Curio Security Review

Pashov Audit Group

Conducted by: peanuts, ubermensch, rvierdiiev

April 15th 2024 - April 16th 2024

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 **curio-research/flagship-contracts** repository was done by **Pashov Audit Group**, with a focus on the security aspects of the application's smart contracts implementation.

4. About Curio

Curio Game smart contract allows the creation of games together with settling their results. Games can be ended and deposited amounts will be distributed.

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 - 19d6cd40f8a28ec842be9a52c91a1fad76d01df2

fixes review commit hash - 25a268fd7a5ecf0dc576f3b1a165fead72ceafa2

Scope

The following smart contracts were in scope of the audit:

- `Game`

7. Executive Summary

Over the course of the security review, peanuts, ubermensch, rvierdiiev engaged with Curio to review Curio. In this period of time a total of **15** issues were uncovered.

Protocol Summary

Protocol Name	Curio
Repository	https://github.com/curio-research/flagship-contracts
Date	April 15th 2024 - April 16th 2024
Protocol Type	Onchain crypto games

Findings Count

Severity	**Amount**
Critical	3
High	1
Medium	1
Low	10
Total Findings	**15**

Summary of Findings

ID	Title	Severity	Status
[C-01]	Game fee handling is incorrect	Critical	Resolved
[C-02]	Winner's payout is incorrectly calculated	Critical	Resolved
[C-03]	Incorrect logical check in endGame function	Critical	Resolved
[H-01]	Game fee locked in earlyEndGame	High	Resolved
[M-01]	Inaccessible pause and unpause functions	Medium	Resolved
[L-01]	earlyEndGame can be called several times for the same user	Low	Resolved
[L-02]	updateGameTakeRate will affect current games	Low	Resolved
[L-03]	The Game should not be started with an empty gameId	Low	Resolved
[L-04]	Slashing amount can be bigger than minStake	Low	Acknowledged
[L-05]	Default Game status set as "InProgress"	Low	Resolved
[L-06]	Missing Game status verification in endGame function	Low	Resolved
[L-07]	Incomplete initialization in init function	Low	Resolved
[L-08]	Lack of player verification in earlyEndGame function	Low	Resolved
[L-09]	updateGameTakeRate() does not have	Low	Resolved

	an upper bound		
[L-10]	Ensure that games is not overriden when starting a new game	Low	Resolved

8. Findings

8.1. Critical Findings

[C-01] Game fee handling is incorrect

Severity

Impact: High

Likelihood: High

Description

`endGame` function is called by the admin to finish the game. Admin provides a list of participants and their payouts and function loops through all of them. In case if user wins, then the game takes a fee from his reward. This fee is accumulated in the `gameFee` variable. It is increased with each new user and the real user's win is an `unsignedWinLossAmount - gameFee`.

```
uint256 gameTakeRate = takeRate[currentSession.gameType];
                gameFee +=
                  (playerPayout.unsignedWinLossAmount * gameTakeRate) / 100;


                                  uint256 adjustedPlayerPayout = playerPayout.unsignedW
```

`gameFee` is the variable that is used for the whole loop and is increased on each cycle. It is not reset for each new cycle. As a result `adjustedPlayerPayout` becomes incorrect and it may revert, when `gameFee > unsignedWinLossAmount`. Then the game will not be finalized and users will not be able to get back their funds.

Also, it is possible that the function will not revert, but `gameFee` and `adjustedPlayerPayout` will be wrong, which means incorrect payout distribution.

Recommendations

First of all, you need to pay gameFee after all loop cycles. Also you need the next change:

```
uint256 gameTakeRate = takeRate[currentSession.gameType];
                uint256 userFee =
                  (playerPayout.unsignedWinLossAmount * gameTakeRate) / 100;
                gameFee += userFee;


                                 uint256 adjustedPlayerPayout = playerPayout.unsignedW
```

[C-02] Winner's payout is incorrectly calculated

Severity

Impact: High

Likelihood: High

Description

When a winner wins money, the function incorrectly calculates the payout for the winner, resulting in much more payout for the winner than intended.

When the admin calls `endGame()`, if the player wins the game, his total withdraw amount will be updated with his payout.

```
if (playerPayout.payout > playerPayout.stake) {
                // Player won money case

                // update player revenue/in-game balance
>               balance[player] += playerPayout.payout;
            } else {
```

However, this addition `balance[player] += playerPayout.payout;` adds the incorrect amount, resulting in more payout than intended.

For example, there are two players, A and B. Both players stake 1 ether each. At the end of the game, player A wins 0.5 ether and player B loses 0.5 ether. Payout refers to how much money the player has at the end.

For player A, his `stake` is 1 ether and his `payout` is 1.5 ether. For player B, his `stake` is 1 ether and his `payout` is 0.5 ether.

Discounting game fees, following the `endGame()` function, player A's balance would be 1 + 1.5 ether which is 2.5 ether. This means that he can withdraw 2.5 ether.

In actuality, his balance should only 1.5 ether. If he calls `withdraw()` with the current calculation, he can get more ether than intended at the expense of other winners and losers.

Recommendations

Update this `balance[player] += playerPayout.payout;` addition:

```
if (playerPayout.payout > playerPayout.stake) {
                // Player won money case

                // update player revenue/in-game balance
>               balance[player] += (playerPayout.payout - playerPayout.stake);
            } else {
```

`playerPayout.payout - playerPayout.stake` gets the winning amount of the player. Add this to the original sum that the player has. This way, following the example given above, player A's balance is now 1 ether + (1.5 - 1) ether which is 1.5 ether.

[C-03] Incorrect logical check in `endGame` function

Severity

Impact: High

Likelihood: High

Description

The `endGame` function in the `Game` contract fails to function correctly due to an erroneous logical check. This function, intended to handle the payouts at the end of a game, incorrectly uses `==` instead of `!=` when comparing the hash of the `gameId` provided with the `gameId` of each player. This results in the function skipping all players who are actually in the game it intends to end. Consequently, no player payouts or updates to game status are processed, and the transaction completes without performing any meaningful actions despite returning a successful execution state. This results in the players' funds staying locked in the contract as they cannot withdraw when the game status is `InProgress`.

Recommendations

Change the comparison operator in the `endGame` function from `==` to `!=` to ensure that the function processes players who are part of the specified game. This modification will correct the logic, allowing the function to execute its intended operations, such as adjusting player balances and updating game status correctly.

```
-if (keccak256(abi.encodePacked(playerGameId[player])) == keccak256
- (abi.encodePacked(gameId))) {
+if (keccak256(abi.encodePacked(playerGameId[player])) != keccak256
+ (abi.encodePacked(gameId))) {
     continue;
}
```

8.2. High Findings

[H-01] Game fee locked in `earlyEndGame`

Severity

Impact: Medium

Likelihood: High

Description

In the `earlyEndGame` function, there is an issue where the game fee deducted from a player's payout is not subsequently transferred to the treasury address. Instead, the fee is subtracted from the player's payout and remains in the contract. This results in the funds being locked within the contract without a dedicated method available to transfer these collected fees to the treasury.

Recommendations

It is crucial to implement a mechanism within the `earlyEndGame` function to ensure that the collected game fee is transferred to the treasury address after it is deducted from the player's payout. This can be achieved by adding a transfer call after the fee deduction logic:

```
function earlyEndGame(
  stringcalldatagameId,
  PlayerPayoutmemoryplayerPayout
) external onlyAdmin whenNotPaused {
    ...

    uint256 gameTakeRate = takeRate[currentSession.gameType];
    uint256 gameFee = (playerPayout.payout * gameTakeRate) / 100;
    playerPayout.payout -= gameFee; // @audit gamefee will be stuck ?

    ...

    // Deduct money from players' balance
    balance[player] -= deduction;

    // Set the player game id to empty
    playerGameId[player] = "";

+    // Deposit game fee to reward pool
+   (bool sent,) = treasury.call{value: gameFee}("");
+   if (!sent) {
+        revert ValueTransferFailed();
+   }

    emit EarlyEndGame(playerPayout);
}
```

8.3. Medium Findings

[M-01] Inaccessible pause and unpause functions

Severity

Impact: Medium

Likelihood: Medium

Description

The `Game` contract inherits from `PausableUpgradeable`, a feature designed to allow the pausing and unpausing of contract functionalities in emergency situations or for maintenance. However, the contract does not expose the `pause` and `unpause` functions to contract administrators. As a result, despite inheriting the pausability feature, administrators are unable to utilize these critical controls to pause or resume the contract's operations when needed, potentially leading to issues during periods requiring immediate intervention.

Recommendations

To leverage the full capabilities of the `PausableUpgradeable` inheritance and enhance the contract's operational security and flexibility, it is recommended to expose the `pause` and `unpause` functions in the `Game` contract. These should be accessible by the contract owner or other authorized roles, ensuring they can respond effectively to operational needs or emergencies:

```
/**
     * @dev Pauses all functions affected by `whenNotPaused`.
     */
    function pause() public onlyOwner {
        _pause();
    }

    /**
     * @dev Unpauses all functions affected by `whenNotPaused`.
     */
    function unpause() public onlyOwner {
        _unpause();
    }
```

8.4. Low Findings

[L-01] earlyEndGame can be called several times for the same user

`earlyEndGame` function can be called by the admin when the user is defeated before the game has finished. In this case user loses some amount of funds and they are deducted from his balance.

```
address player = playerPayout.playerAddress;
        if (balance[player] < playerPayout.unsignedWinLossAmount) {
            revert InsufficientBalance();
        }

        // Deduct money from players' balance
        balance[player] -= playerPayout.unsignedWinLossAmount;
```

There are 2 problems here:

- function doesn't check if the user even participates in the game provided by the admin
- function doesn't check if the user was not slashed already

Both of those cases will be fixed with the check, that the user currently participates in the provided game.

[L-02] updateGameTakeRate will affect current games

`updateGameTakeRate` function allows the owner to change the fee rate for the game type. In case there are games instances of this type currently played, then this will affect the payout of users. As a suggestion, you can store the fee rate for each game session.

[L-03] The Game should not be started with

an empty gameId

`startGame` function has `gameId` param. This game id is set for each user that participates in the game: `playerGameId[player] = gameId;`

The function doesn't have a check that provided `gameId` is not empty. In case it is, then `isPlayerInGame` functionality will not work, which will allow users to withdraw during the game and participate in several sessions.

[L-04] Slashing amount can be bigger than minStake

When a new game is started, the function checks that each user has at least `minStake` amount of funds to participate.

```
if (balance[player] < minStake) {
                revert InsufficientStake();
            }
```

When the game is finalized, the admin provides `unsignedWinLossAmount`, which is the loss amount if the user is defeated. There is no check, that this amount is not bigger than `minStake` that was used in the beginning, which means that the user can lose more than he staked into the game.

[L-05] Default Game status set as "InProgress"

The `GameStatus` enum in the contract defaults all new `GameSession` instances to `InProgress` due to its positioning as the first element in the enum. This default behavior can lead to ambiguity, as uninitialized game sessions will appear to be active. Typically, an uninitialized or newly created game session should not inherently possess an active status, as this could lead to misinterpretations of the session's actual state.

It is recommended to introduce a new enum element, such as `NotStarted` or `Uninitialized`, and place it as the first element in the `GameStatus` enum. This change will ensure that all newly instantiated game sessions default to a neutral state that clearly indicates they have not been explicitly started:

```
enum GameStatus {
    NotStarted, // Default state for new sessions
    InProgress,
    Ended
}
```

This adjustment improves the clarity and accuracy of game session statuses, aiding in the proper management and tracking of game lifecycle states within the contract.

[L-06] Missing Game status verification in `endGame` function

The `endGame` function lacks a necessary check to verify if the game is currently in progress before proceeding to end it. This oversight allows for the possibility of ending a game multiple times or ending games that were never initialized, which can lead to inconsistencies in game management and potential errors in data handling.

To address this issue, add a condition at the beginning of the `endGame` function to check the current status of the game. The function should only allow the ending process to proceed if the game is indeed in progress. If the game is not in progress, the function should revert to prevent unnecessary operations:

```
function endGame
  (string calldata gameId, PlayerPayout[] calldata playerPayouts, uint256 pot)
    external
    onlyAdmin
    whenNotPaused
{
    GameSession memory currentSession = games[gameId];

+    // Verify game is in progress before marking as ended
+    if (currentSession.status != GameStatus.InProgress) {
+        revert("Game is not in progress or already ended");
+    }

    // Mark game as ended status
    games[gameId].status = GameStatus.Ended;
}
```

[L-07] Incomplete initialization in `init` function

The `init` function of the contract does not properly initialize all inherited contracts, specifically `UUPSUpgradeable` and `PausableUpgradeable`. Currently, the function only calls `__Ownable2Step_init_unchained`, bypassing the standard initializers for the other two upgradeable contracts. While this does not pose an immediate risk, it deviates from best practices for using upgradeable contracts.

To align with best practices and ensure that all aspects of the contract are correctly initialized, modify the `init` function to include calls to `__Pausable_init` and `__UUPSUpgradeable_init` alongside the existing initialization of `Ownable2StepUpgradeable`.

[L-08] Lack of player verification in `earlyEndGame` function

The `earlyEndGame` function currently does not verify whether a player is actually part of the game before proceeding with updates to the game state and player's balance. This lack of verification could potentially allow incorrect updates if a non-participant player's data is passed to the function, albeit the likelihood and impact of such an occurrence might be lower due to trust assumptions on the game server.

To mitigate the risk of unintended state changes and enhance the robustness of the contract, it is recommended to incorporate a check that confirms the player's involvement in the game before executing any state updates or balance modifications:

```
// When a player quits or is defeated before the game ends
    function earlyEndGame(
      stringcalldatagameId,
      PlayerPayoutmemoryplayerPayout
    ) external onlyAdmin whenNotPaused {
        GameSession memory currentSession = games[gameId];
        if (currentSession.status == GameStatus.Ended) {
            revert GameEnded();
        }

        address player = playerPayout.playerAddress;

+       if (keccak256(abi.encodePacked(playerGameId[player])) != keccak256
+ (abi.encodePacked(gameId))) {
+
+             // Player is not in the game - they may have ended early and/or joined an
+           revert PlayerNotInGame();
+       }
```

[L-09] updateGameTakeRate() does not have an upper bound

`updateGameTakeRate()` calculates the percentage of `gameFee` given to the treasury after every game. This value can be updated, and in the function, `updateGameTakeRate()`, there is no limit to what the new rate can be.

```
function updateGameTakeRate
   (string calldata gameMode, uint256 _gameTakeRate) external onlyOwner {
        takeRate[gameMode] = _gameTakeRate;

        emit UpdateGameTakeRate(gameMode, _gameTakeRate);
    }
```

Cap the limit to a maximum of x% (20%) to prevent any unfortunate errors during calculation, which will happen if the rate is set >100.

```
function updateGameTakeRate
   (string calldata gameMode, uint256 _gameTakeRate) external onlyOwner {
>     require(_gameTakeRate < 20, maximum 20% fees);
        takeRate[gameMode] = _gameTakeRate;

        emit UpdateGameTakeRate(gameMode, _gameTakeRate);
    }
```

[L-10] Ensure that `games` is not overriden when starting a new game

When the admin calls `startGame()`, he picks a `gameId` with a string data type and sets it to the current `gameSession`.

```
function startGame(
>       string calldata gameId,
        uint256 minStake,
        string calldata gameType,
        address[] calldata playerAddresses
    ) external onlyAdmin whenNotPaused {
           ...
>        games[gameId] = gameSession;
```

Have sufficient checks to ensure that if the admin creates another game, the `gameId` is not reused and the `gameSession` saved in `games[gameId]` is not overridden.