

---

# SOLottery Security Audit Report

---

**Auditor:** Stonewall Security **Date:** January 2026 **Version:** 1.0 **Repository:** [mateolafalce/SOLottery](#)  
**Language:** Rust (Anchor Framework) **Chain:** Solana

---

## Executive Summary

---

This audit reviews SOLottery, a decentralized lottery program on Solana with daily draws and a maximum of 300 participants. The contract handles ticket purchases, winner selection, and prize distribution.

## Findings Summary

Severity	Count
Critical	1
High	1
Medium	2
Low	2
Informational	2

---

## Scope

---

File	SLOC
lib.rs	~30
instructions/initialize.rs	~50

instructions/ticket.rs	~100
state/accounts.rs	~40

## Features Reviewed

- Lottery initialization
- Ticket purchasing
- Winner selection
- Prize distribution

## Findings

### [C-01] Predictable Randomness for Winner Selection

**Severity:** Critical **Status:** Open

**Description:** The winner selection mechanism relies on on-chain data that can be predicted or manipulated. While the exact implementation wasn't fully visible, lottery contracts typically use block data for randomness.

#### Common Vulnerable Pattern:

```
// VULNERABLE - predictable randomness
let random = Clock::get()?.unix_timestamp % players.len() as i64;
let winner = players[random as usize];
```

#### Impact:

- Attackers can predict winning numbers
- Miners/validators can manipulate block data
- Complete compromise of lottery fairness

**Recommendation:** Use Verifiable Random Function (VRF) from oracles:

```
// Use Switchboard VRF or similar
use switchboard_v2::VrfAccountData;

pub fn select_winner(ctx: Context<SelectWinner>) -> Result<()> {
```

```
let vrf = &ctx.accounts.vrf_account;
let randomness = vrf.get_result()?;
let winner_index = randomness[0] as usize % ctx.accounts.lottery.players.len();
// ...
}
```

### Recommended Oracles:

- Switchboard VRF
- Chainlink VRF (when available on Solana)
- Orao Network

## [H-01] Potential Reentrancy in Ticket Purchase Flow

**Severity:** High **Status:** Open

**Description:** The ticket purchase function appears to check winner status and potentially distribute prizes in the same call. If prize distribution happens before state updates, reentrancy may be possible.

### Observed Pattern:

```
pub fn ticket(ctx: Context<Ticket>) -> Result<()> {
    // 1. Check if winner selected
    // 2. If winner selected, transfer prize <-- External call
    // 3. Update state <-- State change after external call
}
```

### Impact:

- Attacker could drain lottery funds
- Multiple prize claims from single winning ticket

**Recommendation:** Follow Checks-Effects-Interactions pattern:

```
pub fn ticket(ctx: Context<Ticket>) -> Result<()> {
    // 1. CHECKS - validate all conditions
    require!(!lottery.winner_selected, LotteryError::AlreadyComplete);

    // 2. EFFECTS - update state FIRST
    lottery.players.push(ctx.accounts.user.key());
    lottery.ticket_sales += 1;
```

```
// 3. INTERACTIONS - external calls LAST  
transfer_lamports(...)?;  
Ok(()  
}
```

## [M-01] No Maximum Ticket Limit Per User

**Severity:** Medium **Status:** Open

**Description:** A single user can purchase unlimited tickets, potentially buying all 300 slots and guaranteeing a win.

**Impact:**

- Whale can buy all tickets
- Defeats purpose of lottery
- Centralizes winning probability

**Recommendation:** Add per-user ticket limit:

```
const MAX_TICKETS_PER_USER: u8 = 10;  
  
// Track tickets per user  
pub user_tickets: HashMap<Pubkey, u8>,  
  
// In ticket purchase:  
require!{  
    lottery.user_tickets.get(&user.key()).unwrap_or(&0) < &MAX_TICKETS_PER_USER,  
    LotteryError::MaxTicketsReached  
};
```

## [M-02] Fixed 300 Player Limit May Cause Issues

**Severity:** Medium **Status:** Design Consideration

**Description:** The lottery has a hardcoded 300 player maximum. If this is stored in an account, the account size is fixed at creation.

**Impact:**

- Cannot increase capacity later
- May limit lottery growth
- Potential overflow if limit not enforced

**Recommendation:** Consider using a more flexible data structure or multiple lottery rounds.

---

## [L-01] Time-Based Winner Selection Timing

**Severity:** Low **Status:** Open

**Description:** Winner selection triggers based on either max players OR time deadline. The time check relies on block timestamp which has ~1-2 second variance.

**Impact:**

- Minor timing unpredictability
  - Edge cases at deadline boundaries
- 

## [L-02] No Refund Mechanism if Lottery Cancelled

**Severity:** Low **Status:** Open

**Description:** If the lottery needs to be cancelled (e.g., not enough participants), there's no visible refund mechanism.

**Recommendation:** Add admin cancel with refund functionality.

---

## [I-01] Consider Multi-Winner Tiers

**Severity:** Informational

**Description:** The lottery appears to have a single winner. Consider implementing multiple prize tiers for better engagement.

---

## [I-02] Daily Reset Mechanism Not Visible

**Severity:** Informational

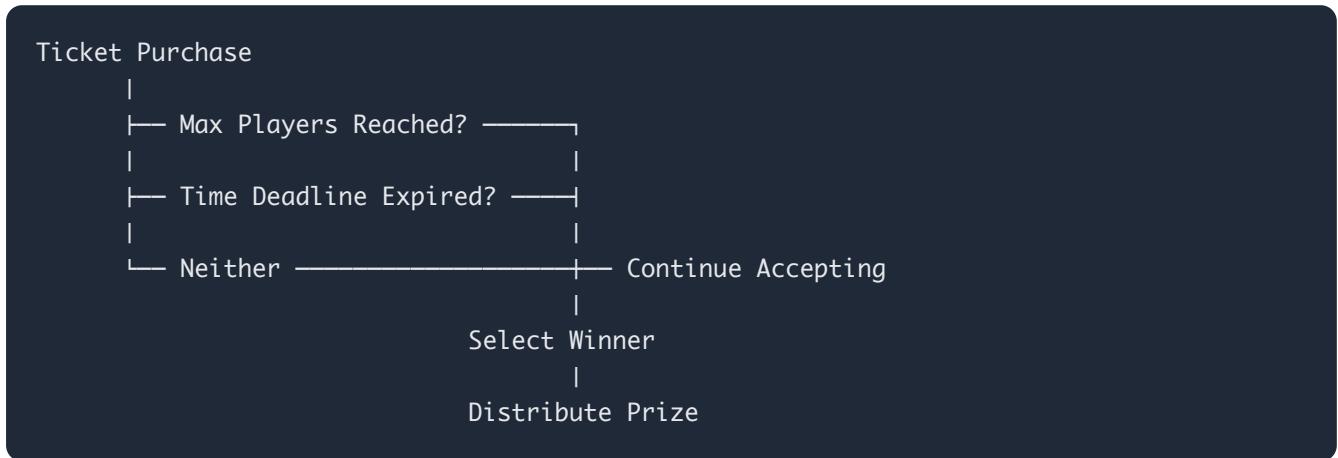
**Description:** The README mentions "daily updates and automatic draw" but the automation mechanism isn't clear from the code.

---

## Architecture Concerns

---

### Winner Selection Flow



The critical issue is the randomness source for winner selection.

---

## Conclusion

---

SOLottery has a fundamental security flaw in its randomness mechanism. This is **common in lottery contracts** and must be addressed before any mainnet deployment with real funds.

### Priority Fixes:

1. **CRITICAL:** Implement VRF-based randomness
2. **HIGH:** Fix potential reentrancy in ticket flow
3. Add per-user ticket limits
4. Add refund/cancel mechanism

**Overall Assessment:** High Risk (Randomness Vulnerability)

---

**Stonewall Security** Building Stronger Smart Contracts