Solana Wagering Smart Contract Security Audit Report

Executive Summary

This audit report covers the security analysis of a Solana-based wagering smart contract system for a competitive FPS game with Win-2-Earn mechanics. The system allows players to stake tokens in matches where the winner takes all, with support for both "Winner Takes All" and "Pay to Spawn" game modes.

Audit Scope: Complete analysis of Rust smart contracts (~500 lines of code) covering player matching, escrow of funds, payouts, and anti-abuse mechanics.

Overall Assessment: The codebase demonstrates good architectural design but contains several critical security vulnerabilities that must be addressed before mainnet deployment.

System Architecture Overview

Core Components

- **Game Session Management**: Creation and management of game sessions with configurable team sizes (1v1, 3v3, 5v5)
- **Token Escrow System**: SPL token vault for holding player stakes during matches
- **Payout Distribution**: Automated distribution of winnings to winning teams
- **Pay-to-Spawn Mechanics**: Additional token payments for respawning during matches
- **Refund System**: Emergency refund mechanism for incomplete games

Key Data Structures

- `GameSession`: Main account storing game state, team information, and player data
- `Team`: Player arrays with bet amounts, spawn counts, and kill counts
- `GameMode`: Enum defining team sizes and game types
- `GameStatus`: State machine for game progression

Critical Security Findings

🔴 CRITICAL - Unauthorized Fund Access

Issue: The `distribute_winnings` and `refund_wager` functions lack proper authorization checks for the game server authority.

Location: `distribute_winnings.rs:107-111`, `refund_wager.rs:104-105`

Impact: Any account can call these functions by providing a valid game server public key, potentially draining all funds from game sessions.

Code Reference:

```
// Missing proper authority validation
require!(
    game_session.authority == ctx.accounts.game_server.key(),
    WagerError::UnauthorizedDistribution
);
```

Recommendation: Implement proper signer validation and authority checks.

🔴 CRITICAL - Integer Overflow in Payout Calculations

Issue: Potential integer overflow in payout calculations, especially in `distribute_pay_spawn_earnings`.

Location: `distribute_winnings.rs:39`

Impact: Could lead to incorrect payout amounts or arithmetic panics.

Code Reference:

```
let earnings = kills_and_spawns as u64 * game_session.session_bet / 10;
```

Recommendation: Use checked arithmetic operations and validate input ranges.

🔴 CRITICAL - Insufficient Input Validation

Issue: Multiple functions lack proper validation of critical inputs.

Locations:

- `join_user.rs:6` - No validation of session_id format
- `pay_to_spawn.rs:6` - No validation of session_id format
- `record_kill.rs:4-11` - No validation of player addresses

Impact: Potential for invalid data processing and state corruption.

Recommendation: Implement comprehensive input validation for all public functions.

High Severity Findings

🟠 HIGH - Race Condition in Team Joining

Issue: Multiple players can potentially join the same team slot simultaneously due to lack of atomic operations.

Location: `join_user.rs:19-52`

Impact: Could lead to players being overwritten or funds being lost.

Code Reference:

```
let empty_index = game_session.get_player_empty_slot(team)?;
// Race condition window here
selected_team.players[empty_index] = player.key();
```

Recommendation: Implement proper locking mechanisms or use atomic operations.

🟠 HIGH - Missing Reentrancy Protection

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

Locations: All instruction handlers

Impact: Potential for reentrancy attacks during token transfers.

Recommendation: Implement reentrancy guards using Anchor's built-in mechanisms.

🟠 HIGH - Inadequate Error Handling

Issue: Several functions use `unwrap()` or lack proper error handling.

Location: `state.rs:195-197`

Code Reference:

```
fn is_team_full_error(error: &Error) -> bool {
    error.to_string().contains("TeamIsFull")  // String matching is fragile
}
```

Impact: Could lead to unexpected panics or incorrect error handling.

Recommendation: Use proper error type matching and avoid string-based error detection.

Medium Severity Findings

🟡 MEDIUM - Insufficient Access Control

Issue: The `record_kill` function only validates game server authority but doesn't verify the kill is legitimate.

Location: `record_kill.rs:4-15`

Impact: Game server could record fake kills, affecting game integrity.

Recommendation: Implement additional validation or require off-chain proof of kills.

🟡 MEDIUM - Potential DoS via Large Remaining Accounts

Issue: Functions accepting `remaining_accounts` don't limit the number of accounts.

Location: `distribute_winnings.rs:21-30`

Impact: Could lead to compute limit exceeded errors or high transaction costs.

Recommendation: Implement account count limits and validation.

🟡 MEDIUM - Missing Event Logging

Issue: Critical state changes lack proper event logging for monitoring and debugging.

Impact: Difficult to track game state changes and debug issues.

Recommendation: Add comprehensive event logging for all state changes.

Low Severity Findings

🟢 LOW - Code Quality Issues

Issue: Several code quality improvements needed.

Examples:

- Inconsistent error handling patterns
- Missing documentation for complex functions
- Unused imports and variables

Recommendation: Implement code quality standards and automated linting.

🟢 LOW - Gas Optimization Opportunities

Issue: Several functions could be optimized for compute usage.

Examples:

- Redundant calculations in loops
- Inefficient account access patterns

Recommendation: Profile and optimize compute-heavy operations.

Test Coverage Analysis

Existing Test Coverage

- ☑ Basic game session creation
- ☑ User joining functionality
- ☑ Winner-takes-all payout distribution
- ☑ Pay-to-spawn mechanics
- ☑ Refund functionality

Missing Test Coverage

- ❌ Edge cases and error conditions
- ❌ Concurrent access scenarios
- ❌ Malicious input handling
- ❌ Boundary value testing
- ❌ Integration testing with real token transfers

Recommended Fixes

1. Implement Proper Authorization

```
#[derive(Accounts)]
pub struct DistributeWinnings<'info> {
    #[account(
        mut,
        constraint = game_session.authority == game_server.key() @ WagerError::UnauthorizedDistribution,
    )]
    pub game_session: Account<'info, GameSession>,

    #[account(
        constraint = game_server.key() == game_session.authority @ WagerError::UnauthorizedDistribution,
    )]
    pub game_server: Signer<'info>,
    // ... other accounts
}
```

2. Add Input Validation

```
pub fn join_user_handler(ctx: Context<JoinUser>, session_id: String, team: u8) -> Result<()> {
    // Validate session_id format
    require!(session_id.len() > 0 && session_id.len() <= 32, WagerError::InvalidSessionId);

    // Validate team number
    require!(team == 0 || team == 1, WagerError::InvalidTeamSelection);

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

3. Implement Reentrancy Protection

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

#[account]
pub struct GameSession {
    // ... existing fields
    pub is_processing: bool,  // Reentrancy guard
}

// In instruction handlers
require!(!game_session.is_processing, WagerError::AlreadyProcessing);
game_session.is_processing = true;
// ... perform operations
game_session.is_processing = false;
```

4. Add Comprehensive Error Handling

```
#[error_code]
pub enum WagerError {
    // ... existing errors
    #[msg("Invalid session ID format")]
    InvalidSessionId,
    #[msg("Operation already in progress")]
    AlreadyProcessing,
    #[msg("Arithmetic overflow in calculation")]
    ArithmeticOverflow,
}
```

Additional Security Recommendations

1. Implement Multi-Signature Authority

Consider implementing a multi-signature system for critical operations like fund distribution.

2. Add Time-based Constraints

Implement timeouts for game sessions to prevent indefinite fund locking.

3. Implement Circuit Breakers

Add mechanisms to pause the system in case of detected anomalies.

4. Enhanced Monitoring

Implement comprehensive logging and monitoring for all critical operations.

Conclusion

The wagering smart contract system shows good architectural design but requires significant security improvements before mainnet deployment. The critical findings related to authorization, input validation, and arithmetic operations must be addressed immediately.

Recommendation: Do not deploy to mainnet until all critical and high-severity issues are resolved and thoroughly tested.

Estimated Fix Time: 2-3 weeks for critical issues, 4-6 weeks for complete security hardening.

Appendix: Test Cases for Validation

Test Case 1: Authorization Bypass

```
it("Should fail when non-authority tries to distribute winnings", async () => {
  // Test with malicious game server key
  const maliciousServer = Keypair.generate();
  // Attempt to distribute winnings with wrong authority
  // Should fail with UnauthorizedDistribution error
});
```

Test Case 2: Integer Overflow

```
it("Should handle large numbers without overflow", async () => {
  // Test with maximum u64 values
  const maxBet = new BN("18446744073709551615");
  // Should not panic or produce incorrect results
});
```

Test Case 3: Race Condition

```
it("Should handle concurrent team joining", async () => {
  // Simulate multiple players joining simultaneously
  // Should not allow duplicate slot assignment
});
```

Audit Date: December 2024
Auditor: AI Security Analysis
Codebase Version: As provided in smart-contracts-refund.zip
Total Lines Analyzed: ~500 lines of Rust code