GWIM Smart Contracts Technical Specification

Overview

This document provides comprehensive technical specifications for the smart contracts required to implement the GWIM token ecosystem on the Solana blockchain. The specification covers four main components:

1. GWIM Token Contract
2. Presale Contract
3. Staking Contract
4. DAO Governance Contract

1. GWIM Token Contract

Contract Type

- Solana Program Library (SPL) Token

Token Parameters

- **Name**: GWIM Token
- **Symbol**: GWIM
- **Total Supply**: 210,000,000 tokens
- **Decimals**: 6
- **Contract Address**: BFF6n9ErsHUyj9MnutmmxxiXSXY1skKX9Pib8bWFgrcU

Token Distribution

- **Presale Allocation**: 52,500,000 tokens (25%)
- **Team Allocation**: TBD
- **Ecosystem Development**: TBD
- **Marketing**: TBD
- **Treasury**: TBD

Key Functions

Initialize Token

```
pub fn initialize_token(
   program_id: &Pubkey,
   mint_authority: &Pubkey,
   freeze_authority: Option<&Pubkey>,
   decimals: u8,
) -> ProgramResult
```

Mint Tokens

```
pub fn mint_to(
   program_id: &Pubkey,
   mint_pubkey: &Pubkey,
   account_pubkey: &Pubkey,
   owner_pubkey: &Pubkey,
   signer_pubkeys: &[&Pubkey],
   amount: u64,
) -> ProgramResult
```

Transfer Tokens

```
pub fn transfer(
   program_id: &Pubkey,
   source_pubkey: &Pubkey,
   destination_pubkey: &Pubkey,
   authority_pubkey: &Pubkey,
   signer_pubkeys: &[&Pubkey],
   amount: u64,
) -> ProgramResult
```

Security Considerations

- Mint authority should be transferred to a multi-signature wallet after initial setup
- Consider implementing a token timelock for team and advisor allocations
- Implement proper access controls for administrative functions

2. Presale Contract

Contract Type

- Custom Solana Program

State Variables

```
pub struct PresaleState {
  // Admin authority
  pub authority: Pubkey,

  // GWIM token mint address
  pub token_mint: Pubkey,

  // Treasury wallet to receive SOL payments
  pub treasury_wallet: Pubkey,

  // Presale stages configuration
  pub stages: [PresaleStage; 4],

  // Current active stage
  pub current_stage: u8,

  // Total tokens sold
  pub total_tokens_sold: u64,

  // Initial presale supply
  pub initial_presale_supply: u64,

  // Remaining presale supply
  pub remaining_presale_supply: u64,

  // Current token price in USD (6 decimals)
  pub current_price_usd: u64,

  // SOL price in USD (6 decimals)
  pub sol_price_usd: u64,

  // Last SOL price update timestamp
  pub last_price_update: i64,

  // Mapping of wallet addresses to amount purchased
  pub purchases: BTreeMap<Pubkey, u64>,

  // Is presale active
  pub is_active: bool,
}
```

```rust
pub struct PresaleStage {
    // Stage start date (Unix timestamp)
    pub start_date: i64,

    // Stage end date (Unix timestamp)
    pub end_date: i64,

    // Initial token price for this stage (in USD, 6 decimals)
    pub initial_price: u64,

    // Maximum tokens to sell in this stage
    pub token_cap: u64,

    // Tokens sold in this stage
    pub tokens_sold: u64,

    // Is stage active
    pub is_active: bool,
}
```

Key Functions

Initialize Presale

```rust
pub fn initialize_presale(
    program_id: &Pubkey,
    authority: &Pubkey,
    token_mint: &Pubkey,
    treasury_wallet: &Pubkey,
    initial_presale_supply: u64,
    initial_price: u64,
    stage_start_dates: [i64; 4],
    stage_end_dates: [i64; 4],
    stage_token_caps: [u64; 4],
) -> ProgramResult
```

Start Presale

```rust
pub fn start_presale(
    program_id: &Pubkey,
    authority: &Pubkey,
) -> ProgramResult
```

Update SOL Price

```rust
pub fn update_sol_price(
    program_id: &Pubkey,
    authority: &Pubkey,
    new_sol_price: u64,
) -> ProgramResult
```

Calculate Token Price

```rust
pub fn calculate_token_price(
    program_id: &Pubkey,
) -> Result<u64, ProgramError>
```

Purchase Tokens

```rust
pub fn purchase_tokens(
    program_id: &Pubkey,
    buyer: &Pubkey,
    amount: u64,
    sol_payment: u64,
) -> ProgramResult
```

Advance Stage

```rust
pub fn advance_stage(
    program_id: &Pubkey,
    authority: &Pubkey,
) -> ProgramResult
```

End Presale

```rust
pub fn end_presale(
    program_id: &Pubkey,
    authority: &Pubkey,
) -> ProgramResult
```

Presale Rules Implementation

Dynamic Pricing Logic

```rust
fn calculate_dynamic_price(
    base_price: u64,
    months_since_start: u64,
    initial_supply: u64,
    remaining_supply: u64,
) -> u64 {
    // Apply 15% monthly increase
    let mut price = base_price;
    for _ in 0..months_since_start {
        price = price.saturating_mul(115).saturating_div(100);
    }

    // Calculate supply decrease percentage
    let supply_decrease = if initial_supply > 0 {
        let sold = initial_supply.saturating_sub(remaining_supply);
        (sold as f64 / initial_supply as f64) * 100.0
    } else {
        0.0
    };

    // Apply 2% price increase for every 1% supply decrease
    let supply_factor = 1.0 + (supply_decrease * 0.02);
    let final_price = (price as f64 * supply_factor) as u64;

    final_price
}
```

Purchase Limit Enforcement

```rust
fn enforce_purchase_limit(
    state: &PresaleState,
    buyer: &Pubkey,
    amount: u64,
) -> ProgramResult {
    let current_purchase = state.purchases.get(buyer).unwrap_or(&0);
    let total_purchase = current_purchase.saturating_add(amount);

    // During presale, limit to 100 tokens per wallet
    if state.is_active && total_purchase > 100_000_000 { // 100 tokens with 6 decimals
        return Err(ProgramError::InvalidArgument);
    }

    Ok(())
}
```

Security Considerations

- Implement circuit breaker pattern to pause presale in case of emergency
- Use secure price oracle for SOL price with fallback mechanism
- Ensure proper access controls for administrative functions
- Implement comprehensive input validation
- Consider rate limiting to prevent flash loan attacks

3. Staking Contract

Contract Type

- Custom Solana Program

State Variables

```
pub struct StakingState {
    // Admin authority
    pub authority: Pubkey,

    // GWIM token mint address
    pub token_mint: Pubkey,

    // DAO treasury wallet to receive penalties
    pub dao_treasury: Pubkey,

    // Total tokens staked
    pub total_staked: u64,

    // Staking pools
    pub pools: [StakingPool; 4],

    // Mapping of staker addresses to their stakes
    pub stakes: BTreeMap<Pubkey, Vec<Stake>>,
}

pub struct StakingPool {
    // Duration in months
    pub duration: u8,

    // APY in basis points (e.g., 1000 = 10%)
    pub apy: u16,

    // Total tokens staked in this pool
    pub total_staked: u64,
}
```

```rust
pub struct Stake {
    // Staker address
    pub staker: Pubkey,

    // Amount staked
    pub amount: u64,

    // Pool index
    pub pool_index: u8,

    // Start timestamp
    pub start_time: i64,

    // End timestamp
    pub end_time: i64,

    // Accumulated rewards
    pub rewards: u64,

    // Is active
    pub is_active: bool,
}
```

Key Functions

Initialize Staking

```rust
pub fn initialize_staking(
    program_id: &Pubkey,
    authority: &Pubkey,
    token_mint: &Pubkey,
    dao_treasury: &Pubkey,
    pool_durations: [u8; 4],
    pool_apys: [u16; 4],
) -> ProgramResult
```

Stake Tokens

```rust
pub fn stake_tokens(
    program_id: &Pubkey,
    staker: &Pubkey,
    amount: u64,
    pool_index: u8,
) -> ProgramResult
```

Calculate Rewards

```
pub fn calculate_rewards(
    program_id: &Pubkey,
    staker: &Pubkey,
    stake_id: u64,
) -> Result<u64, ProgramError>
```

Claim Rewards

```
pub fn claim_rewards(
    program_id: &Pubkey,
    staker: &Pubkey,
    stake_id: u64,
) -> ProgramResult
```

Unstake Tokens

```
pub fn unstake_tokens(
    program_id: &Pubkey,
    staker: &Pubkey,
    stake_id: u64,
) -> ProgramResult
```

Staking Rules Implementation

APY Calculation

```
fn calculate_apy_rewards(
    amount: u64,
    apy: u16,
    start_time: i64,
    current_time: i64,
) -> u64 {
    // Convert APY from basis points to decimal
    let apy_decimal = apy as f64 / 10000.0;

    // Calculate time elapsed in years
    let seconds_in_year = 31536000.0; // 365 days
    let time_elapsed = (current_time - start_time) as f64 / seconds_in_year;

    // Calculate rewards using compound interest formula
    let rewards = (amount as f64 * ((1.0 + apy_decimal).powf(time_elapsed) - 1.0)) as
u64;
```

```
    rewards
}
```

Early Unstaking Penalty

```
fn apply_early_unstaking_penalty(
    state: &mut StakingState,
    stake: &Stake,
    current_time: i64,
) -> (u64, u64) {
    // Check if unstaking is early
    let is_early = current_time < stake.end_time;

    // Calculate amount to return and penalty
    let total_amount = stake.amount.saturating_add(stake.rewards);

    if is_early {
        // Apply 15% penalty
        let penalty_amount = stake.amount.saturating_mul(15).saturating_div(100);
        let return_amount = total_amount.saturating_sub(penalty_amount);

        // Transfer penalty to DAO treasury
        // (implementation details omitted)

        (return_amount, penalty_amount)
    } else {
        // No penalty for regular unstaking
        (total_amount, 0)
    }
}
```

Security Considerations

- Implement reentrancy protection
- Ensure proper validation of staking periods and APY rates
- Use safe math operations to prevent overflow/underflow
- Implement comprehensive testing for reward calculations
- Consider implementing emergency unstake function with governance approval

4. DAO Governance Contract

Contract Type

- Custom Solana Program

State Variables

```rust
pub struct DAOState {
    // Admin authority (multi-sig)
    pub authority: Pubkey,

    // GWIM token mint address
    pub token_mint: Pubkey,

    // DAO treasury wallet
    pub treasury: Pubkey,

    // Staking contract address
    pub staking_contract: Pubkey,

    // Minimum tokens required to create proposal
    pub proposal_threshold: u64,

    // Minimum percentage of votes to pass proposal
    pub quorum_percentage: u8,

    // Proposals
    pub proposals: BTreeMap<u64, Proposal>,

    // Next proposal ID
    pub next_proposal_id: u64,
}

pub struct Proposal {
    // Proposal ID
    pub id: u64,

    // Proposer address
    pub proposer: Pubkey,

    // Proposal title
    pub title: String,

    // Proposal description
    pub description: String,

    // Proposal actions
    pub actions: Vec<ProposalAction>,

    // Votes for
    pub votes_for: u64,

    // Votes against
    pub votes_against: u64,

    // Start timestamp
```

```rust
    pub start_time: i64,

    // End timestamp
    pub end_time: i64,

    // Is executed
    pub is_executed: bool,

    // Is canceled
    pub is_canceled: bool,

    // Voters
    pub voters: BTreeMap<Pubkey, VoteInfo>,
}

pub struct ProposalAction {
    // Target program
    pub target: Pubkey,

    // Function to call
    pub function_id: u8,

    // Data to pass
    pub data: Vec<u8>,
}

pub struct VoteInfo {
    // Voter address
    pub voter: Pubkey,

    // Vote weight
    pub weight: u64,

    // Vote type (0 = against, 1 = for)
    pub vote_type: u8,
}
```

Key Functions

Initialize DAO

```rust
pub fn initialize_dao(
    program_id: &Pubkey,
    authority: &Pubkey,
    token_mint: &Pubkey,
    treasury: &Pubkey,
    staking_contract: &Pubkey,
    proposal_threshold: u64,
```

```
    quorum_percentage: u8,
) -> ProgramResult
```

Create Proposal

```
pub fn create_proposal(
    program_id: &Pubkey,
    proposer: &Pubkey,
    title: String,
    description: String,
    actions: Vec<ProposalAction>,
    duration: u64,
) -> ProgramResult
```

Cast Vote

```
pub fn cast_vote(
    program_id: &Pubkey,
    voter: &Pubkey,
    proposal_id: u64,
    vote_type: u8,
) -> ProgramResult
```

Calculate Voting Power

```
pub fn calculate_voting_power(
    program_id: &Pubkey,
    voter: &Pubkey,
) -> Result<u64, ProgramError>
```

Execute Proposal

```
pub fn execute_proposal(
    program_id: &Pubkey,
    authority: &Pubkey,
    proposal_id: u64,
) -> ProgramResult
```

Cancel Proposal

```
pub fn cancel_proposal(
    program_id: &Pubkey,
    authority: &Pubkey,
```

```
    proposal_id: u64,
) -> ProgramResult
```

DAO Rules Implementation

Voting Power Calculation

```
fn calculate_voting_power(
    staking_contract: &Pubkey,
    voter: &Pubkey,
) -> Result<u64, ProgramError> {
    // Query staking contract to get staked amount
    // (implementation details omitted)

    // Voting power is equal to staked amount
    Ok(staked_amount)
}
```

Proposal Execution Logic

```
fn execute_proposal_actions(
    proposal: &Proposal,
) -> ProgramResult {
    // Check if proposal passed
    let total_votes = proposal.votes_for.saturating_add(proposal.votes_against);
    let vote_percentage = if total_votes > 0 {
        (proposal.votes_for as f64 / total_votes as f64) * 100.0
    } else {
        0.0
    };

    if vote_percentage < quorum_percentage as f64 {
        return Err(ProgramError::InvalidArgument);
    }

    // Execute each action
    for action in &proposal.actions {
        // Call target program with function_id and data
        // (implementation details omitted)
    }

    Ok(())
}
```

Security Considerations

- Implement time-lock for proposal execution
- Use multi-signature for critical DAO operations
- Ensure proper validation of proposal actions
- Implement comprehensive testing for voting mechanisms
- Consider implementing emergency pause functionality

Implementation Guidelines

Development Framework

- Recommended: Anchor Framework for Solana
- Alternative: Native Solana Program Library (SPL)

Development Environment

- Solana CLI tools
- Rust compiler
- Anchor CLI (if using Anchor)
- Solana Program Library (SPL)

Testing Strategy

1. Unit tests for each contract function
2. Integration tests for contract interactions
3. Simulation tests for economic scenarios
4. Security audits before mainnet deployment

Deployment Process

1. Deploy contracts to Solana devnet
2. Verify functionality and security
3. Conduct thorough testing
4. Deploy to Solana mainnet
5. Initialize contracts with correct parameters
6. Transfer authorities to multi-sig wallets

Frontend Integration

- Use @solana/web3.js for blockchain interactions
- Use @solana/spl-token for token operations

- Implement wallet adapters for Phantom, Solflare, etc.
- Replace mock implementations in the provided UI components with actual contract calls

Conclusion

This technical specification provides a comprehensive blueprint for implementing the GWIM token ecosystem on the Solana blockchain. The contracts are designed to work together to create a complete tokenomics system with presale, staking, and governance functionality.

Implementation should be carried out by experienced Solana developers with a strong understanding of blockchain security principles. All contracts should undergo thorough testing and security audits before handling real funds on mainnet.