Cardex Security Review

Pashov Audit Group

Conducted by: Bauchibred, samuraii77, Ch_301

January 21st 2025 - January 23rd 2025

Contents

1. About Pashov Audit Group

Pashov Audit Group consists of multiple teams of some of the best smart contract security researchers in the space. Having a combined reported security vulnerabilities count of over 1000, the group strives to create the absolute very best audit journey possible - although 100% security can never be guaranteed, we do guarantee the best efforts of our experienced researchers for your blockchain protocol. Check 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. Introduction

A time-boxed security review of the **PKbubu/Cardex_Audit** repository was done by **Pashov Audit Group**, with a focus on the security aspects of the application's smart contracts implementation.

4. About Cardex

Cardex is a game that tokenizes collectible TCG cards, allowing users to buy, sell, and trade shares based on a bonding curve while using them to build decks and compete in tournaments. The CardexBeta contract, following a UUPS upgradeable pattern, manages core trading logic, utilizing a parabolic bonding curve for pricing, supporting presale and public trading stages.

5. Risk Classification

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

5.1. Impact

- High - leads to a significant material loss of assets in the protocol or significantly harms a group of users.
- Medium - only a small amount of funds can be lost (such as leakage of value) or a core functionality of the protocol is affected.
- Low - can lead to any kind of unexpected behavior with some of the protocol's functionalities that's not so critical.

5.2. 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.

5.3. Action required for severity levels

- Critical - Must fix as soon as possible (if already deployed)
- High - Must fix (before deployment if not already deployed)
- Medium - Should fix
- Low - Could fix

6. Security Assessment Summary

review commit hash - 2bb0e653e44038918e1f5894a1d3a4cf8848ac53

fixes review commit hash - 5dbabb03124bce7bd412f0ce9e3fb6864ac1779f

Scope

The following smart contracts were in scope of the audit:

- `CardexBeta`
- `ERC1967Proxy`

7. Executive Summary

Over the course of the security review, Bauchibred, samuraii77, Ch_301 engaged with Cardex to review Cardex. In this period of time a total of **5** issues were uncovered.

Protocol Summary

Protocol Name	Cardex
Repository	https://github.com/PKbubu/Cardex_Audit
Date	January 21st 2025 - January 23rd 2025
Protocol Type	Game

Findings Count

Severity	**Amount**
Medium	3
Low	2
Total Findings	**5**

Summary of Findings

ID	Title	Severity	Status
[M-01]	Presale buyers can block other whitelisted buyers from purchases	Medium	Acknowledged
[M-02]	No slippage attached to buys and sells	Medium	Resolved
[M-03]	boughtShares is wrongly initialized	Medium	Acknowledged
[L-01]	No restrictions on stage transitions	Low	Resolved
[L-02]	Wallets are not allowed to partake in presales	Low	Acknowledged

8. Findings

8.1. Medium Findings

[M-01] Presale buyers can block other whitelisted buyers from purchases

Severity

Impact: Medium

Likelihood: Medium

Description

Per the information provided by the sponsors during the audit

> In Cardex, each card will have a one-day presale stage, and after that will be changed to public The presale stage is a reward for eligible users. Cardex backend will determine eligible users. Card owner doesn't have any privileges

Now the presale mechanism uses a nonce-based signature system to authorize buyers. The preSaleSigner signs messages for whitelisted buyers that include a sequential nonce. However, the nonce is only incremented after a successful purchase, creating an issue where buyers can block others by not executing their purchase *(intentionally/unintentionally)*.

```
function buyPreSaleShares(
    uint256 id,
    uint256 amount,
    uint8 v,
    bytes32 r,
    bytes32 s
) external payable {
    require(
        cards[id].tradeStage == TradeStage.PreSale,
        "Presale only available in presale stage"
    );

    bytes32 messageHash = keccak256(
|>          abi.encode(msg.sender, id, amount, preSaleNonce)
    );
    bytes32 signedMessageHash = messageHash.toEthSignedMessageHash();
    require(
        signedMessageHash.recover(v, r, s) == preSaleSigner,
        "Invalid Signature"
    );

|>    preSaleNonce++;

    _buyShares(id, amount);
}
```

The vulnerability arises because:

1. The preSaleSigner signs messages for multiple buyers using sequential nonces (e.g., nonces 1-10)
2. Each signature is tied to a specific nonce value in the message: `abi.encode(msg.sender, id, amount, preSaleNonce)`
3. The nonce only increments after a successful purchase: `preSaleNonce++;`
4. If a buyer with a signature for nonce N never executes their purchase, all buyers with signatures for nonces > N are permanently blocked

POC:

- Alice gets signature for nonce 1
- Bob gets signature for nonce 2
- Charlie gets signature for nonce 3
- If Alice never executes her purchase, the nonce stays at 1
- Bob and Charlie can never use their signatures because they were signed with nonces 2 and 3

This allows buyers to grief other whitelisted buyers by simply not executing their purchase, permanently blocking others in the queue.

Recommendations

If the current noncing logic is to be left, then implement a time-based logic, where if a time has passed we allow purchases for the next nonce, and then users are told to make their purchases on time.

Cardex team comments

Mitigated through an off-chain mechanism (FCFS).

[M-02] No slippage attached to buys and sells

Severity

Impact: High

Likelihood: Low

Description

The `buyShares` and `sellShares` functions lack any slippage protection:

```
function buyShares(uint256 id, uint256 amount) external payable {
    require(
        cards[id].tradeStage == TradeStage.Public,
        "Public sale only available in public stage"
    );
    _buyShares(id, amount);
}

function sellShares(uint256 id, uint256 amount) external payable {
    require(id <= cardID && id > 0, "Invalid card ID");
    require(amount > 0, "Cannot sell zero shares");
    require(
        boughtShares[id] >= amount,
        "Total shares cannot be less than initial owner shares"
    );
    _sellShares(id, amount);
}
```

This can then cause a user to receive unfair value for their fund, this must not even be due to someone front running the transaction as deployment is on Abstract (making likelihood low), but just cause of the transactions to be processed, i.e

- If a user tries to **buy shares**, any other buy transactions that get processed before the user's transaction, increases the price for the user.
- If a user tries to **sell shares**, any other sell transactions that get processed before the user's transaction, decrease the price for the user.

This can lead to unfair trading conditions and financial losses for users.

While a user can limit the impact during `buyShares` by setting a reasonable `msg.value` (to avoid overpaying), the same cannot be said for `sellShares`. During a sale, the price can drop significantly due to earlier transactions, and the user cannot specify a minimum price they are willing to accept.

Recommendations

Allow users to pass in their slippage value for both buy and sell transactions.

[M-03] `boughtShares` is wrongly initialized

Severity

Impact: High

Likelihood: Low

Description

When creating an IPO card, initial shares are allocated to the initial owner:

```
card.initialOwnerShares = _card.initialOwnerShares;
..snip
            sharesBalance[cardID][_card.cardInitialOwner] = _card
                .initialOwnerShares;
```

But these shares are not accounted for in the `boughtShares` mapping. This creates an accounting mismatch that leads to permanent fund loss, causing later when users try to sell shares, the contract checks:

```
require(
    boughtShares[id] >= amount,
    "Total shares cannot be less than initial owner shares"
);
```

This creates a scenario where:

- Initial owner has X shares
- Total bought shares recorded is Y
- Actual shares in circulation is X + Y
- If initial owner sells their X shares, the last Y shares become permanently locked

For example:

1. Card created with the initial owner getting 100 shares
2. 10 Users buy 500 shares (`boughtShares[id] = 500`), *50 shares each*
3. Initial owner sells their 100 shares
4. Only 400 shares can ever be sold back (the last 100 are locked)
5. The last two users to try selling will permanently lose their funds

Coded POC

Run with `forge test --match-test testPOC_ShareAccountingMismatch -vv --via-ir`

```
function testPOC_ShareAccountingMismatch() public {
        // 1. Create a card with initial owner getting 100 shares
        vm.startPrank(cardIpoSigner);
        CardexBeta.curveFactor memory factor = CardexBeta.curveFactor({
            a: 100000000000,
            b: 9850000000000000,
            c: 5000000000000000
        });

        ipoCard(
            "Card 1",
            "pokemon",
            CardexBeta.Rarity.Common,
            "URI1",
            "URI2",
            user1, // initial owner
            100, // initial owner gets 100 shares
            block.timestamp,
            factor,
            CardexBeta.TradeStage.Public
        );
        vm.stopPrank();

        // Verify initial state
        assertEq(cardex.sharesBalance
          (1, user1), 100, "Initial owner should have 100 shares");
        assertEq(cardex.boughtShares
          (1), 0, "boughtShares not counting initial owner shares");

        // 2. Have user2 buy 100 shares
        vm.startPrank(user2);
        uint256 buyValue = cardex.getBuyPriceAfterFee(1, 100);
        cardex.buyShares{value: buyValue}(1, 100);
        vm.stopPrank();

        // Verify state after user2 buys
        assertEq(cardex.sharesBalance
          (1, user2), 100, "Buyer should have 100 shares");
        assertEq(cardex.boughtShares
          (1), 100, "Total bought shares should be 100");
        assertEq(cardex.sharesBalance
          (1, user1), 100, "Initial owner should still have 100 shares");

        // 3. owner tries to sell all their shares - this should work
        vm.startPrank(user1);
        cardex.sellShares(1, 100);
        vm.stopPrank();

        // 4. Now user2 tries to sell their shares - this will fail
        vm.startPrank(user2);
        vm.expectRevert
          ("Total shares cannot be less than initial owner shares");
        cardex.sellShares(1, 100);
        vm.stopPrank();

        // User2 is stuck with their shares because boughtShares is 0
        // but the check requires boughtShares[id] >= amount
        assertEq(cardex.boughtShares(1), 0, "boughtShares is 0");
        assertEq(cardex.sharesBalance
          (1, user2), 100, "User2 still has 100 shares but can't sell");
    }
```

Recommendations

1. Properly account for initial owner shares in `boughtShares`:

```
function ipoCard(...) external {
    // ... existing code ...
    sharesBalance[cardID][cardInitialOwner] = initialOwnerShares;
    boughtShares[cardID] = initialOwnerShares;  // Add this line
    // ... rest of the function
}
```

Cardex team comments

This is a design mechanism and will shown in Cardex docs.

8.2. Low Findings

[L-01] No restrictions on stage transitions

The contracts do not guarantee that stages follow the same order. E.g. cards can be reverted from public to presale stage.

```
function setTradeStage(uint256 id, TradeStage tradeStage) public {
    require(
        msg.sender == cardIpoSigner,
        "Only cardIpoSigner can set trade stage"
    );
    require(id <= cardID && id > 0, "Invalid card ID");
    cards[id].tradeStage = tradeStage;  // No validation on stage transition

    emit StageUpdated(id, tradeStage);
}
```

Recommendations:

- Prevent reverting from the public to the presale stage
- Add proper stage transition validation

[L-02] Wallets are not allowed to partake in presales

Cardex includes a logic that allows for both public and presale trading, with public trading anyone can access it and we just provide `msg.value` for the amout of shares to be bought.

During presale, however, we need to check and verify the signature of the buyer, to ensure it's being signed by the `presaleSigner`:

link

```
function buyPreSaleShares(
..
    ) external payable {
// ..snip
        bytes32 messageHash = keccak256(
            abi.encode(msg.sender, id, amount, preSaleNonce)
        );
        bytes32 signedMessageHash = messageHash.toEthSignedMessageHash();
        require(
            signedMessageHash.recover(v, r, s) == preSaleSigner,
            "Invalid Signature"
        );

        preSaleNonce++;

        _buyShares(id, amount);
    }
```

The issue is that we use the query `msg.sender` when creating the messagerHash. However, most wallets/contracts implement the logic of meta transactions, in which case querying `msg.sender` is wrong. We need to query `_msg.sender()` instead in order to access the real sender of the transaction. In this case, if the wallet attempts to participate in the presale after receiving a valid signature, it can't.

Support meta transactions and query `_msg.sender()` which should retunr the real `msg.sender()` if it's an EOA.