Smart Contract Audit Report

Project: PrimeSkill
Auditor: SergZen
Date: 26.09.2025
Audit Scope: Anchor Program
Codebase: https://drive.google.com/file/d/1YR2s9KgHiQMD--LmBWK_HGAo22gU1A9K/view?usp=sharing

---

Table of Contents

---

1. Executive Summary

This audit covered smart contracts for a Win-2-Earn Solana FPS game, focusing on fund escrow, player matching, and payout logic. The contracts were generally well-structured, but several vulnerabilities and edge cases were identified, which are detailed below.

---

2. Audit Methodology

- Manual code review (line-by-line)
- Static analysis (Solana-specific linters, Clippy)
- Simulation of player flow / match lifecycle
- Testing on localnet

---

3. Architecture Overview

Instructions:

- create_game_session
- join_user
- distribute_winnings
- pay_to_spawn
- record_kill
- refund_wager

- Flow: Flow
- Roles: GameServer, User

---

4. Findings Overview

ID	Title	Severity	Status
F01	Missing Validation and Unsafe Arithmetic in match victim_team	High	Unresolved
F02	Missing Uniqueness Check for Player Addition (Duplicate Players Allowed)	High	Unresolved

---

5. Detailed Findings

- **F01 — Missing Validation and Unsafe Arithmetic in match victim_team**

Severity: High
Description:
The code modifies player_spawns for the victim player based on victim_team without validating the input and without using safe arithmetic operations. Specifically, the decrement operation (-= 1) can underflow if the current value is zero. The match statement only checks for 0 and 1 teams, returning an error for other values, but does not prevent underflow in the subtraction.

Impact:
An attacker can cause an integer underflow, potentially leading to incorrect player state or game logic manipulation, which might disrupt the game flow or create inconsistencies.

Recommendation:
Add explicit validation to ensure player_spawns[victim_player_index] is greater than zero before decrementing. Use safe arithmetic operations (e.g., checked subtraction) to prevent underflow errors.

- **F02 — Missing Uniqueness Check for Player Addition (Duplicate Players Allowed)**

Severity: High
Description:
The code allows the same player address to be added to a team multiple times because there is no validation that the player is already present in the team's player list. Later game actions (e.g., resolving kills, updating spawns, selecting a participant) use a lookup that finds only the first occurrence of the player. This mismatch between insertion and action logic leads to inconsistent game state when duplicates exist.

Impact:

- A malicious actor can add the same wallet/player repeatedly to a team, creating duplicate entries.
- Subsequent game operations that act on a single found entry will ignore additional duplicates, causing incorrect spawn counts, score calculations, or action attribution.
- This can be exploited to manipulate game state, disrupt matches, grief other players, or cause denial of service in match logic.
- Depending on downstream assumptions, duplicates may also cause out-of-bounds accesses or logic errors when removing or iterating players.

Recommendation:
Enforce uniqueness at the point of player addition: check whether the player already exists in the target team and reject duplicate additions.

---

6. Code Improvement Suggestions

- **Update Outdated Anchor Dependencies**

The project is currently using an outdated version of the Anchor framework (0.30.1), which may include known issues, bugs, or lack recent security and performance improvements.

It is recommended to update the dependencies in Anchor.toml to the latest stable versions:

```
[dependencies]
anchor-lang = "0.31.1"
anchor-spl = "0.31.1"
```

Keeping dependencies up to date helps ensure compatibility with the latest tooling, improved developer experience, and better security posture.

- **Optimize GameSession Struct Definition**

a. Add #[derive(InitSpace)] Macro

To ensure accurate account space calculation and compatibility with Anchor's space-related features, it is recommended to add the #[derive(InitSpace)] macro to the GameSession struct in programs/wager-program/src/state.rs.

This macro simplifies space allocation and reduces the risk of under- or over-allocating account data.

---

b. Remove Unused Field: vault_token_bump

The vault_token_bump: u8 field in the GameSession struct is currently unused. It is recommended to remove this field to clean up the code and avoid confusion or unnecessary storage usage.

Removing unused fields improves maintainability and reduces on-chain storage costs.

- **Improvements to create_game_session.rs Instruction**

a. Use InitSpace Macro for Account Size Calculation

Currently, the account space is manually calculated using a hardcoded formula:

space = 8 + 4 + 10 + 32 + 8 + 1 + (2 * (32 * 5 + 16 * 5 + 16 * 5 + 8)) + 1 + 8 + 1 + 1 + 1

It is recommended to replace this with the InitSpace macro, previously added to the GameSession struct, for clarity and maintainability:

space = 8 + GameSession::INIT_SPACE

This approach reduces human error and ensures consistency with struct definitions.

---

b. Remove Deprecated Rent Sysvar

The following line can be safely removed as the Rent sysvar is no longer required by Anchor (since v0.25.0+):

pub rent: Sysvar<'info, Rent>,

---

c. Add associated_token::token_program Attribute

To avoid runtime errors when creating the associated token account, it is recommended to explicitly specify the token program:

```
#[account(
    init,
    payer = game_server,
    associated_token::mint = mint,
    associated_token::authority = vault,
    associated_token::token_program = token_program
)]
pub vault_token_account: Account<'info, TokenAccount>,
```

This ensures compatibility with custom or upgraded token programs if needed.

---

d. Add Validation for bet_amount > 0

The parameter bet_amount: u64 should be validated to ensure it is greater than zero. This prevents zero-value bets, which could otherwise be exploited or lead to unintended behavior:

```
require!(bet_amount > 0, WagerError::InvalidBetAmount);
```

---

e. Update Inaccurate Safety Comment

Replace the outdated comment:

```
/// CHECK: This is safe as it's just used to store SOL
```

with a more appropriate one used consistently throughout the codebase:

```
/// CHECK: Vault PDA that holds the funds
```

This improves code clarity and aligns with Anchor's linting expectations.

---

f. Use a Shorter Name for mint

To improve readability and follow common naming conventions, consider shortening the field name:

```
pub mint: Account<'info, Mint>,
```

where Mint can be imported as:

```
use anchor_spl::token::Mint;
```

This helps reduce visual clutter in the instruction context and aligns with other Anchor-based projects.

- **Improvements to distribute_winnings.rs Instruction**

a. Add associated_token::token_program Attribute

To ensure explicit configuration and compatibility with future changes to the token program, it is recommended to add the associated_token::token_program attribute to the vault_token_account:

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

This change helps prevent potential runtime errors and ensures clarity in the instruction's context.

---

b. Use game_session.session_id Instead of Separate Parameter

Currently, the instruction takes a separate session_id parameter, despite the fact that the GameSession account already contains this value internally.

Consider removing the redundant session_id argument and using game_session.session_id directly. This simplifies the instruction interface, avoids potential mismatches between inputs, and reduces redundancy:

```
let session_id = game_session.session_id;
```

Such simplifications improve maintainability and reduce the risk of logic errors due to inconsistent data sources.

- **Improvements to pay_to_spawn.rs Instruction**

a. Add associated_token::token_program Attribute

To explicitly specify the token program used when referencing the associated token account, add the associated_token::token_program attribute:

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

This ensures compatibility and avoids ambiguity in future updates or audits.

---

b. Use game_session.session_id Instead of Separate Parameter

The session_id parameter is currently passed to the instruction but is not used internally. Consider removing the parameter and instead accessing the value directly from the GameSession account:

```
let session_id = game_session.session_id;
```

This reduces redundancy and potential inconsistencies between passed-in values and stored state.

---

c. Remove Unused Account: game_server

The account game_server is declared but not referenced in the instruction logic:

```
/// CHECK: Game server authority
pub game_server: AccountInfo<'info>,
```

It should be removed to clean up the instruction context and avoid confusion.

---

d. Replace TOKEN_ID With Dynamic mint Account

To improve flexibility and maintainability, replace the hardcoded TOKEN_ID with a dynamic mint account passed into the instruction:

Add the mint account:

```
#[account(
    mut,
    address = TOKEN_ID @ WagerError::InvalidMint
)]
pub mint: Account<'info, anchor_spl::token::Mint>,
```

Then update all references to TOKEN_ID with mint.key():

```
#[account(
    mut,
    constraint = user_token_account.owner == user.key(),
    constraint = user_token_account.mint == mint.key()
)]
pub user_token_account: Account<'info, TokenAccount>,

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

This avoids reliance on hardcoded constants and enables dynamic token support.

---

e. Fix Incorrect Comment in Logic

The comment preceding the following line is misleading:

```
// Check if team is full already
let player_index = game_session.get_player_index(team, ctx.accounts.user.key())?;
```

The method get_player_index retrieves the player's index in the team, not whether the team is full. Update the comment to reflect its actual purpose:

```
// Get the player's index in the specified team
```

Clear and accurate comments are important for code readability and maintainability.

- **Improvement to record_kill.rs Instruction**

  a. Use game_session.session_id Instead of Separate Parameter

The session_id parameter is currently passed into the instruction but is unused within its logic.

It is recommended to remove the redundant session_id parameter and instead use the session_id value stored within the game_session account:

```
let session_id = game_session.session_id;
```

This simplifies the instruction interface, avoids inconsistencies, and improves code clarity.

- ## Improvements to refund_wager.rs Instruction

a. Add associated_token::token_program Attribute

To explicitly specify the token program for the associated token account and avoid ambiguity, add the associated_token::token_program attribute:

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

This enhances clarity and ensures compatibility with different token programs.

---

b. Use game_session.session_id Instead of Separate Parameter

The session_id parameter is passed into the instruction but is not used internally. It is recommended to remove this parameter and use the session_id stored in the game_session account directly:

```
let session_id = game_session.session_id;
```

This reduces redundancy and potential for inconsistent data.

- ## Improvements to state.rs

a. Use an Enum for TeamType Instead of Numeric Codes

Currently, teams are represented by numeric values (0 and 1). It is recommended to define a proper enum for TeamType to improve code readability, type safety, and maintainability:

```
#[derive(Clone, Copy, PartialEq, Eq)]
pub enum TeamType {
    TeamA,
    TeamB,
}
```

Using enums avoids magic numbers and makes the code self-documenting.

---

b. Add a Function to Determine Player's Team

Implement a helper function to determine which team a given player belongs to. This abstracts the logic and avoids repeated code

This improves modularity and code clarity.

---

c. Optimize check_all_filled Using Player's Index

Currently, check_all_filled may perform unnecessary checks over all slots. Optimize this function by using the index of the newly added player to determine if the team is full, reducing computation and improving efficiency.

For example, if the added player's index reaches the maximum team size minus one, the team can be considered full without scanning all players.

This optimization improves performance, especially for larger teams.

---

7. Severity Rating Legend

Severity	Description
High	Critical flaw that may lead to loss of funds or core logic failure
Medium	Logic issue that may cause errors or inefficiencies
Low	Minor issue or best practice deviation
Info	Informational / style / non-breaking issues

---

8. Disclaimer

This audit does not guarantee the absolute security of the code. It reflects the findings and recommendations based on the provided codebase at the time of review. Further updates or external integrations may introduce new vulnerabilities.