SB SECURITY

Cue

Security Review

Dec 19, 2025

Contents

1. About SBSecurity

SBSecurity is a team of skilled smart contract security researchers. Based on the audits conducted and numerous vulnerabilities reported, we strive to provide the absolute best security service and client satisfaction. While it's understood that 100% security and bug-free code cannot be guaranteed by anyone, we are committed to giving our utmost to provide the best possible outcome for you and your product.

Book a Security Review with us at sbsecurity.net or reach out on Twitter @Slavcheww.

2. Disclaimer

A smart contract security review can only show the presence of vulnerabilities **but not their absence**. Audits are a time, resource, and expertise-bound effort where skilled technicians evaluate the codebase and their dependencies using various techniques to find as many flaws as possible and suggest security-related improvements. We as a company stand behind our brand and the level of service that is provided but also recommend subsequent security reviews, on-chain monitoring, and high whitehat incentivization.

3. Risk classification

	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

3.1. Impact

- **High** - leads to a significant loss of assets in the protocol or significantly harms a group of users.
- **Medium** - leads to a moderate loss of assets in the protocol or some disruption of the protocol's functionality.
- **Low** - funds are not at risk.

3.2. Likelihood

- **High** - almost **certain** to happen, easy to perform, or highly incentivized.
- **Medium** - only **conditionally possible**, but still relatively likely.
- **Low** - requires specific state or **little-to-no incentive**.

3.3. Action required for severity levels

- High - **Must** fix (before deployment if not already deployed).
- Medium - **Should** fix.
- Low - **Could** fix.

4. Executive Summary

A time-boxed security review of the **gossipgirlonchain/cue-solana** repository was conducted by **SBSecurity**. The review was performed by a team of 3 security researchers, who identified **7 issues** in total.

Overview

Project	Cue
Commit Hash	f9bdd411ab8bbecff6c805f0427f7f842b9f73b2
Resolution	e24626b1c6b91e55706c37401aad4132f888e4e5
Timeline	December 16, 2025 - December 18, 2025

Scope

lib.rs

Post Audit Condition

The codebase is well-structured and follows solid development practices. 2 Critical severity issues were found, but both required only one-line fixes. Additional audits would be a bonus, but are not mandatory.

5. Findings

Issues Found

Critical Risk	2
High Risk	0
Medium Risk	1
Low/Info Risk	4

ID	Title	Severity	Resolution
[C-01]	users can bypass the late bid cap	Critical	**Fixed**
[C-02]	bet token spoofing via unpinned Mint	Critical	**Fixed**
[M-01]	slippage protection miscalculates expected payout for time-weighted shares	Medium	**Fixed**
[L-01]	time-weighted multiplier decay clipped by trading buffer reducing penalty for late bets	Low	**Fixed**
[L-02]	Late bet cap inconsistency in low-liquidity markets	Low	**Fixed**
[L-03]	`final_payout` rounding up is incorrect and will black fee withdrawal and last user payout claim	Low	**Fixed**
[I-01]	insufficient test coverage	Info	**Fixed**

Critical severity

[C-01] users can bypass the late bid cap

Severity: Critical Risk

Description: `place_bet`'s late bet window cap can be bypassed by placing multiple bets under the cap. This way the cumulative amount bet will surpass the limit, allowing late users abusing this flaw to decide the resolution of the market.

```
pub fn place_bet(
    ctx: Context<PlaceBet>,
    option: u8,
    amount: u64,
    min_payout_bps: Option<u16>,
) -> Result<()> {
    let market = &mut ctx.accounts.market;
    let position = &mut ctx.accounts.position;
    let clock = Clock::get()?;
    let now = clock.unix_timestamp;

    // === GUARDS ===
    require!(market.status == MarketStatus::Open, CueError::MarketNotOpen);
    require!(option < market.num_options, CueError::InvalidOption);
    require!(amount >= MIN_BET, CueError::BetTooSmall);

    // Check trading is still open (with buffer for clock manipulation protection)
    require!(
        now < market.resolve_timestamp - TIME_BUFFER,
        CueError::TradingClosed
    );

    // Slippage protection: verify expected payout before betting
    if let Some(min_bps) = min_payout_bps {
        let total_pool: u64 = market.pools[..market.num_options as usize]
            .iter()
            .copied()
            .sum();
        let option_pool = market.pools[option as usize];

        // Calculate expected payout if this option wins (after this bet and fees)
        let new_total = total_pool.checked_add(amount).ok_or(CueError::Overflow)?;
        let new_option_pool = option_pool.checked_add(amount).ok_or(CueError::Overflow)?;

        // Calculate fee to match actual payout logic in claim_payout
        let fee = new_total//OK
            .checked_mul(market.protocol_fee_bps as u64)
            .ok_or(CueError::Overflow)?
            .checked_div(BPS_DENOMINATOR)
            .ok_or(CueError::Overflow)?;
        let distributable = new_total.checked_sub(fee).ok_or(CueError::Overflow)?;

        // Expected payout ratio in basis points (10000 = 1x, 20000 = 2x)
        let expected_payout_bps = distributable
            .checked_mul(BPS_DENOMINATOR)
            .ok_or(CueError::Overflow)?
            .checked_div(new_option_pool)
            .ok_or(CueError::Overflow)?;
```

```
        require!(
            expected_payout_bps >= min_bps as u64,
            CueError::SlippageExceeded
        );
    }

    // Late bet cap: In final 15 minutes, limit bets to 10% of total pool
    // This prevents large last-minute bets from manipulating outcomes
    let total_pool: u64 = market.pools[..market.num_options as usize]
        .iter()
        .copied()
        .sum();

    if now >= market.resolve_timestamp - LATE_BET_WINDOW_SECS {
        let cap = total_pool
            .checked_mul(LATE_BET_CAP_BPS)
            .ok_or(CueError::Overflow)?
            .checked_div(BPS_DENOMINATOR)
            .ok_or(CueError::Overflow)?;
        // Allow at least MIN_BET even if pool is tiny
        let cap = cap.max(MIN_BET);
        require!(amount <= cap, CueError::LateBetCapExceeded);
    }
```

As we see, the check is based on the amount argument, not the total cumulative amount of user.

Impact: Unfair late window market resolution, allowing users to influence market's outcome, with max bid cap bypass.

Recommendation: Check the cap against the cumulative bet amount of the users.

Resolution: Fixed

[C-02] bet token spoofing via unpinned Mint

Severity: Critical Risk

Description: `place_bet` accepts arbitrary SPL tokens because the USDC mint is not pinned or validated.

While the SPL Token program is enforced, the **mint itself is user-controlled**, allowing attackers to place bets using a fake token instead of real USDC.

```
pub struct PlaceBet<'info> {
    #[account(
        mut,
        associated_token::mint = usdc_mint,
        associated_token::authority = market,
    )]
    pub vault: Account<'info, TokenAccount>,

    #[account(
        mut,
        associated_token::mint = usdc_mint,
        associated_token::authority = user,
    )]
    pub user_usdc: Account<'info, TokenAccount>,

    pub usdc_mint: Account<'info, Mint>,
    pub token_program: Program<'info, Token>,
}
```

Since `usdc_mint` is not validated against the official USDC mint, an attacker can:

- Deploy a fake SPL token mint
- Create valid associated token accounts for the fake mint
- Call `place_bet` using fake tokens
- Inflate `market.pools` and influence market resolution without providing real value

All account constraints and CPI transfers succeed because the SPL Token program does not enforce token value or authenticity.

Impact: Attackers can bet with worthless tokens, manipulate pool balances, and determine the market outcome without depositing real USDC, leading to a complete loss of market integrity.

Recommendation: Explicitly pin and validate the accepted mint in all instructions interacting with token accounts.

```
#[account(address = USDC_MINT)]
pub usdc_mint: Account<'info, Mint>,
```

Alternatively, store the mint in the `Market` account at creation and require all token accounts to match it.

Resolution: Fixed

Medium severity

[M-01] Slippage protection miscalculates expected payout for time-weighted shares

Severity: Medium Risk

Description: The slippage protection implemented in `place_bet` computes an expected payout ratio using stake-based pool ratios, while the actual payout logic in claim_payout distributes funds proportionally to shares, which are time-weighted.

As a result, the slippage check systematically overestimates expected returns for late bettors, allowing bets to pass slippage validation even when the real payout would be significantly lower than the user's specified minimum, potentially resulting in unexpected losses.

Impact: Users can lose funds despite explicitly setting slippage limits.

Proof of Concept: In `claim_payout`, payout is not stake-based. It is share-based:

```
// payout = distributable * user_shares / total_shares
// Early bettors have more shares per $ → get bigger payout!
let payout_u128 = (distributable as u128)
    .checked_mul(position.shares)
    .ok_or(CueError::Overflow)?
    .checked_div(total_shares
    .ok_or(CueError::Overflow)?;
```

Because shares are time-weighted, stake ≠ shares.

Early bettors mint shares at ~1.0x (full amount) while late bettors at 0.5x–1.0x. So `total_shares` ≤ `total_pool`, with early stakes "weighing" more.

However, in `place_bet`, slippage protection assumes a **1:1 stake-to-share relationship**:

```
// Expected payout ratio in basis points (10000 = 1x, 20000 = 2x)
 let expected_payout_bps = distributable
    .checked_mul(BPS_DENOMINATOR)
    .ok_or(CueError::Overflow)?
    .checked_div(new_option_pool)
    .ok_or(CueError::Overflow)?;
```

This is incorrect and will lead to loss of funds for late stakers.

Current protection is only estimation of the expected payout if the market was to be resolved immediately after bid.

Recommendation: Slippage protection must be based on the amount of shares the bidder will receive, not on the expected total pool rewards, which won't correctly reflect the payout of the bidder after market resolution.

Resolution: Fixed

Low severity

[L-01] Time-weighted multiplier decay clipped by trading buffer reducing penalty for late bets

Severity: Low Risk

Description: The time-weighted share multiplier is designed to decay linearly from `1.0x (10,000 basis points)` to `0.5x (5,000 basis points)` over the full 15-minute late betting window (`LATE_BET_WINDOW_SECS = 900 seconds`), incentivizing early participation by penalizing late bets with fewer shares per dollar. However, the 5-minute trading buffer (`TIME_BUFFER = 300 seconds`) halts all betting 300 seconds before the resolution timestamp, rendering the final portion of the late window inaccessible. As a result, the multiplier never decays to the documented minimum of 0.5x during actual betting, instead bottoming out at approximately 0.667x (6,667 basis points). This discrepancy arises because the effective decay window is limited to ~600 seconds, diverging from the intended full-window behavior and reducing the penalty for late bettors.

Impact:

Late bettors receive more shares per dollar than designed, leading to higher relative payouts for them if their option wins. This dilutes the rewards for early participants.

Proof of Concept: The mismatch stems from the interaction between the trading guard in place_bet and the decay logic in `time_multiplier_bps`. The guard prevents bets in the final 300 seconds, limiting the maximum elapsed time in the decay formula to ~600 seconds (out of 900), resulting in incomplete decay.

This is the trading Guard in `place_bet`:

```
let now = clock.unix_timestamp;

// Check trading is still open (with buffer for clock manipulation protection)
require!(
    now < market.resolve_timestamp - TIME_BUFFER,  // ← Bets halt at deadline - 300s
    CueError::TradingClosed
);
```

But decay logic in `time_multiplier_bps` assumes full 900s window:

```
fn time_multiplier_bps(now: i64, deadline: i64) -> Result<u64> {
    let start = deadline.checked_sub(LATE_BET_WINDOW_SECS).ok_or(CueError::Overflow)?;

    // Before late window: full multiplier
    if now <= start {
        return Ok(MULTIPLIER_SCALE_BPS);
    }

    // At or after deadline: minimum multiplier
    if now >= deadline {
        return Ok(MIN_MULTIPLIER_BPS);
    }
```

```
    // During late window: linear decay
    let elapsed = (now - start) as u64;
    let window = LATE_BET_WINDOW_SECS as u64;

    // Calculate decay amount: 0 to (MULTIPLIER_SCALE_BPS - MIN_MULTIPLIER_BPS)
@>  let decay = elapsed
        .checked_mul(MULTIPLIER_SCALE_BPS - MIN_MULTIPLIER_BPS)
        .ok_or(CueError::Overflow)?
        .checked_div(window)
        .ok_or(CueError::Overflow)?;

    Ok(MULTIPLIER_SCALE_BPS - decay)
}
```

Let's walk through what actually happens with a simple timeline, assuming the resolution timestamp is T:

1. When now is exactly T - 900 seconds (start of the 15-minute window): elapsed = 0 → full multiplier of 10,000 bps (1.0x shares per dollar). Everything as expected.

2. The latest a bet can actually succeed is around T - 301 seconds (just before the 300-second buffer kicks in). Here, elapsed is about 599 seconds → decay = floor(599 × 5,000 / 900) = 3,332 bps → multiplier drops to 10,000 - 3,332 = 6,668 bps (roughly 0.667x).

3. If someone tries to bet at T - 200 seconds (still well within the "final 15 minutes" described in the comments): the transaction reverts with TradingClosed. No bet goes through, so the deeper decay never happens.

4. Even in the absolute best case (a bet landing exactly at T - 300 seconds), elapsed maxes out at ~600 seconds → decay = floor(600 × 5,000 / 900) = 3,333 bps → multiplier bottoms at 6,667 bps.

The multiplier never reaches the designed minimum of 0.5x (5000 bps) during actual betting, due to the TIME_BUFFER (300s) closing trades 5 minutes early. Instead, the effective minimum is ~0.667x (6673 bps), meaning late bettors get more shares per dollar than intended, diluting rewards for early bettors.

Recommendation: Can increase the late betting window to 20 minutes (1,200 seconds) while keeping the 5-minute trading buffer intact.

Resolution: Fixed

[L-02] Late bet cap inconsistency in low-liquidity markets

Severity: Low Risk

Description: The late-bet cap is designed to restrict any single bet in the final 15 minutes to 10% of the current total pool, preventing large last-minute bets from manipulating outcomes. However, the implementation floors the cap at MIN_BET (1,000,000 units = 1 USDC), making the effective cap max(10% of pool, 1 USDC). In markets with total pool < 10 USDC (10,000,000 units), this override allows bets exceeding 10% of the pool, contradicting the documented invariant.

Impact: In markets with total pool < 10 USDC (10,000,000 units), the 10% rule is overridden, allowing bets significantly larger than 10% of the pool.

Proof of Concept: The late-bet cap in `place_bet` is intended to limit any single bet in the final 15 minutes to 10% of the current total pool. However, the implementation applies a floor of `MIN_BET` = 1,000,000 units (1 USDC) to the calculated cap:

```
let cap = total_pool
    .checked_mul(LATE_BET_CAP_BPS)?      // 1000
    .checked_div(BPS_DENOMINATOR)?;      // 10000

let cap = cap.max(MIN_BET);

require!(amount <= cap, LateBetCapExceeded);
```

This causes the effective cap to be:

```
cap = max(10% of pool, MIN_BET)
```

where, `MIN_BET` = 1,000,000 (1 USDC)

Recommendation:

Option A: Enforce a strict 10% cap but it makes betting impossible in very small pools.

Option B: Keep logic but update comment and docs

Resolution: Fixed

[L-03] `final_payout` rounding up will block fee withdrawal and last user payout claim

Severity: Low Risk

Description: `final_payout` being rounded to 1 if the user's payout is 0, will break the accounting, stealing the fees from the project, or preventing the last user from claiming his reward.

```
let final_payout = if payout == 0 && position.shares > 0 { 1 } else { payout };
```

This will happen because the payout is based on the holdings % of the bidder. If his payout is 0, it means his contribution is insufficient for a reward.

Impact:

- Fees won't be able to be withdrawn, because the actual balance will be less than `fees_accrued`, as they're pre-calculated when the market is resolved, which uses rounding down math.

- The last user won't be able to claim his reward, because his payout will be greater than the USDC balance.

Recommendation: Remove the `final_payout` rounding up.

Resolution: Fixed

[I-01] insufficient test coverage

Severity: Low Risk

Description: Both `withdraw_fees` and `emergency_cancel` remain untested, which fails to validate, whether the business logic of the program works according to the specifications. Furthermore, having a high test coverage is useful in scenarios, where project evolves, ensuring no regression happens.

Recommendation: Fill the missing test coverage with tests for the above-mentioned functions.

Resolution: Fixed