes are spent)
- Witness data (for proof generation)

Phase 3: Transaction Building & Proof Generation (Days 4-6)

This is the hardest part. Zcash transactions require zk-SNARK proofs taking 7+ seconds to generate.

Transaction Flow:

Step 1: Note Selection

```
User wants to send X ZEC:
    - Query unspent notes from database
    - Select notes totaling >= X ZEC (greedy algorithm works)
    - Must include change output if notes > X ZEC
```

Step 2: Build Transaction Use `zcash_primitives::transaction::builder::Builder`:

```
Builder::new(network, block_height)
    .add_orchard_spend(fvk, note, merkle_path)  // for each input note
    .add_orchard_output(recipient_address, amount, memo)  // payment
```

```
    .add_orchard_output(change_address, change_amount, None)  // change
    .build(prover)  // generates proofs — takes 7+ seconds
```

Step 3: Proof Generation

- Load proving keys (zkparams files, ~50MB download first time)
- Store in persistent cache (`~/.zcash-params/` or server equivalent)
- Generation: `zcash_proofs::prover::LocalTxProver`
- **This blocks for 7-10 seconds** - cannot be avoided

Step 4: Sign & Broadcast

- Add signature with spending key
- Serialize transaction
- Broadcast via lightwalletd `SendTransaction` RPC
- Return transaction ID to frontend

Critical files needed: Download from Zcash parameters:

- `sapling-spend.params` (48MB)
- `sapling-output.params` (3.5MB)
- `orchard.params` (??MB)

Use `zcash_proofs::download_parameters()` to fetch automatically.

Phase 4: Frontend UX (Days 6-8)

Address the proof generation delay:

Bad UX:

```
User clicks "Send" → hangs for 10 seconds → transaction sent
```

Good UX:

```
User clicks "Send"
    → Modal: "Generating private proof..." with progress animation
    → "Signing transaction..." (when proof done)
    → "Broadcasting..."
```

```
    → "Sent! Confirming..."
    → Success with transaction ID
```

Implementation:

- WebSocket connection to backend for real-time status updates
- Backend sends status events: `proving_started`, `proving_complete`, `signing`, `broadcasting`, `confirmed`
- Show animated shield icon during proving (privacy theater)

Transaction Status Tracking:

```
1. Broadcast (immediate)
2. In mempool (10-20 seconds)
3. Included in block (75 seconds = 1 Zcash block)
4. Confirmed (10+ blocks for safety = 12.5 minutes, but show as
"pending" after 1 block)
```

Poll lightwalletd for transaction status or subscribe to new blocks via streaming RPC.

Phase 5: Testing & Demo Prep (Days 8-10)

Get Testnet ZEC:

- Use Zcash testnet faucet: https://faucet.zecpages.com/
- Or mine locally with zcashd in regtest mode for faster testing

Critical Test Cases:

1. **New wallet generation** - verify u-address format starts with `utest1`
2. **Receive transaction** - send from another testnet wallet, verify balance updates
3. **Sync from scratch** - delete wallet state, re-scan from birthday
4. **Send transaction** - verify proof generation completes, transaction broadcasts
5. **Memo support** - send transaction with encrypted memo, verify recipient sees it
6. **Insufficient funds** - handle gracefully when user tries to send more than balance
7. **Invalid address** - reject malformed recipient addresses with clear error

Demo Flow:

```
1. Show landing page - "Get your private wallet in 10 seconds"
2. Login with Google (fastest for demo)
3. Wallet generates (show behind-scenes animation)
4. Show empty wallet with u-address QR code
5. Send testnet ZEC from pre-funded wallet (yours)
6. Balance updates (refresh if needed, or show WebSocket live update)
7. Send ZEC back to your address with memo "Thanks for the demo!"
8. Show proof generation process (7 seconds of suspense)
9. Transaction succeeds
10. Show transaction history with encrypted memo visible
```

Zcash-Specific Challenges & Solutions

Challenge 1: Proof Generation Performance

Problem: 7-10 second blocking operation, can't parallelize easily, too heavy for browser.

Solutions:

- ✅ **Backend-only proving** - Keep proving in Rust service, frontend just waits
- ✅ **UX solutions** - Great progress indicators, explain "generating privacy proof"
- ✅ **Proof caching** - For repeated recipients, consider memo-only changes (out of scope for hackathon)
- ❌ Don't try: Browser WASM proving (doesn't work reliably)

Challenge 2: Initial Sync Latency

Problem: New wallet must scan blockchain from birthday, takes 1-2 minutes.

Solutions:

- ✅ **Set wallet birthday to current block** when generating (not genesis)
- ✅ **Background sync** - Start syncing immediately after wallet creation, before user navigates away
- ✅ **Clear expectations** - "Syncing your private transactions... 45 seconds remaining"
- ✅ **Spend-before-sync** - User can generate address and share it before sync completes
- ❌ Don't try: Running full node (overkill for hackathon)

Challenge 3: Unified Address Complexity

Problem: Zcash has multiple address types (Orchard, Sapling, transparent). Unified Addresses bundle them.

Solutions:

- ✅ **Always generate Unified Addresses** - Use `zcash_keys::keys::UnifiedAddress`
- ✅ **Accept z-addresses as recipient** - Use `zcash_address::ZcashAddress::try_from_encoded()` to parse any valid address type
- ✅ **Orchard-first spending** - Prefer Orchard notes when available (best privacy, most efficient)
- ⚠️ **Testnet format** - Unified addresses start with `utest1` on testnet, `u1` on mainnet

Challenge 4: Note Management

Problem: UTXO-like model where "notes" are spent/unspent, not simple account balances.

Solutions:

- ✅ **Use SQLite with zcash_client_backend schemas** - Don't invent your own note tracking
- ✅ **Nullifier tracking** - Store nullifiers to detect spent notes
- ✅ **Change outputs** - Always generate change back to self when notes > amount
- ✅ **Note selection** - Greedy algorithm (largest notes first) works fine for hackathon
- ❌ Don't try: Fancy coin selection algorithms (overkill)

Challenge 5: Memo Encryption

Problem: Memos are 512 bytes max, encrypted to recipient.

Solutions:

- ✅ **Optional memo field** - Let user add text message (Twitter-style, show character count)
- ✅ **Encrypt automatically** - `zcash_primitives` handles encryption when you call `add_orchard_output` with memo parameter
- ✅ **Display received memos** - Decrypt during sync, store plaintext in database

- ⚠️ **512 byte limit** - Truncate or warn if user types >512 bytes (UTF-8 encoded)

Challenge 6: Fee Calculation

Problem: Zcash transactions have variable fees based on action count.

Solutions:

- ✅ **Default fee** - Use ZIP 317 conventional fee: 10,000 zatoshis (0.0001 ZEC) works for most transactions
- ✅ **Show fee to user** - Display "Network fee: 0.0001 ZEC" in send confirmation
- ✅ **Account for fee** - When sending "max", ensure you send (balance - fee) not full balance
- ❌ Don't try: Dynamic fee estimation (not critical for testnet hackathon)

Challenge 7: Error Handling

Problem: Lots can go wrong (lightwalletd down, invalid address, insufficient balance, proof generation failure).

Solutions:

- ✅ **Graceful degradation** - If lightwalletd is down, show cached balance with warning "Unable to sync"
- ✅ **Input validation** - Validate address format before attempting transaction
- ✅ **Insufficient funds** - Calculate max sendable (balance - fee) and disable send if amount > max
- ✅ **Retry logic** - If broadcast fails, retry up to 3 times with exponential backoff
- ✅ **User-friendly errors** - Don't show "proof generation failed: error code 0xABCD", say "Transaction failed. Please try again."

Key Files & Setup

Backend Rust Service Structure

```
/backend
  /src
    main.rs                 # Axum/Actix server
    wallet.rs              # Key generation, address derivation
```

```
    scanner.rs             # Blockchain scanning via lightwalletd
    transaction.rs         # Transaction building & proving
    lightwalletd.rs        # gRPC client for lightwalletd
  Cargo.toml
```

Environment Setup

```
ZCASH_NETWORK=test
LIGHTWALLETD_URL=testnet.lightwalletd.com:9067
DATABASE_URL=sqlite:wallet.db
ENCRYPTION_KEY=<generated-key>
```

Database (SQLite)

Use `zcash_client_backend::data_api::wallet::init::init_wallet_db` to create schema automatically. Don't create custom schema - the SDK expects specific tables.

Required tables (created by SDK):

- `accounts` - Wallet accounts
- `transactions` - Transaction metadata
- `received_notes` - Shielded notes (Orchard/Sapling)
- `sapling_witnesses` / `orchard_witnesses` - Merkle path data for proving
- `nullifiers` - Spent note tracking

Success Criteria for Hackathon

Must Have:

- [ ] User can authenticate with email or social OAuth
- [ ] Wallet generates automatically with unified address (u-address)
- [ ] Wallet syncs and shows balance from testnet
- [ ] User can receive shielded ZEC to their address
- [ ] User can send shielded ZEC with proof generation
- [ ] Encrypted memos work (send & receive)
- [ ] Transaction history shows all shielded transactions

Nice to Have:

- [ ] QR code generation for receiving
- [ ] Real-time balance updates via WebSocket
- [ ] Mobile-responsive UI
- [ ] Transaction status animations during proof generation
- [ ] Export wallet (show BIP39 mnemonic)

Out of Scope:

- Mainnet deployment
- Hardware wallet integration
- Multi-account support
- Transparent address support
- Custom lightwalletd infrastructure
- Fee optimization

Demo Strategy

30-Second Pitch: "Zcash is the only cryptocurrency with true privacy through zero-knowledge proofs. But every existing Zcash wallet requires seed phrase management and technical setup. Shield makes private transactions as easy as Venmo - just login and send. No seed phrases, no complexity, full privacy maintained."

2-Minute Demo:

1. Load app, click "Get Started", login with Google (5 seconds)
2. Wallet generates, show u-address QR code (5 seconds)
3. "I've already sent testnet ZEC here from another wallet" - refresh, balance appears (10 seconds)
4. Click send, enter recipient + amount + memo (15 seconds)
5. Click send, show "Generating privacy proof..." modal (7 seconds of suspense)
6. Transaction broadcasts, show pending → confirmed (20 seconds)
7. Show transaction in history with decrypted memo (5 seconds)
8. "Under the hood, we just generated a zero-knowledge proof that this transaction is valid WITHOUT revealing sender, recipient, or amount to anyone except the recipient."

Technical Deep-Dive (if judges ask):

- Explain key management (encrypted seed, derived keys)
- Show proof generation backend code
- Explain scanning process (trial decryption with viewing keys)
- Discuss unified addresses vs legacy z-addresses
- Mention Orchard protocol (latest shielded pool)

Risk Mitigation

Risk 1: Proof generation too slow for demo

- **Mitigation:** Pre-generate test wallets with balances, avoid cold starts during demo
- **Backup plan:** Use pre-recorded video of proof generation if live demo fails

Risk 2: Lightwalletd unreliable during demo

- **Mitigation:** Test multiple public servers beforehand, add failover in code
- **Backup plan:** Run local zcashd + lightwalletd if needed (more setup but reliable)

Risk 3: Blockchain sync takes too long

- **Mitigation:** Create demo wallet 1 day before, let it sync overnight
- **Backup plan:** Set wallet birthday to very recent block (last hour)

Risk 4: Complexity exceeds 10 days

- **Mitigation:** Cut scope ruthlessly - remove memos, remove export, simplify UI
- **Backup plan:** Pre-build transaction signing service, focus frontend on UX only

---

Development Priorities

Day 1-2: Prove It Works Get a Rust service that can generate keys and build/prove a transaction successfully. If this doesn't work, pivot immediately. Everything else is polish.

Day 3-4: End-to-End Flow Connect frontend → backend → lightwalletd → blockchain. One complete send transaction working = project is viable.

Day 5-7: UX Polish Make proof generation feel intentional (not broken), add nice animations, improve error messages.

Day 8-10: Demo Prep Test on multiple devices, prepare fallback demos, rehearse pitch, handle edge cases.

Key Differentiators from Existing Wallets

Zashi (official ECC wallet):

- Requires seed phrase backup immediately
- Not embeddable in other apps
- Mobile-first, not web

Nighthawk Wallet:

- Same seed phrase friction
- Privacy-focused power users only

Your Project (Shield):

- Zero seed phrase friction for initial use
- Embeddable architecture (hackathon shows web, but could be SDK for other apps)
- Targets mainstream users, not crypto natives
- Proves "Zcash privacy for everyone" is possible

This is the innovation: **Embedded + Shielded**. Nobody has built this yet. Every embedded wallet is transparent (Privy, Dynamic, etc.). Every shielded wallet is self-custody with seed phrases. You're the first to combine them.