Dumont Security Review

Pashov Audit Group

Conducted by: juancito, Dan Ogurtsov, btk

July 8th 2024 - July 12th 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 **dumontgg/dumont-contract-v1** repository was done by **Pashov Audit Group**, with a focus on the security aspects of the application's smart contracts implementation.

4. About Dumont

Dumont is the blockchain game that allows a single player to guess the numbers of hidden cards, which are initially hashed and stored in the contract. The game uses a commit-reveal mechanism, where the player places bets on their guesses, and the server (or 'revealer') reveals the card numbers to determine the winner.

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 - 3165eff301b2fe23a6ca38d785075e507d9ae879

fixes review commit hash - 8bb7d876d36267ab05604c0e3df71826573e83b9

Scope

The following smart contracts were in scope of the audit:

- `Vault`
- `Revealer`
- `MontRewardManager`
- `MONT`
- `GameFactory`
- `Game`
- `Burner`
- `Initializable`
- `interfaces/`

7. Executive Summary

Over the course of the security review, juancito, Dan Ogurtsov, btk engaged with Dumont to review Dumont. In this period of time a total of **22** issues were uncovered.

Protocol Summary

Protocol Name	Dumont
Repository	https://github.com/dumontgg/dumont-contract-v1
Date	July 8th 2024 - July 12th 2024
Protocol Type	Blockchain card game

Findings Count

Severity	**Amount**
Critical	1
High	2
Medium	7
Low	12
Total Findings	**22**

Summary of Findings

ID	Title	Severity	Status
[C-01]	The house fee cap mechanism is flawed	Critical	Resolved
[H-01]	Fees are not burnt on losing games	High	Resolved
[H-02]	Mont price can be manipulated to earn more rewards	High	Resolved
[M-01]	Using the same hash across games	Medium	Resolved
[M-02]	More rewards than it should be when MONT price is low	Medium	Resolved
[M-03]	Free card reveals by choosing all of remaining cards	Medium	Resolved
[M-04]	Users are penalized when operators are inactive	Medium	Resolved
[M-05]	Some of the expected rewards are not distributed	Medium	Acknowledged
[M-06]	Centralization Risks	Medium	Resolved
[M-07]	Exploiting referrer bonus to self / controlled address	Medium	Acknowledged
[L-01]	Missing sanity checks on _lockedAmount	Low	Resolved
[L-02]	Missing a cap for GameFactory fee	Low	Resolved
[L-03]	TaxBasedLocker can be re-initialized	Low	Resolved
[L-04]	Extra tokens sent to TaxBasedLocker	Low	Resolved
[L-05]	Game creation should not revert if using the same referrer	Low	Resolved

[L-06]	Division by zero in getGuessRate	Low	Resolved
[L-07]	Requesting free card reveals for non-existing card indexes	Low	Resolved
[L-08]	gameCreatedAt should be set during initialization	Low	Acknowledged
[L-09]	Calling revealCard() after claimableAfter has passed	Low	Resolved
[L-10]	Game.initialize() check unique hashes	Low	Acknowledged
[L-11]	Setting a deadline to the current timestamp	Low	Resolved
[L-12]	MONT rewards are multiplied incorrectly	Low	Acknowledged

8. Findings

8.1. Critical Findings

[C-01] The house fee cap mechanism is flawed

Severity

Impact: High

Likelihood: High

Description

The `transferPlayerRewards()` function attempts to implement a protection mechanism to prevent abuses from an operator that can access game outcomes and liquidity, participate as a player, place bets without risk, and deplete the reward pool.

The problem is that the mechanism is flawed, and not only it doesn't protect the protocol from such an attack, but it caps user rewards substantially.

Note how `reward` is set to `_totalAmount` when it is bigger. The error is that `reward` is measured in MONT tokens, and `_totalAmount` represents USDT tokens, which can't be directly compared as they have different values.

Moreover, USDT has 6 decimals of precision in most chains, while MONT has 18, which makes the impact even bigger as the rewards will be significantly capped:

```
function transferPlayerRewards(...) external returns (uint256 reward) {
        uint256 houseFee = calculateHouseFee
          (_betAmount, _houseEdgeAmount, _isPlayerWinner);

        uint256 price = getMontPrice();

        uint256 houseFeeFixedPoint = houseFee * 1e18;

        reward = ((houseFeeFixedPoint * 8) / 10) / price;

@>      if (reward > _totalAmount) {
@>          reward = _totalAmount;
        }

@>      balances[_player] += reward;
```

Proof of Concept

Add these logs to `MontRewardManager.sol` and run the `test_revealCardWithTheRightData` test:

```
+       console.log("USDT bet amount:        ", _betAmount);
+       console.log("USDT total won amount:  ", _totalAmount);
+       console.log("MONT expected reward:   ", reward);

        if (reward > _totalAmount) {
                reward = _totalAmount;
        }

+       console.log("MONT actual reward:     ", reward);
```

Results:

```
Ran 1 test for test/integration/revealer/revealCard.t.sol:RevealCardTest
[PASS] test_revealCardWithTheRightData() (gas: 292421)
Logs:
  USDT bet amount:         1000000
  USDT total won amount:   11800000
  MONT expected reward:    500000000000000000000
  MONT actual reward:      11800000
```

As it can be seen, the user receives much less rewards. MONT precision is 18 decimals, so the rewards are several orders of magnitude lower than expected.

Recommendations

1. As the code stands, 80% of the USDT house fees are sent to the burner via the Vault to be burnt later, and the equivalent in MONT tokens is rewarded to the player and their referrer according to the price of the MONT/USDT pool, which is the expected distribution.

So the cap should be removed:

```
-    if (reward > _totalAmount) {
-        reward = _totalAmount;
-    }
```

An operator who misbehaves would still lose the equivalent of 20% of the house fees on each win (as they are kept in the vault). On the other hand, they can deplete the vault by winning continuously with a big guess rate multiplier.

So, in the case of untrusted operators, a more robust prevention/penalization mechanism should be built taking into account the mentioned issues, while not penalizing fair users.

2. In addition, consider scaling `_totalAmount` to 18 decimals before comparing it with the reward.

8.2. High Findings

[H-01] Fees are not burnt on losing games

Severity

Impact: Medium

Likelihood: High

Description

According to the burning mechanism, a 10% fee should applied on the winning or **losing** amount of every bet. 80% of those USDT fees will be swapped at the end of each month to buy and burn MONT tokens.

The problem is that the Vault is only sending assets to the burner when the player is a winner:

```
function transferPlayerRewards(...) external {
        IGameFactory.GameDetails memory game = gameFactory.games(_gameId);

        ...

        uint256 burnAmount = (houseEdge * 8) / 10;

        if (_isPlayerWinner) {
            usdt.safeTransfer(game.player, _totalAmount);
@>          usdt.safeTransfer(address(burner), burnAmount);

            emit PlayerRewardsTransferred(game.player, _totalAmount);
        }
```

`MontRewardManager` will still distribute MONT rewards on losing games, equivalent to 10% of the bet amount.

The issue with this is that selling pressure won't be compensated with the corresponding buying pressure expected from the token burning, leading to an impact on the price of MONT.

Recommendations

```
uint256 burnAmount = (houseEdge * 8) / 10;
+   usdt.safeTransfer(address(burner), burnAmount);

    if (_isPlayerWinner) {
        usdt.safeTransfer(game.player, _totalAmount);
-       usdt.safeTransfer(address(burner), burnAmount);

        emit PlayerRewardsTransferred(game.player, _totalAmount);
    }
```

[H-02] Mont price can be manipulated to earn more rewards

Severity

Impact: High

Likelihood: Medium

Description

The `MontRewardManager` contract uses the `quoteExactInputSingle()` function from the Uniswap Quoter to get the USDT price of one unit of MONT.

The problem is that this price can be easily manipulated since it uses the current pool price for its calculations.

```
function transferPlayerRewards(...) external returns (uint256 reward) {
        uint256 price = getMontPrice();
        ...

        reward = ((houseFeeFixedPoint * 8) / 10) / price;

        balances[_player] += reward;
    }

    function getMontPrice() private returns (uint256 price) {
        price = quoter.quoteExactInputSingle(address(mont), address
          (usdt), poolFee, 1e18, 0);
    }
```

A player can perform an attack sandwiching the `revealCard()` transaction sent by an operator.

This attack can be timed as operators will most likely call the reveal card transaction within an expected timeframe window after the player calls `guessCard()`.

This is how an attack may proceed:

1. Estimate when the `revealCard()` transaction will be sent or frontrun it if possible
2. Swap to manipulate the price of MONT to go down
3. The operator calls `revealCard()` and triggers the reward transfer
4. An unfairly big amount of rewards will be added to the player as the MONT price will be low
5. Claim the MONT rewards via `claim()`
6. Swap to manipulate the price to get back to its initial value or keep it as it is
7. The attacker will profit from an additional amount of MONT tokens

It can also be performed the other way around, by making the price go up before the reveal so that other players receive fewer rewards.

Recommendations

Use an oracle or TWAP to get the price of MONT.

8.3. Medium Findings

[M-01] Using the same hash across games

Severity

Impact: High

Likelihood: Low

Description

If a hash is revealed in one game its params become public - `_number` and `_salt`. This hash must not be used in other games. Otherwise, there is a risk of exploiting a game (guaranteed wins leverage by max available amounts).

Recommendations

Consider additionally encoding `address(this)` in `verifySalt()` to ensure that the hash can only be used in this game.

[M-02] More rewards than it should be when MONT price is low

Severity

Impact: Medium

Likelihood: Medium

Description

The MONT rewards are calculated given this formula:

```
function transferPlayerRewards(...) external returns (uint256 reward) {
    ...
    uint256 price = getMontPrice();
    uint256 houseFeeFixedPoint = houseFee * 1e18;
    reward = ((houseFeeFixedPoint * 8) / 10) / price;
    ...
}

function getMontPrice() private returns (uint256 price) {
    price = quoter.quoteExactInputSingle(address(mont), address
      (usdt), poolFee, 1e18, 0);
}
```

The problem is that MONT tokens have 18 decimals of precision, while USDT has 6 decimals of precision in most chains.

By requesting the price of 1 whole unit of MONT (1e18 tokens), the returned price will suffer from precision loss and will impact the calculated rewards.

Assuming the USDT price is $1 and the MONT price is $0.0000159, the `getMontPrice()` function will return `15`, which incurs in precision loss of around 6%.

This means that users would receive around 6% more MONT rewards than they should in this case. The lower the MONT price, the bigger the rewards will be.

Recommendations

Considering USDT has 6 decimals of precision, one possible way is to quote the inverse price (how many MONT tokens per 1e6 unit of USDT) and refactor the reward formula, so that the precision loss is insignificant and players get the rewards they should.

[M-03] Free card reveals by choosing all of remaining cards

Severity

Impact: Low

Likelihood: High

Description

The `getGuessRate()` function lets players choose all the remaining cards as `_guessedNumbers`, meaning that they will always win:

```
function getGuessRate(uint256 _numbers) public view returns (UD60x18 rate) {
    uint256 remainingCards = 52 - cardsRevealed;
    uint256 remainingSelectedCard = 0;

    for (uint256 i = 0; i < 13; ++i) {
        if ((_numbers & (1 << i)) > 0) {
            remainingSelectedCard += 4 - revealedCardNumbersCount[i];
        }
    }

    rate = ud(remainingCards).div(ud(remainingSelectedCard));
}
```

The total winning amount will be 1x:

```
uint256 totalWinningBetAmount = getGuessRate(_guessedNumbers).mul(ud
  (_betAmount)).unwrap();
```

They will receive the same bet amount as a reward:

```
uint256 reward = totalWinningBetAmount - (
  (totalWinningBetAmount - _betAmount) / 10);
```

While the house edge will be zero:

```
_card.houseEdgeAmount = totalWinningBetAmount - reward;
```

In other words, players can get free card guesses, bypassing the `maxFreeReveals` limit implemented in `requestFreeRevealCard()`, and also making the operator process more free reveals than intended.

Recommendations

Consider adding this check to `getGuessRate()`, so that there is always a house edge amount to be taken from the player.

```
+   require(remainingCards != remainingSelectedCard);
```

[M-04] Users are penalized when operators are inactive

Severity

Impact: High

Likelihood: Low

Description

When operators are inactive and don't call `revealCard()` in a timely manner, users can still claim their win via `claimWin()`.

This triggers `transferPlayerRewards()` with the conditional that the player shouldn't receive reward tokens:

```
function claimWin(uint256 _index) external onlyPlayer onlyInitialized {
    ...

    vault.transferPlayerRewards(

                gameId, _card.betAmount, _card.totalAmount, _card.houseEdgeAmount, tr
    );
}
```

The problem is that house edge fees were already taken from the player. 80% will be burnt despite no rewards being issued and the other 20% will be kept in the Vault.

So, the users are penalized in this scenario where they did nothing wrong, but where the operators were at fault.

```
function transferPlayerRewards(...) external {
        ...

        uint256 houseEdge = _betAmount / 10;

        if (_isPlayerWinner) {
            houseEdge = _houseEdgeAmount;
        }

        uint256 burnAmount = (houseEdge * 8) / 10;

        if (_isPlayerWinner) {
            usdt.safeTransfer(game.player, _totalAmount);
            usdt.safeTransfer(address(burner), burnAmount);
        }

@>      if (_receiveMontReward) {
@>          montRewardManager.transferPlayerRewards
  (_betAmount, _totalAmount, houseEdge, game.player, _isPlayerWinner);
        }
    }
```

Recommendations

In the scenario that no MONT rewards are issued, consider compensating the player by transferring back the house edge fees or a part of them (and not sending them to the burner).

[M-05] Some of the expected rewards are not distributed

Severity

Impact: Low

Likelihood: High

Description

The `transferPlayerRewards()` function recalculates the rewards to be distributed to the player and their referral.

In the case there is no referral, only 80% are expected to be distributed.

If there exists a referral all the rewards are expected to be distributed, but only 99% are distributed due to how they are calculated.

Example for 100_000 tokens: 90_000 for the player + 9_000 for the referrer + 1_000 not distributed

```
(bool isReferrerSet, address referrer) = checkReferrer(_player);

    if (!isReferrerSet) {
        reward = (reward * 8) / 10;
    } else if (isReferrerSet) {
        reward = (reward * 9) / 10;
        balances[referrer] += reward / 10;
    }

    balances[_player] += reward;
```

Recommendations

Here are two suggested alternative distributions.

Option 1: The player receives 90% of the original reward, and the referrer receives 10% of the original reward. The referrer would get 9% of the wins received by the player.

Example for 100_000 tokens: 90_000 for the player + 10_000 for the referrer

```
(bool isReferrerSet, address referrer) = checkReferrer(_player);

    if (!isReferrerSet) {
        reward = (reward * 8) / 10;
    } else if (isReferrerSet) {
-       reward = (reward * 9) / 10;
        balances[referrer] += reward / 10;
+       reward = (reward * 9) / 10;
    }

    balances[_player] += reward;
```

Option 2: Calculate the amount so that the referrer will receive 10% of the rewards the player receives. This means ~90.9% of the original rewards for the player + ~9.1% of the original rewards for the referrer.

Example for 100_000 tokens: 90_909 for the player + 9_090 for the referrer + 1 wei not distributed

```
(bool isReferrerSet, address referrer) = checkReferrer(_player);

    if (!isReferrerSet) {
        reward = (reward * 8) / 10;
    } else if (isReferrerSet) {
-       reward = (reward * 9) / 10;
+       reward = (reward * 10) / 11;
        balances[referrer] += reward / 10;
    }

    balances[_player] += reward;
```

[M-06] Centralization Risks

Severity

Impact: High

Likelihood: Low

Description

Given a malicious or compromised owner, user assets may be at risk in the following scenarios:

- USDT to pay rewards for ongoing games can be withdrawn from the Vault via `withdraw()` at any time.
- The vault can also be depleted by operators playing themselves and making them always win with big guess rate multipliers.
- The `gameFactory` from the Vault can be changed at any time via `setGameFactory()` to later call `transferPlayerRewards()` with deceiving game information to claim any amount of rewards.
- The `montRewardManager` from the Vault can be changed at any time via `setMontRewardManager()` to prevent `transferPlayerRewards()` from executing and reverting on player wins.
- The `minimumBetAmount` and `maximimBetRate` values can be modified at any time in the Vault to prevent players from guessing cards.
- The `gameCreationFee` in the GameFactory can be changed right before a game is created to take all the approved USDT from a player when calling `createGame()`. Other game settings that the player may not agree with can be changed at any time, right before they call `createGame()`.

Recommendations

Some suggestions to mitigate centralization risks:

- Lock USDT rewards in the Vault when a bet is made via `guessCard()` until `revealCard()` or `claimWin()` are executed. Release the lock after it. This will make sure that there is enough liquidity to pay for ongoing bets.
- Create a timelock contract and apply any changes to Vault and GameFactory settings after a certain time, so that users are aware of them. This will prevent changing the underlying logic of the system with new external contracts or setting unfair values.

[M-07] Exploiting referrer bonus to self / controlled address

Severity

Impact: Low

Likelihood: High

Description

Dumont implemented an on-chain referral program that offers bonus rewards equivalent to 10% of the in-game $MONT rewards earned by invitees. For instance, if your invitees earn 20,000 $MONT in rewards, you will receive an additional 2,000 $MONT as a referral reward. Additionally, users who join the game through a referral link earn 10% more than those who join without one.

Having a referrer is always profitable, as the total reward with a referrer is always higher than without a referrer.

```
if (!isReferrerSet) {
            reward = (reward * 8) / 10;
        } else if (isReferrerSet) {
            reward = (reward * 9) / 10;

            balances[referrer] += reward / 10;
        }

        balances[_player] += reward;
```

As a result, users can indicate their controlled EOAs as a referrer to benefit from this additional bonus.

Recommendations

It would be fair if a referrer is an optional parameter that does not influence the final total sum of rewards. Consider taking referrer rewards as a fee from user rewards, not a bonus on top.

Or, consider moving the referral program off-chain.

8.4. Low Findings

[L-01] Missing sanity checks on `_lockedAmount`

In the initialize() function of the TaxBasedLocker contract, the contract is initialized with a specified amount of tokens to lock. According to NatSpec requirements, it is mandated that "The specified amount of tokens must be greater than zero." However, the current implementation lacks validation to ensure `_lockedAmount` is indeed greater than zero, as illustrated below:

```
function initialize(uint256 _lockedAmount) external onlyOwner {
        if (lockedAmount > 0) {
            revert AlreadyInitialized();
        }
        lockedAmount = _lockedAmount;
        uint256 balance = token.balanceOf(address(this));
        if (balance < lockedAmount) {
            token.safeTransferFrom(owner(), address
              (this), lockedAmount - balance);
        }
        startTime = block.timestamp;
        emit Initialized();
    }
```

Add a check to confirm that the specified amount of tokens is greater than zero to align with NatSpec requirements.

[L-02] Missing a cap for GameFactory fee

The function `setGameCreationFee()` allows the owner to set a new `gameCreationFee` percentage for a given instance. However, there is no upper limit or cap defined for the `_gameCreationFee` parameter:

```
function setGameCreationFee(uint256 _gameCreationFee) external onlyOwner {
        emit GameFeeChanged(gameCreationFee, _gameCreationFee);

        gameCreationFee = _gameCreationFee;
    }
```

This could potentially allow for an excessively high fee to be set, which could discourage users from interacting with the contract. Introduce a maximum

limit for the `gameCreationFee` to ensure that it cannot be set to an excessively high value.

[L-03] TaxBasedLocker can be re-initialized

The `initialize()` function only checks for `lockedAmount > 0` for it to be initialized, but that value is reset on `withdraw()`, making it possible to re-initialize the contract:

```
function initialize(uint256 _lockedAmount) external onlyOwner {
        if (lockedAmount > 0) {
            revert AlreadyInitialized();
        }
        ...
    }

    function withdraw() external onlyOwner {
        ...
        lockedAmount = 0;
    }
```

Consider setting a proper initializer variable for the check.

[L-04] Extra tokens sent to TaxBasedLocker

Tokens locked in TaxBasedLocker can either be first sent to the contract or transferred directly when initializing it.

If the balance is lower than the locked amount, the remaining funds will be transferred.

The problem is that if more tokens were sent beforehand than the `lockedAmount` value, those will never be unlockable, remaining stuck in the contract.

```
function initialize(uint256 lockedAmount) external onlyOwner {
        ...
        uint256 balance = token.balanceOf(address(this));

        if (balance < lockedAmount) {
            token.safeTransferFrom(owner(), address
              (this), lockedAmount - balance);
        }
    }
```

Consider reverting when `balance > _lockedAmount` (while making sure to allow both values to be equal).

[L-05] Game creation should not revert if using the same referrer

The `createGame()` function reverts when using the same referrer previously used by the same user.

This leads to a sub-optimal UX, as users may enter the same valid referral code for a new game but the transaction will now revert.

```
function setReferrer(address _referee, address _referrer) private {
        ...
        if (referrals[_referee] != address(0)) {
            revert ReferralAlreadySet(_referee, referrals[_referee]);
        }

        referrals[_referee] = _referrer;
    }
```

Consider avoiding additional checks when using the same referrer:

```
function setReferrer(address _referee, address _referrer) private {
-       if (_referrer == address(0)) {
+       if (_referrer == address(0) || referrals[_referee] == _referrer) {
            return;
        }
```

[L-06] Division by zero in getGuessRate

The `remainingSelectedCard` can end up being zero if a user picks a card that has already been revealed. For example, picking an Ace if all Aces have been revealed. This would lead to a division by zero when calculating the `rate`:

```
function getGuessRate(uint256 _numbers) public view returns (UD60x18 rate) {
        uint256 remainingCards = 52 - cardsRevealed;
        uint256 remainingSelectedCard = 0;

        for (uint256 i = 0; i < 13; ++i) {
            if ((_numbers & (1 << i)) > 0) {
                remainingSelectedCard += 4 - revealedCardNumbersCount[i];
            }
        }

        rate = ud(remainingCards).div(ud(remainingSelectedCard));
    }
```

The transaction would still revert as the `prb` library does, but regardless of that, it would be recommended to prevent the division by zero with a check like

```
+   require(remainingSelectedCard > 0);
```

[L-07] Requesting free card reveals for non-existing card indexes

Card indexes range from 0 to 51, but it is possible to request a card reveal for any value > 51.

Since the `uint8` equivalent to `CardStatus.SECRETED` is zero, the `CardIsNotSecret` error will not be triggered as `_card.status` for non-existing cards will always be zero.

This will emit a `RevealFreeCardRequested` that the operator will try to process, which can lead to off-chain errors or unnecessary computation.

```
function requestFreeRevealCard
      (uint256 _index) external onlyPlayer onlyInitialized notExpired {
        Card storage _card = _cards[_index];

        if (cardsFreeRevealedRequests == maxFreeReveals) {
            revert MaximumFreeRevealsRequested();
        }

@>      if (_card.status != CardStatus.SECRETED) {
            revert CardIsNotSecret(_index);
        }

        ++cardsFreeRevealedRequests;
        _card.requestedAt = block.timestamp;
        _card.status = CardStatus.FREE_REVEAL_REQUESTED;

        emit RevealFreeCardRequested(_index, block.timestamp);
    }
```

Consider checking that the index is between the expected boundaries:

```
+   if (_index > 51) {
+       revert InvalidGameIndex();
+   }
```

[L-08] `gameCreatedAt` should be set during initialization

Each game contract has a specified duration. For instance, if the duration is set to 12 hours, the game becomes unplayable after that period. This constraint is enforced as follows:

```
modifier notExpired() {
        if (gameDuration + gameCreatedAt < block.timestamp) {
            revert GameExpired();
        }

        _;
    }
```

- `gameDuration` refers to the length of the game.
- `gameCreatedAt` is the timestamp when the game was deployed.

Games cannot be played immediately after deployment; players must wait for the operator to initialize their game. The issue arises because operators are EOAs, and they might take some time to initialize all created games. For example, if Bob creates a game at 6 am, gameCreatedAt will be set to that current timestamp. If the operator takes 2 hours to initialize Bob's game (which

is likely given there are no incentives for operators), the actual playable duration for Bob's game becomes 10 hours instead of the intended 12 hours.

Consider setting gameCreatedAt during initialization instead of at deployment.

[L-09] Calling `revealCard()` after `claimableAfter` has passed

When a player guesses the value of a card, the operator is responsible for sending the card's number and salt to the Game contract. The operator has a window of 6 hours to call `revealCard()`. If the operator fails to do so, the winner can claim their bet using the `claimWin()` function:

```
function claimWin(uint256 _index) external onlyPlayer onlyInitialized {
        Card storage _card = _cards[_index];
        if (_card.status != CardStatus.GUESSED) {
            revert CardIsNotGuessed(_index);
        }
        if (_card.requestedAt + claimableAfter > block.timestamp) {
            revert NotYetTimeToClaim(_index);
        }
        _card.status = CardStatus.CLAIMED;
        vault.transferPlayerRewards(

                        gameId, _card.betAmount, _card.totalAmount, _card.houseEdgeAm
        );
        emit CardClaimed(_index, block.timestamp);
    }
```

As shown, the winner can only invoke this function after the `claimableAfter` period has elapsed. However, there is no safeguard to prevent the operator from calling `revealCard()` after `claimableAfter` has passed, which contradicts the documentation:

> As you can see, the player bears no risks of getting manipulated, and all the responsibility for security and integrity falls on the shoulders of the operator.

For instance, if the operator fails to submit the salt and card number before `claimableAfter`, and the winner is not available to claim their win immediately after `claimableAfter`, the operator can still call `revealCard()`.

To address this issue, consider adding the following check to the `revealCard()` function:

```
if (_card.requestedAt + claimableAfter <= block.timestamp) revert();
```

[L-10] `Game.initialize()` check unique hashes

It is expected that hashes provided through `Game.initialize()` are unique. But it is not checked. There should not be two cards with the same hash. It will mean that the revealer registered two cards with the same number. It will break significant probability assumptions used in the game. E.g. it will be possible to perfectly guess a number if the hash is revealed once.

Describe your recommendation here

[L-11] Setting a deadline to the current timestamp

The `deadline` parameter on an `exactInputSingle()` swap operation is used to to protect against long-pending transactions and wild swings in prices.

When the deadline is set to `block.timestamp`, the operation will always succeed as the time when the transaction is executed in the node will always be `block.timestamp`.

For long-pending transactions, the `amountOut` may not result in the most favorable outcome. This affects the `Burner` contract when it swaps USDT for MONT to be burnt via `burnTokens()`:

```
function _swap(
      uint256_amountIn,
      uint256_amountOutMinimum
    ) private returns (uint256 amountOut

            fee: uniswapPoolFee,
            amountIn: _amountIn,
            tokenIn: address(usdt),
            tokenOut: address(mont),
            recipient: address(this),
@>          deadline: block.timestamp,
            amountOutMinimum: _amountOutMinimum,
            sqrtPriceLimitX96: 0
        });

        amountOut = swapRouter.exactInputSingle(params);
    }
```

Add a `_deadline` parameter set by the caller to `Burner::burnTokens()` and pass it to the `_swap()` function:

```
-   function _swap
- (uint256 _amountIn, uint256 _amountOutMinimum) private returns (uint256 amountOut) {
+   function _swap
+ (uint256 _amountIn, uint256 _amountOutMinimum, uint256 _deadline) private returns (u

            fee: uniswapPoolFee,
            amountIn: _amountIn,
            tokenIn: address(usdt),
            tokenOut: address(mont),
            recipient: address(this),
-           deadline: block.timestamp,
+           deadline: _deadline,
            amountOutMinimum: _amountOutMinimum,
            sqrtPriceLimitX96: 0
        });

        amountOut = swapRouter.exactInputSingle(params);
    }
```

[L-12] MONT rewards are multiplied incorrectly

The `transferPlayerRewards()` function is intended to distribute rewards to players. Initially, it multiplies the reward by 0.8 to calculate the reward for non-referral program users:

```
reward = ((houseFeeFixedPoint * 8) / 10) / price;
```

At this stage, the `reward` variable is already reduced by 20%. However, within the `transferPlayerRewards()` function, this reward is further reduced by 20%

if the user is not part of the referral program:

```
if (!isReferrerSet) {
            reward = (reward * 8) / 10;
        } else if (isReferrerSet) {
            reward = (reward * 9) / 10;

            balances[referrer] += reward / 10;
        }
```

As a result, rewards for both non-referral and referral users are being depleted. Consider this example (excluding price for simplicity):

- `houseFeeFixedPoint = 200e18`
- `reward = (200e18 * 8) / 10 = 160e18`
- For non-referral users:
  - `reward = (reward * 8) / 10 = 128e18`
- For referral users:
  - `reward = (reward * 9) / 10 = 144e18`

As shown above, non-referral users receive 64% (0.64) of the initial reward, and referral users receive 72% (0.72) instead of the expected 80% (0.8) and 90% (0.9) respectively.

To resolve this issue, update the `transferPlayerRewards()` function to avoid multiple reductions of the reward:

```
function transferPlayerRewards(
        uint256 _betAmount,
        uint256 _totalAmount,
        uint256 _houseEdgeAmount,
        address _player,
        bool _isPlayerWinner
    ) external returns (uint256 reward) {
        if (msg.sender != vault) {
            revert Unauthorized();
        }
        uint256 houseFee = calculateHouseFee
          (_betAmount, _houseEdgeAmount, _isPlayerWinner);
        uint256 price = getMontPrice();
        reward = (houseFee * 1e18) / price;
        if (reward > _totalAmount) {
            reward = _totalAmount;
        }
        (bool isReferrerSet, address referrer) = checkReferrer(_player);
        if (!isReferrerSet) {
            reward = (reward * 8) / 10;
        } else if (isReferrerSet) {
            reward = (reward * 9) / 10;
            balances[referrer] += reward / 10;
        }
        balances[_player] += reward;
        emit MontRewardAssigned(_player, reward);
    }
```