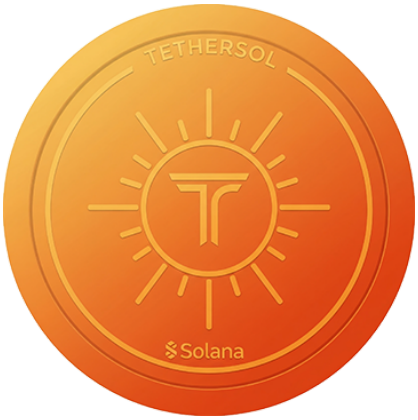

Tethersol Protocol: A Comprehensive Technical Paper

Document Version: 3.2

Publication Date: September 6, 2025

Abstract

This document serves as an in-depth technical reference for the Tethersol protocol, a decentralized system built on the Solana blockchain for the tokenization of real-world assets (RWAs). This iteration provides a comprehensive overview of the system's architecture, including its hybrid on-chain/off-chain design, a detailed breakdown of the smart contract suite, on-chain data structures, and end-to-end workflows. We delve into specific implementation details, security measures, and future evolutions. This paper is intended to provide developers, security auditors, system architects, and stakeholders with the necessary insights to build, audit, or integrate with the protocol.

1.0 System Philosophy & Design Goals

Tethersol addresses the fundamental challenges of illiquidity, high entry barriers, and the trust gap in RWA tokenization by synthesizing RWA stability with DeFi efficiency. Our architecture is built on five core pillars:

- **Security:** We emphasize fund safety through rigorous security audits, a public bug bounty program, and secure on-chain patterns such as **Program Derived Accounts (PDAs)** and reentrancy guards.
- **Scalability:** We utilize Solana's Sealevel runtime and Proof-of-History (PoH) consensus to support high throughput (e.g., 50,000 TPS) and low transaction costs ($<0.00025 per transaction), making micro-transactions like daily yield distributions economically viable.
- **Decentralization:** We transition progressively from a multi-signature controlled entity to a fully community-governed DAO.
- **Composability:** We adhere to Solana Program Library (SPL) standards to enable seamless integration with other DeFi protocols (e.g., Serum, Raydium), allowing $TRSOL to be used as collateral in the broader DeFi ecosystem.
- **Legal Compliance:** We ensure a robust off-chain legal framework, including the use of bankruptcy-remote SPVs, to legally bind the on-chain tokens to their real-world counterparts.

2.0 Core Protocol Architecture

The Tethersol architecture employs a hybrid model that tightly synchronizes off-chain legal assets with on-chain transparency and functionality. The system is designed to minimize the trust required in the off-chain processes while using the blockchain to enforce a transparent and immutable ledger of ownership and value.

2.1 On-Chain Layer (Solana)

The on-chain layer is a suite of modular programs written in Rust using the Anchor framework, providing a secure and developer-friendly environment.

- **Program Derived Accounts (PDAs):** All protocol funds are held in program-controlled PDAs, eliminating the need for private keys and mitigating custodial risk. For example, the staking vault PDA is deterministically derived from the program ID and a unique seed.
- **Solana Account Model:** We utilize rent-exempt accounts for persistent state storage, with efficient serialization via Borsh.

2.2 Off-Chain Layer

This layer handles the physical and legal aspects of the assets.

- **Legal & Asset Management:** Assets are owned and managed by professional firms, each held within a separate, bankruptcy-remote SPV (e.g., a Delaware Series LLC). The SPV's legal ownership shares are then tokenized.
- **Data Verification & Oracle Bridge:** Independent auditors verify the financial performance of the assets (e.g., Net Operating Income). This data is securely signed and pushed to a decentralized oracle network. The on-chain protocol validates this signed data before triggering any financial actions.

3.0 Smart Contract Suite: A Deep Dive

The Tethersol protocol consists of a set of interconnected Solana programs, each serving a specific and auditable purpose. These programs communicate via **Cross-Program Invocations (CPIs)** to execute complex workflows securely.

3.1 The Tethersol Staking Program

This is the primary user-facing program. It manages the staking and unstaking of $TRSOL tokens and facilitates the claiming of yield. The program's state is stored in two key accounts: StakingVaultState (a global account) and StakerAccount (a per-user account).

- **initialize_vault**: A one-time instruction to initialize the protocol's central staking vault and state accounts, controlled by a PDA.
- **deposit**: Allows a user to transfer $TRSOL tokens into the staking vault. It updates the user's StakerAccount and the global StakingVaultState. The function ensures the transaction is an atomic operation, updating both accounts before the CPI is completed.
- **withdraw**: Enables a user to unstake and retrieve their $TRSOL tokens, subject to any potential lock-up periods or governance decisions.
- **claim_yield**: This function is stateless and permissionless. It calculates and distributes a user's pro-rata share of the collected stablecoin yield from the Yield Distributor PDA. This function relies on a CPI to the SPL Token program to perform the transfer. The yield calculation is based on the formula:
  Y_user=(S_user/S_total)timesA_yield
  Where:
  - Y_user is the yield to be claimed by the user.
  - S_user is the user's staked balance.
  - S_total is the total staked balance in the vault.
  - A_yield is the total available yield in the Yield Distributor PDA.

After the transfer, the last_claimed_timestamp is updated to prevent double-claims of the same yield distribution.

```
// TethersolStaking/lib.rs (Simplified Pseudocode)
use anchor_lang::prelude::*;
use anchor_spl::token::{Token, TokenAccount, transfer};
use std::convert::TryInto;

#[program]
pub mod tethersol_staking {
    use super::*;

    // State Account for the Staking Vault
```

```rust
    #[account]
    pub struct StakingVaultState {
        pub total_staked_balance: u64,
        pub bump: u8,
    }

    // Per-user Staker Account
    #[account]
    #[derive(Default)]
    pub struct StakerAccount {
        pub owner: Pubkey,
        pub staked_balance: u64,
        pub last_claimed_timestamp: i64,
        pub bump: u8,
    }

    pub fn initialize_vault(ctx: Context<InitializeVault>) -> Result<()> {
        let vault_state = &mut ctx.accounts.vault_state;
        vault_state.total_staked_balance = 0;
        Ok(())
    }

    // Function to handle staking
    pub fn deposit(ctx: Context<Deposit>, amount: u64) -> Result<()> {
        // Transfer $TRSOL from user to the vault PDA
        let cpi_accounts = Transfer {
            from: ctx.accounts.user_tsol_account.to_account_info(),
            to: ctx.accounts.vault_tsol_account.to_account_info(),
            authority: ctx.accounts.user_authority.to_account_info(),
        };
        let cpi_program = ctx.accounts.token_program.to_account_info();
        transfer(CpiContext::new(cpi_program, cpi_accounts), amount)?;

        // Update user's and vault's staked balance
        let staker_account = &mut ctx.accounts.staker_account;
        staker_account.staked_balance =
staker_account.staked_balance.checked_add(amount).unwrap();
        ctx.accounts.vault_state.total_staked_balance =
ctx.accounts.vault_state.total_staked_balance.checked_add(amount).unwrap();

        Ok(())
    }
```

```
    // Function to claim yield
    pub fn claim_yield(ctx: Context<ClaimYield>) -> Result<()> {
        let staker = &mut ctx.accounts.staker_account;
        let vault_state = &ctx.accounts.vault_state;
        let yield_vault = &ctx.accounts.yield_vault;
        let yield_token_program = &ctx.accounts.yield_token_program;

        // Calculate user's pro-rata share
        let user_share = (staker.staked_balance as f64) /
(vault_state.total_staked_balance as f64);
        let available_yield = yield_vault.amount as f64;
        let yield_to_claim = (user_share * available_yield) as u64;

        // Transfer yield from the yield vault PDA to the user
        let bump =
*ctx.accounts.yield_vault.to_account_info().try_borrow_data()?[0];
        let seeds = &[
            b"yield_vault".as_ref(),
            &[bump],
        ];
        let signer_seeds = &[&seeds[..]];
        let cpi_accounts = Transfer {
            from: ctx.accounts.yield_vault_token_account.to_account_info(),
            to: ctx.accounts.user_yield_account.to_account_info(),
            authority: ctx.accounts.yield_vault.to_account_info(),
        };
        let cpi_program = yield_token_program.to_account_info();
        transfer(CpiContext::new_with_signer(cpi_program, cpi_accounts,
signer_seeds), yield_to_claim)?;

        // Update user's last claim timestamp
        staker.last_claimed_timestamp = Clock::get()?.unix_timestamp;

        Ok(())
    }
}
```

3.2 The Tethersol Governance Program

The core of the DAO, this program manages the lifecycle of all on-chain proposals and voting

using staked $TRSOL. It works in conjunction with a separate **Timelock program** to enforce a delay before execution.

- **create_proposal**: A user with a minimum staked amount of $TRSOL can submit a proposal. The instruction includes a serialized payload containing the actions to be executed upon a successful vote.
- **cast_vote**: Stakers can vote 'For' or 'Against' a proposal. The voting power is based on the user's staked $TRSOL at the time the proposal was created.
- **queue_proposal**: When a proposal passes, it is moved to a queue within the Timelock program, starting a mandatory waiting period.
- **execute_proposal**: After the Timelock period expires, anyone can trigger the execution of the proposal's instructions via a CPI.

3.3 The Tethersol Yield Distributor Program

This is a permissioned program that acts as the bridge between the oracle network and the staking vault. It is responsible for the secure and verifiable distribution of yield.

- **ingest_oracle_data**: A privileged function that validates signed data from the oracle network. It checks the signature and the integrity of the data before using it to calculate the amount of stablecoin to distribute.
- **distribute_yield**: Queues the calculated stablecoin amount from the protocol treasury PDA to the Yield Distributor PDA. This amount is then made available for users to claim.

3.4 The Tethersol Asset Registry Program

This program provides an immutable, on-chain record for each tokenized RWA. It links the on-chain representation (SPL token) to the off-chain legal documents and metadata.

- **register_asset**: A one-time, permissioned function that creates an on-chain account for a new RWA. The account's data includes a hash of the legal documents and a link to their storage on a decentralized file system like IPFS.

3.5 The Tethersol Vesting Program

This program enforces the vesting schedules for the team, advisors, and treasury. This ensures that tokens are released according to a predetermined schedule and are not immediately available for sale.

- **create_vesting_schedule**: Initializes a vesting account for a specific beneficiary, defining the cliff duration, total duration, and total amount of tokens to be released.
- **claim_vested_tokens**: Allows the beneficiary to claim any tokens that have been released from their vesting schedule.

3.6 The Tethersol Lending Program (Future Implementation)

This program will allow users to borrow stablecoins against their staked $TRSOL or other tokenized assets.

- **create_loan**: Creates a loan account for a user, locking their collateral in a PDA. The loan parameters (e.g., loan-to-value ratio, interest rate) are defined.
- **repay_loan**: Allows a user to repay their loan and retrieve their locked collateral.
- **liquidate_loan**: If a loan's collateral falls below a predefined threshold, this function allows liquidators to repay the loan and seize the collateral.

4.0 Security and Risk Mitigation

Security is multi-layered and paramount.

Attack Vector	Mitigation Strategy
Reentrancy Attack	All sensitive state changes (e.g., updating balances) are performed before any cross-program invocations (CPIs) that transfer tokens out of the protocol. We use Anchor's built-in #[payable] and reentrancy guards.
Oracle Manipulation	The protocol relies on a multi-oracle fallback system and on-chain price bounds. If the oracle feed provides data outside of a predefined range, the YieldDistributor program will revert the transaction.
Malicious Governance Proposals	A **Timelock** contract queues all passed proposals for a minimum of 48-72 hours before they can be executed. This provides the community and security auditors ample time to detect and respond to any malicious proposals.
Admin Key Compromise	All administrative keys start as a multi-sig wallet. Over time, control of these keys is progressively migrated to the DAO's Timelock program, making the protocol immutable and censorship-resistant.
Asset Performance Risk	The DAO and a dedicated asset committee

	will implement a diversification strategy across different asset classes (real estate, agriculture) and geographies to minimize risk. All assets are insured, and this information is verifiable on-chain via the TethersolAssetRegistry.
Front-running	The protocol is designed to minimize opportunities for front-running. The claim_yield function, for example, is not time-sensitive, so users cannot be out-competed to claim yield.

5.0 Future Development

Building on the foundation laid in Phase 2, we have a clear vision for the protocol's evolution:

- **Native Lending Protocol:** We will launch a native lending and borrowing platform. Tokenized asset shares will serve as a new class of on-chain collateral, allowing users to unlock liquidity against their assets. This will be a major step in expanding the utility of RWAs in DeFi.
- **Cross-Chain Interoperability:** Using bridge solutions like Wormhole or LayerZero, we will enable the seamless transfer of $TRSOL to other major blockchain ecosystems (e.g., Ethereum, Base), expanding the token's utility and liquidity. This will allow the protocol to tap into new markets and user bases.
- **Advanced Features:** We will explore the use of zero-knowledge proofs (zk-proofs) to enable private yield claims and other privacy-preserving features. We will also investigate AI-driven asset selection and portfolio management tools for the DAO.
- **Token Upgrades:** We will migrate the $TRSOL token to the SPL Token-2022 standard to take advantage of new features such as transfer hooks and confidential transfers.

6.0 Conclusion

The Tethersol protocol pioneers the RWA-DeFi integration, offering a secure, scalable, and legally-compliant platform for tokenized assets. This technical paper provides a detailed blueprint for a more equitable financial future where tangible assets can fuel decentralized economies.