External Audit Report — Wagering SmartContract (Solana)

Repositories/Sources: ZIP attachments and: https://github.com/Ethaga/Solana-Smart-Contract/tree/main/wagering-contract/smart-contracts-refund

Scope: smart contract audit wager-program (Rust / Anchor) — program modules in programs/wager-program/src (~X line of code). Focus: security (vulnerability), business logic, and performance/gas optimization.

Audit date: 22 Sep 2025

---

1. Executive Summary

I have reviewed the source code of the program wager-program (main file: lib.rs , folder instructions/ , state.rs , utils.rs , errors.rs). Here's a quick summary:

- A number of high-level issues were found (lack of validation authority in some flows, potential double-spend/race conditions on returns/distribution of funds), some mid-level (inconsistent account/token checks; use of AccountInfo for vaults), and some low (explicit logs, iteration optimization, use of small size type u16 that might overflow in unexpected scenarios). There are also a number of optimization recommendations (avoid building a recurring Vec<Pubkey> from
- state for each call; calculate token sharing with rounding handling; subtract recurring CPIs to program tokens). This report provides: a list of findings (with file/function locations), written test cases (unit/integration), detailed fix suggestions, and steps to reproduce/trace the findings with
- anchor test/solana-test-validator and TypeScript scripts.

---

2. Findings (grouped by priority)

High Level (Immediate action required)

1. **Distribution / refund authority not strong enough / possibility of**
2. **spoofingLocation: instructions/distribute_winnings.rs (distribute_* handler) and** instructions/refund_wager.rs (refund_wager_handler). Problem: despite the require!
3. (game_session.authority ==ctx.accounts.game_server.key(), ...) true in some pipelines, some branches (especially fallback/refund pipelines and some utilities) rely on vault as a

   AccountInfo without strong token owner/minted checks. If the account parameters are sent incorrectly by the client, the program may fail to be secure. Impact: An attacker controlling the
4. input of a malicious account or client can prevent distribution/refund or (worse) trigger a transfer to an incorrect account if the CPI to the spl-token is misconfigured.

5. **Potential double-spend / race condition on distribution vs refund**

6. Locations: distribute_winnings.rs and refund_wager.rs — both iterate players and transfer from vault to player. Problem: no flag finalized/closed is atomic before starting the CPI transfer; Two
7. calls (distribution and refund) submitted simultaneously (by the game server and other players) can decharge the vault more than once or cause a partial failure. Impact: loss of funds or inconsistent circumstances in GameSession.
8.

9. **Use of AccountInfo for vaults and assumptions about ATA**

10. Locations: refund_wager.rs and distribute_winnings.rs (vault accounts are declared as AccountInfo<'info> with seeds [b"vault", session_id.as_bytes()]). Problem: The program
11. seems to make the assumption that the vault is a PDA that holds tokens, and that there is another vault_token_account (TokenAccount type) for mint/tokens. If the token's ownership, bump, or account isn't properly secured, the transfer CPI may fail or, if misconfigured, move tokens from an account that doesn't belong to the program. Impact: operation failure or, if
12. there are other bugs, token drain.

Middle Level

1. **Not enough check on the Token / associated token accountLocation:**
2. **instructions that made the transfer (join_user.rs , pay_to_spawn.rs ,** distribute_winnings.rs , refund_wager.rs).

3. Issue: some contexts request associated_token::mint = TOKEN_ID and associated_token::authority = vault but doesn't always verify owner/pda/ Authority token account at runtime. Make sure that the TokenAccount.owner == program or that the PDA holding the token is a signer via PDA-derived authority.

4. **Reward sharing: unsecured rounding and share counting**

5. Location: distribute_winnings.rs (share calculation logic for winners or share of results). Problem: no handling of rounding and remaining tokens; also there is no zero divisor check
6. when winners_count == 0 . Impact: lost rewards (leftover in the vault) or panic due to zero.

7.

8. **Accessibility of record_kill / game-server authority functions**

9. Location: instructions/record_kill.rs and the like. Problem: game_server role seems to be
10. authorized to trigger important logic. Make sure game_server bound to a verified key and not replaceable by the player.

Low Levels

1. **Potential overflow/underflow in small types**

2. Location: state.rs (e.g. player_spawns using u16 , addition += 10u16). In large sessions or recursive counting, there is a possibility of overflow.

3. **The use of msg! () to write sensitive internal data**

4. Location: various instructions. Problem: msg! will be recorded in the public transaction log
5. (costless). Avoid writing secret data or internal keys.

6. **Efficiency: build Vec<Pubkey> and iterations**

7. Location: state.rs (get_all_players() and its caller in many places). Problem: creating a Vec
8. and copying Pubkey repeatedly on the on-chain increases computing costs. Preferences: process slot by slot directly, or keep the number of players and a more compact index.

---

3. Proof of Concept & Exploit Examples (Sketch)

Note: this is a high-level sketch — don't run the exploit on the mainnet.

Race condition (double-spend)

1. Setup: game server creates session; several players join; The PDA vault contains tokens. The
2. attacker pushes two concurrent transactions: one distribute_winnings (sent by a game server or compromise) and one refund_wager (sent by another player/party) that uses the same account parameters. Both transactions pass the initial check and begin the player's iteration;
3. No Flags

finalized which is set atomically at the beginning, both call the CPI transfer darivault_token_account to the respective recipient. Depending on the execution order in the local cluster, the amount of tokens can be paid twice. Spoofed ATA/vault

- If the client provides a different token account that appears valid but is not the correct PDA ownership, and the program does not strongly verify the owner/PDA owner, the CPI transfer may fail or interact with another account.

---

4. Improvement Recommendations (technical & priority)

High Level Fixes (must be implemented immediately)

1. **Atomic finalize flagAdd fields in GameSession, e.g. Finalized: Bool and closing:**
2. **Bool or**
   state: enum { Open, InProgress, Finalized } . Set / check atomically before performing CPI transfer so that only one flow (distribution or refund) can run.

3. Implement examination require! (!game_session.finalized, WagerError::SessionClosed) dan set game_session.finalized = true before do token transfers; use anchor_lang::solana_program::p rogram::invoke_signed with PDA authority when necessary.

4. **Stricter ownership / PDA verification**

5. Make sure the vault is a program PDA whose ownership is clear; Verify vault.owner ==&spl_token::id() if the vault is the token account, and the verification of the owner token account is the expected PDA. Use typedAccount<'info,TokenAccount>for vaulttokenaccount if
6. possible, instead of generic AccountInfo. If the vault stores SOL (system lamports), specify the appropriate account type.

7. **Add checks to the number of winners/dividers**

8. Make sure winners_count > 0 before splitting; use safe math; Deal with the remainder(remainder) with the option: (a) return the remaining to the treasury/authority; (b) Deposit the balance into the vault and deposit it into the House Fee.

9. **Batasi authority game_server**

10. Save the pubkey game_server in GameSession at the time of creation, only receive instructions from game_server . If game_server can change, add a voting or multi-sig mechanism.

Intermediate Repair

1. **Explicitly check all token accountsAll contexts that use ATA must**
2. **require a type of** Account<'info, TokenAccount> and check mint==TOKEN_ID and owner == expected_owner.

3. **Handling CPI errors & partial transfer**

4. Make sure that if a transfer fails, the program performs a logical rollback (e.g., does not mark the session as finalized). Use the try pattern or check the return result of the CPI and handle the failure safely.

5. **Improve logging & eventing**

6. Use Anchor events to allow off-chain services to search the state without relying on Msg! Excess.

Performance / Gas Optimization

1. **Reduce on-chain Vec allocationAvoid get_all_players() copying Pubkeys. If you need**
2. **an iteration, create an iterator that reads slot by slot without making a big mistake.**

3. **CPIJif batch does multiple transfers, do multiple transfers in batches as little as**
4. **possible; Consider bundling transfers to off-chain (but secure) handles if the number of recipients is large. Use sufficient integer (u64) for token amountsMake sure the token amount is u64 for match with SPL tokens; Avoid U16 for values related to**
5. **economic value.**
6.

---

5. Written Test Case (Unit/Integration) — Recommended

The following is a list of test cases that can be implemented with `anchor test` and/or scripts TypeScript (`@project-serum/anchor`). For each test: steps, inputs, expected.

Test 1 — `create_game_session` happy path

- **Objective: The session is created with the correct parameters. Step: call**
- **create_game_session with a new session_id, authority=admin. Expectation: GameSession**
- **account is created, field authority is the same, PDA vault is created with valid bumps.**

Test 2 — `join_user` up to capacity & overflow

- **Purpose: Users can join to the full; rejection after full. Step: create a 1v1 session;**
- **call join_user twice for teams A and B; call again for team A (or B) and make sure the TeamIsFull error . Expectation: the corresponding error is returned.**
-

Test 3 — `distribute_winnings` by unauthorized key

- **Objective: to ensure that only game_server /authority can trigger the**
- **distribution. Step: call distribute_winnings from another key. Expectation:**
- **the transaction failed with UnauthorizedDistribution.**

Test 4 — `refund_wager` race-sim (sequential simulation)

- **Objective: make sure there is no double-spend on refund+distribute. Step: run**
- **refund_wager, then immediately run distribute_winnings — tested on a test validator with different orders. Expectation: only one succeeds or one will fail safely; There**
- **should be no double payment.**

Test 5 — `distribute_winnings` division by zero & remainder

- **Objective: check the distribution of prizes when winners_count == 0 or the rest**
- **of the distribution. Step: create a session without winners; Call it Distribute.**
- **Expectation: the transaction fails or is handled gracefully (not panic).**

Test 6 — Token account mismatch

- **Objective: verify rejection when `vault_token_account.mint != TOKEN_ID` or** `owner != expected`. Step: call the transfer function with the wrong
- vault_token_account. Expectation: `require!` failed, the transaction aborted.
-

Test 7 — Overflow on `player_spawns`

- **Objective: ensure overflow handlingStep: set the spawn value close to u16::MAX - 5**
- **and then call the add_spawns repeat. Expectation: the transaction fails or the value**
- **saturates with the check.**

---

6. Reproduction Steps & Tracking Findings (Commands/Scripts)

Local environment (suggestions):

1. Install Anchor (v0.28+), Rust, Solana
2. CLI.Run local validator:

```
solana-test-validator -r --resetanchor test --skip-deploy || anchor build &&
anchor test
```

> Note: use `ANCHOR_WALLET` environment variables to replace keypairgame_server and admin when running tests.

Example of a TypeScript script (race reproduction)

(Sketch) — make 2 consecutive transactions: `distribute_winnings` and `refund_wager` With `Promise.all` to send almost simultaneously on test-validators. Pay attention to the sequence and results.

```
// pseudocode - tsconst tx1 = program.rpc.distributeWinnings(sessionId, { accounts: { /* …*/ },
signers: [gameServer]}); const tx2 = program.rpc.refundWager(sessionId, { accounts: { /* … */
},signers: [someUser]}); await Promise.all([tx1.catch(e=>e), tx2.catch(e=>e)]);
```

Verify the vault ATA balance after both are complete.

---

7. Improvement Checklist (Recommended Actions)

- [ ] Add finalized `enum/flag` in GameSession and check at the beginning of the function `distribute / refund`.

- [ ] Use Account<'info, TokenAccount> to vault the token account; verify the owner and mint [ ] Add saturation/overflow checks for player_spawns . [ ] Make sure all SPL transfers use the
- PDA authority invoke_signed with the correct bump. [ ] Add automated tests (anchor tests) for
- race conditions and error paths.

-

---

8. Appendix: List of reviewed key files

- programs/wager-program/src/lib.rsprograms/wager-
- program/src/state.rsprograms/wager-
- program/src/instructions/create_game_session.rsprograms/wager-
- program/src/instructions/join_user.rsprograms/wager-
- program/src/instructions/refund_wager.rsprograms/wager-
- program/src/instructions/distribute_winnings.rsprograms/wager-
- program/src/instructions/pay_to_spawn.rsprograms/wager-
- program/src/instructions/record_kill.rsprograms/wager-
- program/src/utils.rs

---

9. If you want me to continue

I have already written test cases and reproductive steps. I was also able to: - Write concrete Anchor/TypeScript tests for all of the above cases (in the form of files you can run), - Provide a Rust code patch (diff) that implements finalized flags, account token verification, and other fixes.

Mark which option you want and I'll create an additional file (test/patch) on the canvas.

Final note: this audit was created from the source code you attached. Before deploying to the mainnet, re-audit after the patch is applied and run the pentest on an isolated testnet environment.