Pashov Audit Group

TopStrike Security Review

December 18th 2025 - December 22nd 2025

Contents

1. About Pashov Audit Group

Pashov Audit Group consists of 40+ freelance security researchers, who are well proven in the space - most have earned over $100k in public contest rewards, are multi-time champions or have truly excelled in audits with us. We only work with proven and motivated talent.

With over 300 security audits completed — uncovering and helping patch thousands of vulnerabilities — the group strives to create the absolute very best audit journey possible. While 100% security is never possible to guarantee, we do guarantee you our team's best efforts for your project.

Check out our previous work here or reach out on Twitter @pashovkrum.

2. Disclaimer

A smart contract security review can never verify the complete absence of vulnerabilities. This is a time, resource and expertise bound effort where we try to find as many vulnerabilities as possible. We can not guarantee 100% security after the review or even if the review will find any problems with your smart contracts. Subsequent security reviews, bug bounty programs and on-chain monitoring are strongly recommended.

3. Risk Classification

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

Impact

• **High** - leads to a significant material loss of assets in the protocol or significantly harms a group of users
• **Medium** - leads to a moderate material loss of assets in the protocol or moderately harms a group of users
• **Low** - leads to a minor material loss of assets in the protocol or harms a small group of users

Likelihood

• **High** - attack path is possible with reasonable assumptions that mimic on-chain conditions, and the cost of the attack is relatively low compared to the amount of funds that can be stolen or lost
• **Medium** - only a conditionally incentivized attack vector, but still relatively likely
• **Low** - has too many or too unlikely assumptions or requires a significant stake by the attacker with little or no incentive

4. About TopStrike

TopStrike lets users buy, sell, and transfer fractional “player shares” priced by a quadratic bonding curve, with an IPO-style launch window that adds special buy fees plus lot limits and rate limiting. It routes sell commissions and IPO fees to prize and protocol wallets (including optional referral payouts), supports ETH/share prize distribution.

5. Executive Summary

A time-boxed security review of the **SoccerEth/TopstrikePashovContractReview** repository was done by Pashov Audit Group, during which **t.aksoy**, **Hunter**, **ast3ros**, **BengalCatBalu** engaged to review **TopStrike**. A total of **19** issues were uncovered.

Protocol Summary

Project Name	TopStrike
Protocol Type	Bonding Curve Tokensale
Timeline	December 18th 2025 - December 22nd 2025

Review commit hash:
- e5d1cd0eca8ae3544dc3137e53883a70a4d2efb3 (SoccerEth/TopstrikePashovContractReview)

Fixes review commit hash:
- 48d8dbfcd4f989d3422fbdd9f936e3fb2470130d (SoccerEth/TopstrikePashovContractReview)

Scope

`TopStrike.sol` `TopStrike.flattened.sol`

6. Findings

Findings count

Severity	Amount
Medium	3
Low	16
Total findings	**19**

Summary of findings

ID	Title	Severity	Status
[M-01]	`MIN_NOTIONAL_WEI` would DOS users sells in an edge case	Medium	**Resolved**
[M-02]	User balances partially non-withdrawable after `LOT_SIZE_UNITS` changes	Medium	**Resolved**
[M-03]	Missing slippage protection in sell functions	Medium	**Resolved**
[L-01]	Missing post-transfer minimum notional check	Low	**Resolved**
[L-02]	Inconsistent address validation in share transfer functions	Low	**Resolved**
[L-03]	Wrong player ID in `EthPrizeAwarded` event	Low	Acknowledged
[L-04]	Users can be locked from selling if trading parameters are increased	Low	**Resolved**
[L-05]	Trading disabled in the first block after enabling when IPO is disabled	Low	**Resolved**
[L-06]	ETH sent when adding player without snipe is not refunded	Low	**Resolved**
[L-07]	`MIN_NOTIONAL_WEI` check can be bypassed with buy price approximation	Low	**Resolved**
[L-08]	Buy Price should be rounded up	Low	**Resolved**
[L-09]	UX related view functions are broken	Low	**Resolved**
[L-10]	`MIN_NOTIONAL_WEI` check provides no additional safety and overlaps existing limits	Low	**Resolved**

ID	Title	Severity	Status
[L-11]	Unbounded ETH transfers enable gas griefing and denial-of-service	Low	**Resolved**
[L-12]	`LOT_SIZE_UNITS` will cause issues to different player IDs	Low	Acknowledged
[L-13]	Malicious referrer can selectively block users from selling shares	Low	**Resolved**
[L-14]	Wallet migration blocked by pause logic and is impractical	Low	Acknowledged
[L-15]	IPO window is vulnerable to sequencer failure	Low	Acknowledged
[L-16]	`LOTS_PER_TX` limits apply per function call, not per transaction	Low	**Resolved**

Medium findings

[M-01] `MIN_NOTIONAL_WEI` would DOS users sells in an edge case

Severity

Impact: Medium

Likelihood: Medium

Description

In `_sellSharesUnits()`, there is a check for `MIN_NOTIONAL_WEI` if enabled, the check is made on buy and sell operations, but it overlook a case where the following can happen: 1. A big whale buy a very large amounts of `player x`, increasing its price. 2. Some users buys `player x` shares at the border of `MIN_NOTIONAL_WEI` value. 3. The big whale sells his shares making the share price go lower. 4. Other users can't sell now cause their shares are worth less than `MIN_NOTIONAL_WEI` users will have to buy more `player x` shares to then sell their initial shares.

Same scenario applied for `transferSharesByUnits()` and `awardSharePrizeByUnits()`

Recommendations

Efficiently implementing `MIN_NOTIONAL_WEI` per every player curve will bring more complexity, so it would be better to turn off `MIN_NOTIONAL_WEI` for sells.

[M-02] User balances partially non-withdrawable after `LOT_SIZE_UNITS` changes

Severity

Impact: Medium

Likelihood: Medium

Description

The protocol allows the contract owner to dynamically update `LOT_SIZE_UNITS` using `setTradingParameters`, with the stated goal of keeping trading accessible as prices increase.

However, **all share-changing operations** (buy, sell, transfer, prize distribution) enforce that `amountUnits` must be a multiple of the *current* `LOT_SIZE_UNITS`.

```
require(amountUnits >= LOT_SIZE_UNITS, "below min size");
require(amountUnits % LOT_SIZE_UNITS == 0, "bad step");

function setTradingParameters(
    uint256 _lotSizeUnits,
    uint256 _minTradeUnits,
    ...
) external onlyOwner {
    LOT_SIZE_UNITS = _lotSizeUnits;
}
```

This creates a divisibility issue when `LOT_SIZE_UNITS` is changed to a value that is **not a divisor of existing user balances.**

As a result, users may end up holding balances that cannot be fully sold or transferred, leaving a permanently locked remainder.

Example scenarios

LOT_SIZE increase

- Old LOT_SIZE = 1e17
- User balance = 1e17, 2e17, 3e17, 4e17
- New LOT_SIZE = 5e17

None of these users can sell or transfer any amount, since their balances are below the new minimum step.

LOT_SIZE decreases with incompatible granularity

- Old LOT_SIZE = 1e17
- User balance = 1e17
- New LOT_SIZE = 3e16

The user can only act in steps of 3e16, leaving an unwithdrawable remainder of 1e16.

When **decreasing** `LOT_SIZE_UNITS`, the owner can avoid this issue by carefully choosing a value that divides the previous lot size. While error-prone, this scenario is at least controllable.

However, once a decrease has occurred, **subsequent increases of** `LOT_SIZE_UNITS` **cannot safely account for all existing balances.** Even if the new value divides the previous lot size, it may still fail to divide user balances accumulated under intermediate configurations.

This makes upward adjustments fundamentally unsafe and creates a worst-case scenario where parts of user balances become permanently non-withdrawable.

Recommendations

Ensure that any new `LOT_SIZE_UNITS` is a strict divisor of the previous `LOT_SIZE_UNITS` and lower of all existing balances.

Alternatively, introduce a balance normalization or migration mechanism when `LOT_SIZE_UNITS` is updated.

At minimum, document this risk clearly and restrict `LOT_SIZE_UNITS` changes to values that preserve divisibility.

[M-03] Missing slippage protection in sell functions

Severity

Impact: High

Likelihood: Low

Description

The `sellSharesByLots` and `sellSharesByUnits` functions lack slippage protection, exposing users to receiving significantly less ETH than expected when selling shares.

When a user submits a sell transaction, the payout is calculated based on the current supply at execution time. If other transactions execute first (either from natural market activity or frontrunning), the payout can change. If the supply decreases before execution, the user receives less than expected. This is particularly problematic because: - In active markets with frequent trading, the share supply and prices can change substantially between transaction submission and execution. - Users selling significant positions have no way to protect against unfavorable execution prices. - Malicious actors could perform sandwich attacks (sell first and buy back after victim) to extract value from users if the gain > sell fees.

In contrast, buy functions are protected by max input through `msg.value` sending amount. Sell functions have no equivalent protection.

Recommendations

Add a `minNetPayout` parameter to both sell entrypoints and revert when `netPayout < minNetPayout`.

Low findings

[L-01] Missing post-transfer minimum notional check

The transferSharesByUnits function enforces a minimum notional check on the transferred amount. However, it does not verify the remaining balance of the sender after the transfer.

```
    function transferSharesByUnits(.)
    {
        ...
        uint256 baseCost = getBuyPriceRaw(playerId, amountUnits); // Approximate notional for
check
        require(
            MIN_NOTIONAL_WEI == 0 || baseCost >= MIN_NOTIONAL_WEI,//@audit left amount could be
less
            "below min notional"
        );
```

As a result, a sender can transfer shares in a way that leaves their remaining balance below MIN_NOTIONAL_WEI, making the leftover shares not transferable or sellable (min-notional constraints).

Enforce that sharesBalance[playerId][msg.sender] - amountUnits == 0 or the remaining balance still satisfies: >= MIN_NOTIONAL_WEI.

[L-02] Inconsistent address validation in share transfer functions

The `transferSharesByUnits` and `transferSharesByLots` functions lack address validation checks that are consistently implemented in other transfer functions throughout the contract. While `transferAllSharesOfIndividualPlayer` and `transferAllSharesInWallet` verify that the recipient address is not `address(0)` or `msg.sender`, the standard transfer functions do not perform these checks.

It's recommended to add address verification so the `to` address is not `address(0)`, `msg.sender` or `address(this)`.

[L-03] Wrong player ID in `EthPrizeAwarded` event

The `awardEthPrize` and `awardEthPrizeBatch` functions emit playerId: 0 to indicate no player association. However, `nextPlayerId` starts from 0, meaning the first player added has `playerId` 0. This creates ambiguity in off-chain indexing.

```
    function awardEthPrize(
        address winner,
        uint256 amount,
        string calldata description
    ) external payable whenNotPaused onlyEthPrizeManager nonReentrant {
        require(msg.value == amount, "ETH sent must match prize amount");
        totalEthPrizesAwarded += amount;
        (bool success, ) = payable(winner).call{value: amount}("");
        require(success, "ETH transfer failed");

        emit EthPrizeAwarded(winner, 0, amount, description);
    }
```

```
    function awardEthPrizeBatch(
        address[] calldata winners,
        uint256[] calldata amounts,
        string[] calldata descriptions
    ) external payable whenNotPaused onlyEthPrizeManager nonReentrant {
        ...
        for (uint256 i = 0; i < winners.length; i++) {
            ...
>>>         emit EthPrizeAwarded(winner, 0, amount, description);
        }
    }
```

It's recommended to start player IDs from 1 instead of 0, reserving 0 as a sentinel value for "no player".

[L-04] Users can be locked from selling if trading parameters are increased

The `_sellSharesUnits` function enforces minimum trading requirements that could trap users if parameters are increased. - If `MIN_NOTIONAL_WEI` and `LOT_SIZE_UNITS` are raised, and the user's position value falls below this threshold, they cannot exit. - If a user holds a position slightly above the minimum value, and the player's share price drops due to other sells, the total value of their position may fall below `MIN_NOTIONAL_WEI`. They also cannot sell (and even block transfer in `transferSharesByUnits`).

```
    function _sellSharesUnits(
        uint256 playerId,
        uint256 amountUnits
    ) internal nonReentrant {
        require(amountUnits > 0, "amount=0");
>>>     require(amountUnits >= LOT_SIZE_UNITS, "below min size");
        require(amountUnits % LOT_SIZE_UNITS == 0, "bad step");
        uint256 lots = amountUnits / LOT_SIZE_UNITS;
        uint256 grossPayout = calculateGrossSellPayoutByUnits(playerId, amountUnits);
        require(
>>>         MIN_NOTIONAL_WEI == 0 || grossPayout >= MIN_NOTIONAL_WEI,
            "below min notional"
        );
        // ...
    }
```

It's recommended to allow users to sell their entire balance regardless of minimum parameters.

[L-05] Trading disabled in the first block after enabling when IPO is disabled

When `IPO_FEE_DURATION` is set to `0` (IPO disabled) and trading is enabled for a player, the contract temporarily blocks all buy transactions in the same block.

If IPO is disabled, the `IPO_FEE_DURATION` is set to 0. Then, when a player trade is enabled, the `players[playerId].ipoWindowStartTimestamp = players[playerId].ipoWindowEndTimestamp = block.timestamp`.

```
    function setTradingStatus(
        uint256 playerId,
        bool enabled
    ) public playerExists(playerId) onlyTradingToggler {
        if (
            enabled &&
            !players[playerId].tradingEnabled &&
            players[playerId].ipoWindowStartTimestamp == 0
        ) {
>>>         players[playerId].ipoWindowStartTimestamp = block.timestamp;
>>>         players[playerId].ipoWindowEndTimestamp
= // = block.timestamp when IPO_FEE_DURATION is 0
                block.timestamp +
                IPO_FEE_DURATION;
            ...
    }
```

This causes `isIpoPrizeFeeActive` to return `true` for that block:

```
    function isIpoPrizeFeeActive(
        uint256 playerId
    ) public view playerExists(playerId) returns (bool) {
        return
            block.timestamp >= players[playerId].ipoWindowStartTimestamp &&
            block.timestamp <= players[playerId].ipoWindowEndTimestamp;
    }
```

When users attempt to buy, the rate limit calculation fails:

```
    function _buySharesUnits(
        uint256 playerId,
        uint256 amountUnits
    ) internal nonReentrant {
        ...
        if (
            isIpoPrizeFeeActive(playerId) &&
            !hasRole(ROLE_CARD_PRIZE_MANAGER, msg.sender)
        ) {
            ...
            uint256 start = players[playerId].ipoWindowStartTimestamp;
            uint256 allowance = (block.timestamp - start) * // = 0
                IPO_TRADES_PER_SECOND;
            require(
>>>             _ipoTradesUsed[playerId][msg.sender] + 1 <= allowance, // require(1 <= 0)
```

```
                "IPO rate limit exceeded"
            );
        ...
    }
```

This prevents any user from buying shares in the same block that trading is enabled, even though IPO is effectively disabled.

It's recommended that if `IPO_FEE_DURATION` is 0, leave timestamps at 0 to keep IPO disabled.

[L-06] ETH sent when adding player without snipe is not refunded

When calling `addPlayerAndSnipeByUnits` or `addPlayerAndSnipeByLots` with a snipe amount of 0, any ETH sent with the transaction remains stuck in the contract. The refund logic is only executed when `snipeAmountUnits > 0`, meaning the `msg.value` check and refund are skipped entirely for zero-snipe player additions.

```
    function _addPlayerAndSnipeByUnits(
        string memory name,
        uint256 snipeAmountUnits
    ) internal nonReentrant {
        ...
>>>     if (snipeAmountUnits > 0) {
            ...
            if (msg.value > cost) {
>>>             (bool success, ) = payable(msg.sender).call{
                    value: msg.value - cost
                }("");
                require(success, "Refund failed");
            }
        }
    }
```

It's recommended to move the refund outside the `snipeAmountUnits > 0` conditional.

[L-07] `MIN_NOTIONAL_WEI` check can be bypassed with buy price approximation

Several flows use `getBuyPriceRaw()` as an **approximate notional** for enforcing `MIN_NOTIONAL_WEI`. However, `getBuyPriceRaw()` represents the **buy-side** bonding curve integral (cost to mint/buy), which is typically **higher than the sell-side value** for the same `amountUnits` at a given supply.

As a result, the `MIN_NOTIONAL_WEI` restriction can be satisfied using a buy-price estimate even when the economically relevant value (e.g., sell-side payout or "realizable" value) would be below `MIN_NOTIONAL_WEI`. This weakens the intended meaning of `MIN_NOTIONAL_WEI` and makes the check inconsistent across actions.

```
// Example: awardSharePrizeBatchMultiPlayerMultiUserByLots()
uint256 baseCost = getBuyPriceRaw(playerId, amount); // buy-side approximation
require(
    MIN_NOTIONAL_WEI == 0 || baseCost >= MIN_NOTIONAL_WEI,
    "below min notional"
);
```

This is mainly a UX/parameter-consistency issue because it is unclear which value the protocol *intends* to enforce for transfers and prize distributions. Still, it is worth documenting because it can lead to unexpected behavior when `MIN_NOTIONAL_WEI` is enabled.

[L-08] Buy Price should be rounded up

The buy price calculation currently rounds **down** due to integer division. While the difference is usually only a few wei, rounding down allows users to slightly reduce total cost by splitting a larger buy into multiple smaller buys.

This is a known best-practice issue for bonding curve implementations: buy operations should round **up**, while sell operations should round **down**, to prevent micro-arbitrage and ensure users always pay at least the theoretical price.

With the current implementation, the following pattern can be cheaper: - buy `a * n` units in n transactions instead of: - buy `a * n` units in a single transaction

You can see this on this python POC

```
def buy_price_up(s, a):
    n = (s + a) ** 3 - s ** 3
    den = 3 * 32000 * 1e54
    if n * 1e18 % den == 0:
        return n * 1e18 // den
    else:
        return n * 1e18 // den + 1

def buy_price_down(s, a):
    n = (s + a) ** 3 - s ** 3
    den = 3 * 32000 * 1e54
    return n * 1e18 // den

print(buy_price_down(0, 2e18), buy_price_down(1e18, 1e18) + buy_price_down(0, 1e18))
print(buy_price_up(0, 2e18), buy_price_up(1e18, 1e18) + buy_price_up(0, 1e18))
```

Recommendation:
Round buy prices up when computing the final cost.

[L-09] UX related view functions are broken

Several `view` helpers iterate over **all players** (`0 .. nextPlayerId-1`). As `nextPlayerId` grows (e.g., many real-world football players), these calls become increasingly expensive and unreliable for RPC usage, and can fail due to gas / execution limits on nodes. MegaETH rpc call gas limit is 10,000,000 - so many of these functions will DOS in near future.

Affected view functions: - `getAllPlayers()` : iterates over all players to return arrays of ids/names/tradingEnabled flags. - `getPortfolioHoldingsInFullShares(address user)` : two-pass iteration over all players (count + populate). - `getPortfolioHoldingsInLots(address user)` : two-pass iteration over all players (count + populate).

In addition, `getUserPlayerHoldingInFullShares()` is misleading for UX because it **rounds down** by integer division, potentially hiding fractional share ownership

[L-10] `MIN_NOTIONAL_WEI` check provides no additional safety and overlaps existing limits

The `MIN_NOTIONAL_WEI` check does not meaningfully strengthen trade validation and largely overlaps with existing lot-based restrictions.

Because trades are already constrained by `LOTS_PER_TX_MIN` and `LOT_SIZE_UNITS` , users cannot submit a trade smaller than the minimum lot size. Assuming the bonding curve pricing is correct, the minimum possible cost of a single lot (at zero supply) is already bounded and non-zero (e.g., ~1.04e10 wei for one lot of 1e17 units).

As a result, `MIN_NOTIONAL_WEI` either: - duplicates the effective lower bound already enforced by lot-based checks, or - introduces additional configuration complexity without improving safety.

The check is only useful if the bonding curve pricing itself is not trusted or is expected to misbehave, which contradicts the contract's core assumptions.

Recommendation:

Either remove `MIN_NOTIONAL_WEI` to reduce unnecessary complexity, or note that it duplicates the min_lots_per_tx check when setting values, so as not to break one check with another.

[L-11] Unbounded ETH transfers enable gas griefing and denial-of-service

The contract performs ETH transfers to external addresses using low-level `.call{value: ...}("")` without specifying a gas limit. This pattern appears in multiple sensitive paths, including:

- Referral payouts during `sellSharesByUnits` .
- Prize payouts in `awardEthPrize` .
- Batch payouts in `awardEthPrizeBatch` .

Because the recipient addresses are not guaranteed to be EOAs, a malicious recipient can deploy a contract with a fallback or receive function that deliberately consumes excessive gas or reverts conditionally.

In single-recipient cases, this allows the recipient to grief the caller by forcing repeated reverts or high gas usage.

Recommendations

Consider using a pull-based payout model for referrals and prizes.

Alternatively, limit the gas forwarded to external calls or isolate failures.

[L-12] `LOT_SIZE_UNITS` will cause issues to different player IDs

Description

`TopStrike` Contract uses a global `LOT_SIZE_UNITS` for every playerId when checking during users buy and sell, the value is by default 0.1e18 for buys/sells

```
        require(amountUnits >= LOT_SIZE_UNITS, "below min size");
```

The problem here is that different playerIds with different curves will have different share prices, this will create instances where: 1. Shares are very expensive for `player A` and it's hard for normal users to afford `LOT_SIZE_UNITS` amount. 2. Shares are very cheap for `player B` that `LOT_SIZE_UNITS` is very small to prevent wash trading.

So one value for `LOT_SIZE_UNITS` for different players will either cause issues or will be inefficient.

Recommendations

Implement a per playerId `LOT_SIZE_UNITS`.

[L-13] Malicious referrer can selectively block users from selling shares

Description

When selling shares, the contract sends a 1% referral fee directly to the referrer's address via `call`. If this transfer fails, the entire transaction reverts:

```
    function _sellSharesUnits(
        uint256 playerId,
        uint256 amountUnits
    ) internal nonReentrant {
        ...
        if (toReferrer > 0) {
>>>         (bool s3, ) = payable(referrer).call{value: toReferrer}("");
            require(s3, "Referral payout failed");
            emit ReferralFeePaid(referrer, msg.sender, toReferrer);
        }
        ...
    }
```

A malicious referrer using an upgradeable contract can exploit this to selectively control who or when users can sell: - Use `tx.origin` in the referrer's `receive` to decide whether to revert (select by the EOA that initiated the sell). - Configure to revert in the `receive` function in a specific period.

This enables sophisticated griefing attacks where a referrer can strategically prevent sells during price movements, forcing users to hold through unfavorable conditions. A compromised referrer address could continuously manipulate referred users until an admin manually migrates them via `updateReferrerForUsers` .

Recommendations

- Let referrer pull ETH from the contract instead of pushing OR
- If ETH send fails, fallback to wrapping into WETH and sending the ERC20.

[L-14] Wallet migration blocked by pause logic and is impractical

The protocol intends to support wallet migration and future contract upgrades via an opt-in approach, relying on users transferring their full portfolio using `transferAllSharesInWallet` , followed by off-chain indexing and coordinated migration steps.

```
    function transferAllSharesInWallet(address to) external whenNotPaused nonReentrant {
        require(to != address(0), "Cannot transfer to zero address");
        require(to != msg.sender, "Cannot transfer to self");

        uint256 transferCount = 0;
        for (uint256 playerId = 0; playerId < nextPlayerId; playerId++) {
            uint256 balance = sharesBalance[playerId][msg.sender];
            if (balance > 0) {
                sharesBalance[playerId][msg.sender] = 0;
                sharesBalance[playerId][to] += balance;
                emit SharesTransferred(msg.sender, to, playerId, balance);
                _emitTransferSharesChanged(msg.sender, to, playerId, balance, REASON_TRANSFER);
                transferCount++;
            }
        }

        require(transferCount > 0, "No shares to transfer");
    }
```

In practice, this flow is fragile and may not work as intended for several reasons:

1. **Gas inefficiency of full portfolio transfers**

`transferAllSharesInWallet` iterates over all existing players (`nextPlayerId`) and performs balance checks and transfers per player. As the number of players grows (which is expected in production), this operation becomes increasingly expensive and may exceed reasonable gas limits even on MegaETH, making wallet migration costly or impossible (in the worst scenario) for users with diversified portfolios.

1. **Trading shutdown vs. pause-state conflict**

To prepare for migration, the protocol suggests disabling trading for all players. This can be done either by: - calling `setTradingStatusBulk` across all players (gas-inefficient and expensive), or - pausing the contract.

However, `transferAllSharesInWallet` is protected by `whenNotPaused`. This means that if the contract is paused (the most gas-efficient way to halt trading), users are **unable to migrate their portfolios at all**.

1. **Compromised wallet scenario**

One of the stated goals is allowing users to migrate funds from a compromised wallet. If the contract is paused during such an incident, the affected user cannot execute `transferAllSharesInWallet`, directly contradicting this goal.

Overall, the documented upgrade and migration plan relies on mechanisms that are either gas-prohibitive at scale or logically incompatible with the pause mechanism, making the process unreliable in practice.

Recommendations

Redesign the migration flow to avoid iterating over all players in a single transaction (e.g., per-player opt-in transfers or batched migrations).

Allow portfolio transfers during a paused state, or introduce a dedicated “migration mode” that disables trading while still permitting withdrawals and transfers.

[L-15] IPO window is vulnerable to sequencer failure

The contract defines a per-player IPO window using `block.timestamp`:

- IPO starts once (first time trading is enabled).
- IPO ends at `ipoWindowStartTimestamp + IPO_FEE_DURATION`.
- After that, the IPO fee window cannot be restarted or extended (because `ipoWindowStartTimestamp` is only set once, and `ipoWindowEndTimestamp` is computed once).

```
function isIpoPrizeFeeActive(uint256 playerId) public view returns (bool) {
    return
        block.timestamp >= players[playerId].ipoWindowStartTimestamp &&
        block.timestamp <= players[playerId].ipoWindowEndTimestamp;
}
```

```
// IPO window is set once when trading is first enabled:
players[playerId].ipoWindowStartTimestamp = block.timestamp;
players[playerId].ipoWindowEndTimestamp = block.timestamp + IPO_FEE_DURATION;
```

If the MegaETT sequencer fails, users can lose effective access to the IPO window simply due to time passing. Currently, MegaETH's has a centralized sequencer, so you should be aware of potential failures.

Since the contract has no "emergency extend/reopen IPO window" mechanism, the system cannot recover the intended IPO period for a player once the window is effectively missed or invalidated.

Recommendations

Add an admin-controlled "emergency IPO management" mechanism, for example:

- Allow extending `ipoWindowEndTimestamp` for a player (bounded by a max extension).

[L-16] `LOTS_PER_TX` limits apply per function call, not per transaction

The contract enforces IPO limits using `LOTS_PER_TX` and `IPO_LOTS_PER_TX` on buy/sell operations.
The variable naming and revert message imply that these limits apply **per transaction**:

```
require(
    lots >= IPO_LOTS_PER_TX_MIN && lots <= IPO_LOTS_PER_TX_MAX,
    "IPO lots per txn out of range"
);
```

In practice, this restriction is enforced only within a single call to `_buySharesUnits` or `_sellSharesUnits`.
There is no aggregation of purchased lots across multiple function calls executed within the same transaction.

As a result, a user can call `buySharesByLots` multiple times in one transaction (e.g., via a wrapper contract) and exceed the intended "per transaction" lot limit, while still passing all checks in each individual call.

This creates ambiguity between intent and implementation: - Either the restriction is meant to be **per function call**, in which case the naming and error messages are misleading. - Or it is meant to be **per transaction**, in which case the current logic does not enforce it.

Recommendations

Clarify the intended scope of the limit and align the implementation accordingly. If the limit is meant to be per call, rename the parameters and error messages (e.g., `IPO_LOTS_PER_CALL_*`).
If the limit is meant to be per transaction, redesign the logic to account for multiple calls within the same transaction.