Curio Security Review

Pashov Audit Group

Conducted by: dirk_y, zigtur, peanuts

July 23th 2024 - July 25th 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/duper-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 - 9f5534d88eb71d73f1007540c1a6dbbf0e83d4a8

fixes review commit hash - 9e331ee3139a3f2614fdc70460f46ef728089e96

Scope

The following smart contracts were in scope of the audit:

- `Game`

7. Executive Summary

Over the course of the security review, dirk_y, zigtur, peanuts engaged with Curio to review Curio. In this period of time a total of **16** issues were uncovered.

Protocol Summary

Protocol Name	Curio
Repository	https://github.com/curio-research/duper-contracts
Date	July 23th 2024 - July 25th 2024
Protocol Type	Onchain crypto games

Findings Count

Severity	**Amount**
Low	16
Total Findings	**16**

Summary of Findings

ID	Title	Severity	Status
[L-01]	The earlyEndGame method can be called with non-existent game ids	Low	Resolved
[L-02]	No validation on player stakes and payouts in end-game logic	Low	Acknowledged
[L-03]	Player can be slashed by an arbitrary amount	Low	Acknowledged
[L-04]	Slashed player may break endGame	Low	Resolved
[L-05]	gameId can contain non-printable characters	Low	Acknowledged
[L-06]	Owner setters do not emit an event	Low	Resolved
[L-07]	emergencyEnd does not emit an event	Low	Resolved
[L-08]	StartWithdrawal can be emitted multiple times for a single withdrawal	Low	Resolved
[L-09]	RemoveAdmin event is emitted even if a user was not admin	Low	Acknowledged
[L-10]	emergencyEnd can be used with a player that has no game	Low	Resolved
[L-11]	Admins can call multiple functions when the contract is paused	Low	Acknowledged
[L-12]	Updating withdrawal period impacts ongoing withdrawals	Low	Acknowledged
[L-13]	Withdrawals can be of an unexpected duration	Low	Acknowledged
[L-14]	Players can gas grief admins	Low	Acknowledged
[L-15]	Admin can drain Game's funds	Low	Acknowledged

[L-16]	The gameFee calculation is unfair towards the winners	Low	Acknowledged

8. Findings

8.1. Low Findings

[L-01] The `earlyEndGame` method can be called with non-existent game ids

Currently, the `earlyEndGame` method validates that a game has not ended:

```
if (currentSession.status == GameStatus.Ended) revert GameEnded();
```

A more sensible check would be to validate that the game is in progress; this is a stricter check. With the current loose check, it is possible for the admin to call with an empty string game id, since that game will not exist and thus would be considered not-started. A player that is not in another game will be assigned to an empty string game id, so it is possible to deduct arbitrary balances from the user via this route.

[L-02] No validation on player stakes and payouts in end-game logic

Currently, the admin can arbitrarily provide user stakes and payouts when ending a game. There is no on-chain validation that the player stakes were what they were when the game started, that the player stakes sum correctly, and that the payouts also sum to the correct value.

As such, if the off-chain process was exploited it would be possible to make an arbitrary payout to a user and drain the contract. I would highly recommend adding some on-chain checks to validate that the payouts don't break any key invariants.

The only validation that does exist is:

```
if (balance[player] < deduction) revert InsufficientBalance();
```

Since the admin can provide arbitrary inputs this is actually not the best check in my opinion. Either this should be adjusted to set the user balance to 0 (instead of reverting and acting as a DoS), or the inputs should be validated on-chain as suggested above.

[L-03] Player can be slashed by an arbitrary amount

The `slashPlayer` allows an admin to slash a player. This function is meant to be used when the player cheats.

However, this function does not cap the amount being slashed. This amount should be capped to the `minStake` value of the game for which the player cheated.

[L-04] Slashed player may break `endGame`

The `endGame` function can revert when players lose money and their balance doesn't cover the loss.

This is likely to happen when a player is slashed (reducing balance). Then, `endGame` will revert due to insufficient balance.

```
function endGame(
      stringcalldatagameId,
      PlayerPayout[]memoryplayerPayouts
    ) external onlyAdmin whenNotPaused {
        ...

        for (uint256 i = 0; i < playerPayouts.length; i++) {
            ...

            if (playerPayout.payout > playerPayout.stake) {
                // Player won money case
                ...
            } else {
                // Player lost money case

                uint256 deduction = playerPayout.stake - playerPayout.payout;

                if (balance[player] < deduction) revert InsufficientBalance
                //(); // @POC: balance can be lower than deduction, reverting the tran

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

            ...
        }

        ...
    }
```

When a player is slashed, the player should be out of the game.

[L-05] `gameId` can contain non-printable characters

When registering a `gameId` string, there is no check to ensure that this string contains only printable characters.

This provides the ability to register `gameId` strings that are the same when printed but different in Solidity representation.

Here is an example:

- `0461626364` byte representation is displayed `abcd`
- `056162636400` byte representation is also displayed `abcd` but will be considered different gameId in the contract

`startGame` should check that the string parameters contain only printable characters. This applies to `gameId` and `gameType` parameters.

[L-06] Owner setters do not emit an event

The `updateWithdrawalPeriod` and `updateSparksTopUpPrice` setters are only callable by the owner.

These two functions do not emit events. This leads to the incapacity of monitoring the update of the withdrawal period and sparks top-up prices.

[L-07] `emergencyEnd` does not emit an event

The `emergencyEnd` function does not emit an event.

This leads to an incapacity to monitor the use of this functionality.

[L-08] `StartWithdrawal` can be emitted multiple times for a single withdrawal

The `startWithdrawal` function creates a withdrawal with an unique identifier derived from the caller's address, the amount and the current timestamp. However, the function does not check that this unique identifier is already used or not.

By calling `startWithdrawal` with the same amount in a single block, this lack of check leads a player to being able to emit multiple times the same event. This could potentially break off-chain monitoring.

A check should be done to ensure that the withdrawal identifier is not already in use.

```
@@
   -89,6 +89,9 @@ contract Game is Ownable2StepUpgradeable, UUPSUpgradeable, PausableU
     /// @notice Thrown when player has topped up sparks
     error HasToppedUpSparks();

+    /// @notice Thrown when withdrawal already exist
+    error WithdrawalIdUsed();
+
     /*////////////////////////////////////////////////////////////////
                                 MODIFIERS
     ////////////////////////////////////////////////////////////////*/
@@
   -216,6 +219,7 @@ contract Game is Ownable2StepUpgradeable, UUPSUpgradeable, Pausabl
         withdrawal.timestamp = block.timestamp;
         string memory withdrawalId = _generateWithdrawalId(withdrawal);

+        if
+ (allWithdrawals[withdrawalId].timestamp != 0) revert WithdrawalIdUsed();
         allWithdrawals[withdrawalId] = withdrawal;

         if (address(withdrawalIdSet[msg.sender]) == address(0)) {
```

[L-09] `RemoveAdmin` event is emitted even if a user was not admin

`removeAdmin` function calls the `remove` function of the `admins` AddressSet contract. This `AddressSet.remove` function does not revert when the entry does not exist.

```
function remove(address e) public onlyOwner {
        if (!includes(e)) return; // @POC: return and not revert

        ...
    }
```

Calling `Game.removeAdmin` with a non-admin address will succeed and will emit the `RemoveAdmin` event when it should not.

[L-10] `emergencyEnd` can be used with a player that has no game

`emergencyEnd` function can be used with a player who has no game.

The function must check that the player is currently in a game.

```
/// @dev In emergency scenarios where players are locked, this function can
     // be used to unlock player and pay
     function emergencyEnd
       (address player, uint256 payout) external onlyAdmin whenNotPaused {
+        if (_strEq(playerGameId[player], "")) revert PlayerNotInGame(player);
         playerGameId[player] = "";

         balance[player] += payout;
```

[L-11] Admins can call multiple functions when the contract is paused

Multiple admin functions are still callable when the contract is paused, while others are not.

`startGame`, `earlyEndGame`, `emergencyEnd`, and `endGame` are not callable when the contract is paused while `giveCredit`, `depositTotalCredits`, `setInstantWithdraw`, and `slashPlayer` can be called.

[L-12] Updating withdrawal period impacts ongoing withdrawals

The `updateWithdrawalPeriod` function allows the owner to modify the withdrawal period.

However, modifying the withdrawal period will impact all ongoing withdrawals and not only the new withdrawals.

[L-13] Withdrawals can be of an unexpected duration

When a user starts a withdrawal by calling `startWithdrawal`, the block timestamp is stored in the withdrawal struct. After the withdrawal period has passed, the user can call `finishWithdrawal` to cash out.

However, the problem with the current logic is that the `withdrawalPeriod` variable can change in the time between the user starting their withdrawal and finishing their withdrawal, providing a non-deterministic experience for users

and violating the transparency you'd expect from a smart contract. For example, at the time of starting a withdrawal, the withdrawal period might be 1 day. After 1 day the user comes back to finish their withdrawal, but in the time since the withdrawal was started, the owner changed the withdrawal period to 30 days. The user is now forced to wait another 29 days, despite initiating when the withdrawal period was 1 day.

Instead of storing the withdrawal initiation time in the withdrawal structs, instead, store the unlock time. This way the unlock time is deterministic for a user when they initiate a withdrawal.

[L-14] Players can gas grief admins

At any time a player can start a withdrawal by calling `startWithdrawal` with an arbitrary amount. There is no restriction as to the minimum amount that a user can withdraw at a time and how many withdrawals a user can start at the same time.

As such, a user could start a large number of very small (e.g. 1 wei in the extreme case) withdrawals at the same time and leave them unfinished. Now, any time a user wants to join a game, the admin calling `startGame` is forced to iterate through all the pending withdrawals of the user and burn an unnecessary amount of gas. This is because `getWithdrawableBalance` is called, which iterates through all the pending withdrawals of the user under the hood. The users are gas griefing the admins whilst still legitimately playing games.

Cap the number of concurrent withdrawals that a player can make. For example, max 5 pending withdrawals at one time (the exact number is up for debate).

[L-15] Admin can drain Game's funds

Through the `emergencyEnd` function, an admin can allocate an arbitrary amount of tokens to any player (including himself).

Then, he can set this player as an instant withdrawer through the `setInstantWithdraw`.

Finally, this player is able to instantly withdraw the allocated arbitrary amount of tokens through `startWithdrawal`.

The `emergencyEnd` function must be callable only for players that are currently in a game. Moreover, the allocated payout should be capped to the `minStake` amount of this player's game.

```
/// @dev In emergency scenarios where players are locked, this function can
     // be used to unlock player and pay
     function emergencyEnd
       (address player, uint256 payout) external onlyAdmin whenNotPaused {
+        string memory gameId = playerGameId[player];
+        if (_strEq(gameId, "")) revert PlayerNotInGame(player);
+        if (payout > games[gameId].minStake) revert ValueOutsideRange();
         playerGameId[player] = "";

         balance[player] += payout;
```

[L-16] The gameFee calculation is unfair towards the winners

In every game, all players must pay a certain amount of fees, depending on the fee rate (percentage).

The fees are calculated through the player's payout when the game ends, either through `earlyEndGame()` or `endGame()`.

When calculating player fees, it seems that the higher the payout of the player, the higher the fees paid. The losers of the game will not have to pay as their payout will be less than their stake amount.

```
// Fetch game take rate
>            uint256 playerFee = (playerPayout.payout * gameFeeRate) / 100;
             gameFee += playerFee;
             playerPayout.payout -= playerFee;
```

This test case scenario shows that the loser does not have to pay any fee because the payout is zero, and the winner has to pay a much higher fee.

```
struct PlayerPayout {
    address playerAddress;
    uint256 stake;
    uint256 payout;
}

    function testEndGame() public {
        ...
        PlayerPayout[] memory payouts = new PlayerPayout[](2);
>       payouts[0] = PlayerPayout(player1, 0, depositAmount * 2);
>       payouts[1] = PlayerPayout
//(player2, depositAmount * 2, 0); // we are taking twice what player2 has
```

If not intended, recommend everyone paying a flat base fee at the start of the game, instead of at the end of the game.