Suggested Improvements for Solana Wagering Smart Contract

Overview

This document provides detailed implementation suggestions for fixing the security vulnerabilities identified in the audit report.

Critical Fixes

1. Authorization System Overhaul

Current Issue

The current authorization system is insufficient and allows unauthorized access to critical functions.

Recommended Solution

Implement a comprehensive authorization system with proper signer validation:

```
// Enhanced authorization structure
#[derive(Accounts)]
#[instruction(session_id: String)]
pub struct DistributeWinnings<'info> {
    /// The game server authority that created the session
    #[account(
        mut,
        constraint = game_session.authority == game_server.key() @ WagerError::UnauthorizedDistribution,
        constraint = game_server.key() == game_session.authority @ WagerError::UnauthorizedDistribution,
    )]
    pub game_session: Account<'info, GameSession>,

    /// Game server must be a signer
    #[account(
        constraint = game_server.key() == game_session.authority @ WagerError::UnauthorizedDistribution,
    )]
    pub game_server: Signer<'info>,

    /// CHECK: Vault PDA that holds the funds
    #[account(
        mut,
        seeds = [b"vault", session_id.as_bytes()],
        bump = game_session.vault_bump,
    )]
    pub vault: AccountInfo<'info>,

    #[account(
        mut,
        associated_token::mint = TOKEN_ID,
        associated_token::authority = vault
    )]
    pub vault_token_account: Account<'info, TokenAccount>,

    pub token_program: Program<'info, Token>,
    pub associated_token_program: Program<'info, AssociatedToken>,
    pub system_program: Program<'info, System>,
}
```

Additional Security Measures

```
// Add authority validation in instruction handlers
pub fn distribute_winnings_handler<'info>(
    ctx: Context<'_, '_, 'info, 'info, DistributeWinnings<'info>>,
    session_id: String,
    winning_team: u8,
) -> Result<()> {
    let game_session = &ctx.accounts.game_session;

    // Double-check authority
    require!(
        game_session.authority == ctx.accounts.game_server.key(),
        WagerError::UnauthorizedDistribution
    );

    // Verify game server is the original creator
    require!(
        game_session.authority == ctx.accounts.game_server.key(),
        WagerError::UnauthorizedDistribution
    );

    // ... rest of function
}
```

2. Input Validation Framework

Current Issue

Multiple functions lack proper input validation, leading to potential security vulnerabilities.

Recommended Solution

Implement comprehensive input validation:

```
// Input validation utilities
pub mod validation {
    use anchor_lang::prelude::*;
    use crate::errors::WagerError;

    pub fn validate_session_id(session_id: &str) -> Result<()> {
        require!(!session_id.is_empty(), WagerError::InvalidSessionId);
        require!(session_id.len() <= 32, WagerError::InvalidSessionId);
        require!(session_id.chars().all(|c| c.is_alphanumeric() || c == '-' || c == '_'),
                 WagerError::InvalidSessionId);
        Ok(())
    }

    pub fn validate_team_number(team: u8) -> Result<()> {
        require!(team == 0 || team == 1, WagerError::InvalidTeamSelection);
        Ok(())
    }

    pub fn validate_bet_amount(amount: u64) -> Result<()> {
        require!(amount > 0, WagerError::InvalidBetAmount);
        require!(amount <= 1_000_000_000_000, WagerError::InvalidBetAmount); // Max 1000 tokens
        Ok(())
    }

    pub fn validate_player_address(player: &Pubkey) -> Result<()> {
        require!(*player != Pubkey::default(), WagerError::InvalidPlayer);
        Ok(())
    }
}

// Enhanced instruction handlers with validation
pub fn join_user_handler(ctx: Context<JoinUser>, session_id: String, team: u8) -> Result<()> {
    // Input validation
    validation::validate_session_id(&session_id)?;
    validation::validate_team_number(team)?;

    let game_session = &mut ctx.accounts.game_session;

    // Validate game status
    require!(
        game_session.status == GameStatus::WaitingForPlayers,
        WagerError::InvalidGameState
    );

    // ... rest of function
}
```

3. Arithmetic Safety

Current Issue

Potential integer overflow in calculations, especially in payout distributions.

Recommended Solution

Implement safe arithmetic operations:

```rust
use anchor_lang::prelude::*;

// Safe arithmetic utilities
pub mod safe_math {
    use anchor_lang::prelude::*;
    use crate::errors::WagerError;

    pub fn safe_multiply(a: u64, b: u64) -> Result<u64> {
        a.checked_mul(b).ok_or(error!(WagerError::ArithmeticOverflow))
    }

    pub fn safe_divide(a: u64, b: u64) -> Result<u64> {
        require!(b > 0, WagerError::ArithmeticError);
        a.checked_div(b).ok_or(error!(WagerError::ArithmeticOverflow))
    }

    pub fn safe_add(a: u64, b: u64) -> Result<u64> {
        a.checked_add(b).ok_or(error!(WagerError::ArithmeticOverflow))
    }

    pub fn safe_subtract(a: u64, b: u64) -> Result<u64> {
        a.checked_sub(b).ok_or(error!(WagerError::ArithmeticUnderflow))
    }
}

// Enhanced payout calculation
pub fn distribute_pay_spawn_earnings<'info>(
    ctx: Context<'_, '_, 'info, 'info, DistributeWinnings<'info>>,
    session_id: String,
) -> Result<()> {
    let game_session = &ctx.accounts.game_session;

    for player in game_session.get_all_players() {
        if player == Pubkey::default() {
            continue;
        }

        let kills_and_spawns = game_session.get_kills_and_spawns(player)?;
        if kills_and_spawns == 0 {
            continue;
        }

        // Safe arithmetic for earnings calculation
        let earnings = safe_math::safe_multiply(
            kills_and_spawns as u64,
            game_session.session_bet
        )?;

        let earnings = safe_math::safe_divide(earnings, 10)?;

        // ... rest of distribution logic
    }

    Ok(())
}
```

4. Reentrancy Protection

Current Issue

No reentrancy guards on functions that modify state and transfer tokens.

Recommended Solution

Implement reentrancy protection:

```rust
// Add reentrancy guard to GameSession
#[account]
pub struct GameSession {
    // ... existing fields
    pub is_processing: bool,  // Reentrancy guard
    pub last_processed_at: i64,  // Timestamp for additional protection
}

// Reentrancy protection macro
macro_rules! reentrancy_guard {
    ($game_session:expr) => {
        require!(!$game_session.is_processing, WagerError::AlreadyProcessing);
        $game_session.is_processing = true;
    };
}

macro_rules! release_reentrancy_guard {
    ($game_session:expr) => {
        $game_session.is_processing = false;
    };
}

// Enhanced instruction handlers with reentrancy protection
pub fn distribute_winnings_handler<'info>(
    ctx: Context<'_, '_, 'info, 'info, DistributeWinnings<'info>>,
    session_id: String,
    winning_team: u8,
) -> Result<()> {
    let game_session = &mut ctx.accounts.game_session;

    // Apply reentrancy guard
    reentrancy_guard!(game_session);

    // Perform operations
    let result = distribute_all_winnings_handler_internal(ctx, session_id, winning_team);

    // Release reentrancy guard
    release_reentrancy_guard!(game_session);

    result
}
```

High Priority Fixes

5. Race Condition Prevention

Current Issue

Race conditions in team joining and state modifications.

Recommended Solution

Implement atomic operations and proper locking:

```rust
// Enhanced team joining with atomic operations
pub fn join_user_handler(ctx: Context<JoinUser>, session_id: String, team: u8) -> Result<()> {
    // ... validation code ...

    let game_session = &mut ctx.accounts.game_session;

    // Atomic check and update
    let empty_index = game_session.get_player_empty_slot(team)?;

    // Verify slot is still empty (atomic check)
    let selected_team = if team == 0 {
        &mut game_session.team_a
    } else {
        &mut game_session.team_b
    };

    require!(
        selected_team.players[empty_index] == Pubkey::default(),
        WagerError::SlotAlreadyOccupied
    );

    // Atomic update
    selected_team.players[empty_index] = ctx.accounts.user.key();
    selected_team.player_spawns[empty_index] = 10;
    selected_team.player_kills[empty_index] = 0;

    // ... rest of function
}
```

6. Enhanced Error Handling

Current Issue

Fragile error handling with string matching and insufficient error types.

Recommended Solution

Implement robust error handling:

```
#[error_code]
pub enum WagerError {
    // ... existing errors ...

    // Enhanced error types
    #[msg("Invalid session ID format")]
    InvalidSessionId,

    #[msg("Operation already in progress")]
    AlreadyProcessing,

    #[msg("Arithmetic overflow in calculation")]
    ArithmeticOverflow,

    #[msg("Arithmetic underflow in calculation")]
    ArithmeticUnderflow,

    #[msg("Invalid bet amount")]
    InvalidBetAmount,

    #[msg("Slot already occupied")]
    SlotAlreadyOccupied,

    #[msg("Invalid kill data")]
    InvalidKill,

    #[msg("Too many remaining accounts")]
    TooManyRemainingAccounts,
}

// Enhanced error checking
fn is_team_full_error(error: &Error) -> bool {
    matches!(error, Error::from(WagerError::TeamIsFull))
}
```

7. Access Control Improvements

Current Issue

Insufficient access control and validation in critical functions.

Recommended Solution

Implement comprehensive access control:

```
// Enhanced record_kill with additional validation
pub fn record_kill_handler(
    ctx: Context<RecordKill>,
    session_id: String,
    killer_team: u8,
    killer: Pubkey,
    victim_team: u8,
    victim: Pubkey,
) -> Result<()> {
    // Input validation
    validation::validate_session_id(&session_id)?;
    validation::validate_team_number(killer_team)?;
    validation::validate_team_number(victim_team)?;
    validation::validate_player_address(&killer)?;
    validation::validate_player_address(&victim)?;

    // Additional validation
    require!(killer_team != victim_team, WagerError::InvalidKill);
    require!(killer != victim, WagerError::InvalidKill);

    let game_session = &mut ctx.accounts.game_session;

    // Verify players are actually in the specified teams
    require!(
        game_session.get_player_index(killer_team, killer).is_ok(),
        WagerError::InvalidKill
    );
    require!(
        game_session.get_player_index(victim_team, victim).is_ok(),
        WagerError::InvalidKill
    );

    game_session.add_kill(killer_team, killer, victim_team, victim)?;
    Ok(())
}
```

Medium Priority Improvements

8. Event Logging System

Current Issue

Lack of comprehensive event logging for monitoring and debugging.

Recommended Solution

Implement structured event logging:

```
// Event definitions
#[event]
pub struct GameSessionCreated {
    pub session_id: String,
    pub authority: Pubkey,
    pub bet_amount: u64,
    pub game_mode: GameMode,
    pub timestamp: i64,
}

#[event]
pub struct PlayerJoined {
    pub session_id: String,
    pub player: Pubkey,
    pub team: u8,
    pub timestamp: i64,
}

#[event]
pub struct KillRecorded {
    pub session_id: String,
    pub killer: Pubkey,
    pub victim: Pubkey,
    pub killer_team: u8,
    pub victim_team: u8,
    pub timestamp: i64,
}

#[event]
pub struct WinningsDistributed {
    pub session_id: String,
    pub winning_team: u8,
    pub total_amount: u64,
    pub timestamp: i64,
}

// Enhanced instruction handlers with event logging
pub fn create_game_session_handler(
    ctx: Context<CreateGameSession>,
    session_id: String,
    bet_amount: u64,
    game_mode: GameMode,
) -> Result<()> {
    let clock = Clock::get()?;
    let game_session = &mut ctx.accounts.game_session;

    // ... existing logic ...

    // Emit event
    emit!(GameSessionCreated {
        session_id: session_id.clone(),
        authority: ctx.accounts.game_server.key(),
        bet_amount,
        game_mode,
        timestamp: clock.unix_timestamp,
    });

    Ok(())
}
```

9. Compute Optimization

Current Issue

Inefficient compute usage in several functions.

Recommended Solution

Optimize compute-heavy operations:

```rust
// Optimized payout distribution
pub fn distribute_all_winnings_handler<'info>(
    ctx: Context<'_, '_, 'info, 'info, DistributeWinnings<'info>>,
    session_id: String,
    winning_team: u8,
) -> Result<()> {
    let game_session = &ctx.accounts.game_session;

    // Pre-calculate values to avoid repeated computation
    let players_per_team = game_session.game_mode.players_per_team();
    let total_pot = safe_math::safe_multiply(
        game_session.session_bet,
        players_per_team as u64 * 2
    )?;

    let winning_amount = safe_math::safe_multiply(
        game_session.session_bet,
        2
    )?;

    // Batch process winners
    let winning_players = if winning_team == 0 {
        &game_session.team_a.players[0..players_per_team]
    } else {
        &game_session.team_b.players[0..players_per_team]
    };

    // Process winners in batches to avoid compute limit
    for (i, player) in winning_players.iter().enumerate() {
        if i >= ctx.remaining_accounts.len() / 2 {
            break; // Prevent out-of-bounds access
        }

        // ... transfer logic ...
    }

    Ok(())
}
```

10. Enhanced Testing Framework

Current Issue

Insufficient test coverage for edge cases and security scenarios.

Recommended Solution

Implement comprehensive testing:

```rust
// Test utilities
pub mod test_utils {
    use anchor_lang::prelude::*;

    pub fn create_test_game_session(
        program: &Program,
        session_id: String,
        bet_amount: u64,
    ) -> Result<()> {
        // Test implementation
        Ok(())
    }

    pub fn simulate_concurrent_joins(
        program: &Program,
        session_id: String,
        players: Vec<Keypair>,
    ) -> Result<()> {
        // Test implementation
        Ok(())
    }
}

// Property-based testing
#[cfg(test)]
mod property_tests {
    use super::*;
    use proptest::prelude::*;

    proptest! {
        #[test]
        fn test_session_id_validation(session_id in "[a-zA-Z0-9_-]{0,50}") {
            let result = validation::validate_session_id(&session_id);
            if session_id.is_empty() || session_id.len() > 32 {
                assert!(result.is_err());
            } else {
                assert!(result.is_ok());
            }
        }
    }
}
```

Implementation Timeline

Phase 1 (Week 1-2): Critical Fixes

- [ ] Implement authorization system overhaul
- [ ] Add comprehensive input validation
- [ ] Implement arithmetic safety
- [ ] Add reentrancy protection

Phase 2 (Week 3-4): High Priority Fixes

- [ ] Fix race conditions
- [ ] Enhance error handling
- [ ] Improve access control
- [ ] Add event logging

Phase 3 (Week 5-6): Medium Priority Improvements

- [ ] Optimize compute usage
- [ ] Enhance testing framework
- [ ] Add monitoring and alerting
- [ ] Implement additional security measures

Phase 4 (Week 7-8): Testing and Validation

- [ ] Run comprehensive security tests
- [ ] Perform integration testing
- [ ] Conduct penetration testing
- [ ] Final security review

Code Quality Standards

1. Documentation

- All public functions must have comprehensive documentation
- Include examples for complex functions
- Document all security considerations

2. Error Handling

- Use specific error types instead of generic errors
- Provide meaningful error messages
- Implement proper error propagation

3. Testing

- Achieve 90%+ test coverage
- Include property-based testing
- Implement fuzz testing for critical functions

4. Security

- Follow Solana security best practices
- Implement defense in depth
- Regular security audits

Monitoring and Maintenance

1. Continuous Monitoring

- Set up alerts for failed transactions
- Monitor for suspicious patterns
- Track compute usage and costs

2. Regular Updates

- Keep dependencies updated
- Monitor for new security vulnerabilities
- Implement security patches promptly

3. Documentation Maintenance

- Keep documentation up to date
- Document all changes and fixes
- Maintain security guidelines

Note: These improvements should be implemented incrementally with thorough testing at each stage. All changes should be reviewed by security experts before deployment to mainnet.