Linea SpinGame Audit Report

Prepared by Cyfrin

Version 2.0

Lead Auditors

Immeas

Farouk

March 19, 2025

Contents

1 About Cyfrin

Cyfrin is a Web3 security company dedicated to bringing industry-leading protection and education to our partners and their projects. Our goal is to create a safe, reliable, and transparent environment for everyone in Web3 and DeFi. Learn more about us at cyfrin.io.

2 Disclaimer

The Cyfrin team makes every effort to find as many vulnerabilities in the code as possible in the given time but holds no responsibility for the findings in this document. A security audit by the team does not endorse the underlying business or product. The audit was time-boxed and the review of the code was solely on the security aspects of the solidity implementation of the contracts.

3 Risk Classification

	Impact: High	**Impact: Medium**	**Impact: Low**
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

4 Protocol Summary

Linea SpinGame is part of the Linea ecosystem, designed to provide additional incentives to users in the form of ERC20, ERC721, or native tokens. It is a spin/lottery-style contract that utilizes Gelato VRF to generate randomness. All user interactions require off-chain signatures from a trusted service before being validated on-chain. Additionally, the protocol team has the ability to boost the win probability for certain participants, increasing their chances of receiving a prize. Users who fail to win any prizes receive a consolation NFT as compensation.

4.1 Actors and Roles

- **1. Actors:**
  - **Protocol Team:** Manages the contracts and provides prizes. Ensures that the contract has a sufficient balance of ERC20, ERC721, or native tokens to pay out prizes.
  - **Off-Chain Service:** Signs user requests for participation and prize claims.
  - **Users:** Participate in the Linea ecosystem and have a chance to win prizes.
- **2. Roles:**
  - `CONTROLLER`:
    - Can update core contract settings, including:
      - Changing the Gelato VRF Operator
      - Updating the signer address
      - Adding or updating prizes
      - Withdraw ERC20, ERC721 and native tokens from the contract
      - Cancelling expired participation requests (pending for more than 1 hour)
      - Managing consolation NFTs (changing the address, updating URIs, and minting)

  - `DEFAULT_ADMIN_ROLE`:
    - Has all the permissions of `CONTROLLER`
    - Can assign new controllers
  - `signer`:
    - A wallet responsible for signing user interactions and handing out boosts

4.2 Key Components

- **Spin:** The core contract responsible for handling the lottery and prize distribution.
- **LuckyNFT:** A consolation NFT awarded to users who do not win any other prizes.

4.3 Centralization Risks

The `CONTROLLER` and `DEFAULT_ADMIN_ROLE` accounts have extensive privileges, including full access to the contract and the ability to withdraw funds.

Additionally, the `signer` wallet has the ability to spam participation requests, which could deplete available prizes and drain the protocol's Gelato balance.

To mitigate risks, these wallets must be securely managed to prevent unauthorized access or potential takeovers.

4.4 ERC20 Token Concerns

The protocol team noted that some ERC20 prize tokens may include common DeFi protocol tokens, such as AAVE aTokens. Some DeFi tokens are rebasing and can decrease in balance over time, leading to potential depreciation in value.

If rebasing tokens are used as prizes, the team should monitor the contract's balance to ensure it remains sufficient for prize payouts.

5 Audit Scope

```
contracts/src/LuckyNFT.sol
contracts/src/Spin.sol
```

6 Executive Summary

Over the course of 4 days, the Cyfrin team conducted an audit on the Linea SpinGame smart contracts provided by Linea. In this period, a total of 8 issues were found.

During the audit, one medium-severity issue was identified, related to the inability to fund the contract with native tokens for prizes.

Additionally, three low-severity findings were discovered:

1. An edge case where a user with 100% win probability could still fail to win.
2. A strategic claiming issue, allowing users to delay prize claims to maximize the value of NFT rewards.
3. A potential overflow when the admin adds prizes, which could bypass validation checks.

The test suite was extensive and provided good coverage of both successful and failure scenarios.

Summary

Project Name	Linea SpinGame
Repository	linea-hub
Commit	295344925ec4...
Audit Timeline	Mar 11th - Mar 14th
Methods	Manual Review

Issues Found

Critical Risk	0
High Risk	0
Medium Risk	1
Low Risk	3
Informational	4
Gas Optimizations	0
Total Issues	8

Summary of Findings

[M-1] Native token prizes cannot be funded due to missing `receive()` function	Resolved
[L-1] Rounding errors in boosted probability calculation can cause guaranteed wins to fail	Resolved
[L-2] Users can select higher-value NFTs by delaying prize claims	Acknowledged
[L-3] Probability overflow can bypass `MaxProbabilityExceeded` check	Resolved
[I-1] Scaling `winningThreshold` incorrectly reduces randomness distribution	Resolved
[I-2] Native token transfers lack explicit balance check	Resolved
[I-3] Race Condition in `updatePrizes` Leading to Unexpected Prizes	Acknowledged
[I-4] Assembly blocks could benefit from `"memory-safe"` annotation	Resolved

7 Findings

7.1 Medium Risk

7.1.1 Native token prizes cannot be funded due to missing `receive()` function

Description: SpinGame supports multiple prize types, including ERC721, ERC20, and native tokens, where native tokens are represented as `prize.tokenAddress = address(0)`.

To ensure that prizes can be successfully claimed, the protocol team is responsible for maintaining a sufficient token balance in the contract by transferring the necessary assets to the Spin contract.

However, there is an issue specifically with native token prizes: the Spin contract does not have a `receive()` or `fallback()` function, and none of its functions are `payable`. This means there is no way for the team to fund the contract with native tokens using a standard transfer, preventing users from successfully claiming native token prizes.

Impact: Native token prizes cannot be claimed because there is no mechanism to deposit native tokens into the contract. The only way to provide a native token balance would involve esoteric workarounds, such as self-destructing a contract that sends funds to the Spin contract.

Proof of Concept: Add the following test to `Spin.t.sol`:

```
function testTransferNativeToken() public {
    vm.deal(admin,1e18);

    vm.prank(admin);
    (bool success, ) = address(spinGame).call{value: 1e18}("");

    // transfer failed as there is no `receive` or `fallback` function
    assertFalse(success);
}
```

Recommended Mitigation: Consider adding a `receive()` function to the contract to allow native token deposits:

```
receive() external payable {}
```

Linea: Fixed in commit d1ab4bd

Cyfrin: Verified.

7.2 Low Risk

7.2.1 Rounding errors in boosted probability calculation can cause guaranteed wins to fail

Description: The Linea SpinGame includes a boosting feature that allows the protocol to increase a specific user's chance of winning. However, this mechanism introduces the possibility of a user's total winning probability exceeding 100%, as the boosted probabilities can sum to a value greater than 100%. To address this, the contract normalizes the total boosted probability in `Spin::_fulfillRandomness`:

```
// Apply boost on the sum of totalProbabilities.
uint256 boostedTotalProbabilities = totalProbabilities * userBoost / BASE_POINT;

// If boostedTotalProbabilities exceeds 100% we have to increase the winning threshold so it stays in
↪  bound.
//
// Example:
//   PrizeA probability: 50%
//   PrizeB probability: 30%
//   User boost: 1.5x
//   boostedPrizeAProbability: 75%
//   boostedPrizeBProbability: 45%
//
//   We now have a total of 120% totalBoostedProbability so we need to increase winning threshold by
↪  boostedTotalProbabilities to BASE_POINT ratio.
//
//   winningThreshold = winningThreshold * 12_000 / 10_000
if (boostedTotalProbabilities > BASE_POINT) {
    winningThreshold =
        (winningThreshold * boostedTotalProbabilities) /
        BASE_POINT;
}
```

Later in `Spin::_fulfillRandomness`, each prize probability is independently scaled when checking if the user has won:

```
// Apply boost on a single prize probability.
uint256 boostedPrizeProbability = prizeProbability * userBoost / BASE_POINT;

unchecked {
    cumulativeProbability += boostedPrizeProbability;
}

if (winningThreshold < cumulativeProbability) {
    selectedPrizeId = localPrizeIds[i];

    // ... win
    break;
}
```

The issue arises from the probability calculation:

```
uint256 boostedPrizeProbability = prize.probability +
    ((prize.probability * userBoost) / BASE_POINT);
```

Due to this calculation, the final `cumulativeProbability` can be lower than `boostedTotalProbabilities`, leading to a scenario where a user who should be guaranteed a win might still lose due to rounding errors.

Impact: A user who theoretically has a 100% chance of winning can still lose. While this is an unlikely edge case, it would be highly problematic for the unlucky user who, despite the math suggesting they are guaranteed a win, does not receive a prize due to numerical precision issues.

Proof of Concept: Consider the following example:

- There are three prizes, each with a 30% probability of being won.
- A user receives a 133% probability boost.

Calculating the boosted probabilities:

```
boostedTotalProbabilities = 0.9e8*133_333_333/1e8 = 119_999_999
boostedPrizeProbability = 0.3e8*133_333_333/1e8 = 39_999_999
```

and

```
3*39_999_999 = 119_999_997
```

In the worst case, the user could get:

```
winningThreshold = 99_999_999
```

Applying the threshold adjustment:

```
if (boostedTotalProbabilities > BASE_POINT) {
    winningThreshold =
        (winningThreshold * boostedTotalProbabilities) /
        BASE_POINT;
}
```

This results in:

```
winningThreshold = 99_999_999 * 119_999_999 / 1e8 = 119_999_997
```

Since `winningThreshold < cumulativeProbability` is the condition for winning, and:

```
119_999_997 < 119_999_997  // (false)
```

The condition fails, meaning the user loses, even though they were supposed to be guaranteed a win. This issue is caused by the rounding errors in scaling probabilities.

Recommended Mitigation: Consider ensuring that in the last iteration of the loop, if `boostedTotalProbabilities >= BASE_POINT`, the user is guaranteed a win:

```
- if (winningThreshold < cumulativeProbability) {
+ if (winningThreshold < cumulativeProbability ||
+     boostedTotalProbabilities >= BASE_POINT && i == prizeLen - 1 // last iteration and win is
↪   guaranteed
+ ) {
      selectedPrizeID = prizeIds[i];
```

This change slightly favors the last prize in the list in extremely rare cases, but given that this situation is already highly improbable, this trade-off is reasonable.

Linea: Fixed in commit b32e038

Cyfrin: Verified.

7.2.2 Users can select higher-value NFTs by delaying prize claims

Description: When a user wins, the contract only tracks that they have won a specific `prizeID` in `Spin::_fulfillRandomness`:

```
    if (winningThreshold < cumulativeProbability) {
        selectedPrizeId = localPrizeIds[i];

        // ...
        break;
    }
}

userToPrizesWon[user][selectedPrizeId] += 1;
```

However, when a user claims their prize, if the prize is an NFT, the contract simply assigns them the last available NFT in the list in `Spin::_transferPrize`:

```
uint256 tokenId = prize.availableERC721Ids[
    prize.availableERC721Ids.length - 1
];
```

Since NFTs are non-fungible, each `tokenId` represents a unique item, meaning that a user who wins can wait to claim their prize until the highest-value NFT remains in the collection. This allows them to strategically claim the best available token, potentially at the expense of users who claim their prizes immediately.

Impact: Users could delay claiming to secure a more valuable NFT from a collection, while other users who claim immediately may unknowingly receive lower-value tokens. This could create an unfair advantage for informed users who understand the mechanics of prize allocation.

Recommended Mitigation: There is no perfect solution, as all potential fixes come with trade-offs. One approach would be to assign a specific NFT at the time of winning in `_fulfillRandomness`. However, this would require tracking both which NFTs each user has won and which remain available, significantly increasing state complexity and gas costs.

Instead, the protocol should be aware of this issue and ensure that NFTs within each prize category have similar values. If a collection includes NFTs with widely varying values, they should be added as separate prizes, ensuring fairer distribution and preventing users from gaming the system.

Linea: Acknowledged. Higher value NFTs should be added as separate prizes.

Cyfrin: Acknowledged.

7.2.3 Probability overflow can bypass `MaxProbabilityExceeded` check

Description: When adding new prizes, the contract includes a check to ensure that the total probability does not exceed 100% in `Spin::_addPrizes#L511-L513`:

```
if (totalProbabilities > BASE_POINT) {
    revert MaxProbabilityExceeded(totalProbabilities);
}
```

However, this check can be bypassed due to how `totalProbabilities` is calculated. The accumulation of probabilities happens in `unchecked` blocks at the following locations:

- First accumulation of individual probability values, `Spin::_addPrizes#L503-L505`:

  ```
  unchecked {
      totalProbIncrease += probability;
  }
  ```

- Final update of `totalProbabilities`, `Spin::_addPrizes#L508-L510`:

  ```
  unchecked {
      totalProbabilities += totalProbIncrease;
  }
  ```

Because both updates occur within `unchecked` blocks, a very large probability value can overflow, effectively bypassing the `MaxProbabilityExceeded` check. This could allow `totalProbabilities` to wrap around and appear valid, even if it exceeds `BASE_POINT`.

Impact: Although this function can only be called by trusted users (e.g., the `CONTROLLER` or `DEFAULT_ADMIN` role), a mistake or a compromised account could still trigger this issue by adding an excessively large probability value. This would cause an overflow, allowing the “MaxProbabilityExceeded check to be bypassed and potentially breaking the integrity of the game by distorting the prize distribution.

Proof of Concept: Add the following test to `Spin.t.sol`:

```
function testUpdateWithMoreThanMaxProba() external {
    MockERC721 nft = new MockERC721("Test NFT", "TNFT");
    nft.mint(address(spinGame), 10);
    nft.mint(address(spinGame), 21);

    ISpinGame.Prize[] memory prizesToUpdate = new ISpinGame.Prize[](2);
    uint256[] memory empty = new uint256[](0);

    uint256[] memory nftAvailable = new uint256[](2);
    nftAvailable[0] = 10;
    nftAvailable[1] = 21;

    prizesToUpdate[0] = ISpinGame.Prize({
        tokenAddress: address(nft),
        amount: 0,
        lotAmount: 2,
        probability: type(uint64).max - 1,
        availableERC721Ids: nftAvailable
    });

    prizesToUpdate[1] = ISpinGame.Prize({
        tokenAddress: address(0),
        amount: 1e18,
        lotAmount: 2,
        probability: 2,
        availableERC721Ids: empty
    });

    vm.prank(admin);
    spinGame.updatePrizes(prizesToUpdate);

    assertEq(spinGame.getPrize(1).probability, type(uint64).max - 1);
}
```

Recommended Mitigation: Consider removing the `unchecked` blocks in both calculations.

Linea: Fixed in commit e840e2f

Cyfrin: Verified.

7.3 Informational

7.3.1 Scaling winningThreshold incorrectly reduces randomness distribution

Description: When a user has a boost that results in a >100% probability of winning, the contract adjusts winningThreshold to match boostedTotalProbabilities in Spin::_fulfillRandomness:

```
uint256 winningThreshold = _randomness % BASE_POINT;

// ...

if (boostedTotalProbabilities > BASE_POINT) {
    winningThreshold =
        (winningThreshold * boostedTotalProbabilities) /
        BASE_POINT;
}
```

The issue here is that _randomness is first scaled down to BASE_POINT before being scaled up to boostedTotalProbabilities. This process reduces the effective randomness (entropy) because some values in the original _randomness range will no longer be represented in the final winningThreshold after scaling. As a result, the final threshold may not be evenly distributed, potentially introducing bias.

Consider applying _randomness directly to boostedTotalProbabilities when the win probability exceeds 100%, ensuring no loss of entropy:

```
  if (boostedTotalProbabilities > BASE_POINT) {
-     winningThreshold =
-         (winningThreshold * boostedTotalProbabilities) /
-         BASE_POINT;

+     winningThreshold = _randomness % boostedTotalProbabilities;
  }
```

This preserves the full randomness range and ensures a more uniform distribution of possible winning thresholds.

Linea: Fixed in commit 37a18ca

Cyfrin: Verified.

7.3.2 Native token transfers lack explicit balance check

Description: One possible prize type is native tokens, represented by tokenAddress = address(0). A user who wins native tokens can claim them in Spin::_transferPrize:

```
if (prize.tokenAddress == address(0)) {
    (bool success, ) = _winner.call{value: prize.amount}("");
    if (!success) {
        revert NativeTokenTransferFailed();
    }
} else {
```

For ERC20 prizes (handled here) and ERC721 prizes (handled here), the contract explicitly checks whether it has a sufficient balance or ownership of the token before proceeding with the transfer.

While the current implementation would still revert if the contract lacks the required native token balance, consider adding an explicit balance check for native tokens as it would provide consistency across all prize types and ensure uniform error messages, improving usability and debugging.

Linea: Fixed in commit 7675766

Cyfrin: Verified.

7.3.3 Race Condition in `updatePrizes` Leading to Unexpected Prizes

Description: The `updatePrizes` function allows modifying the list of available prizes. However, it does not consider ongoing participations where the randomness request has not yet been fulfilled, leaving some participants without an assigned prize ID. This means a participant can initiate a spin, and before the VRF provides the random number, the `updatePrizes` function can be called. As a result, the prize mapping is updated, causing the participant to receive a prize from a different list than the one they originally played for.

Impact: Participants may receive prizes from an updated list rather than the one that was active when they initially participated.

Proof of Concept:

1. A participant initiates a spin.
2. Before the VRF fulfills the randomness request, updatePrizes is called, modifying the prize distribution.
3. The participant then receives a prize from the updated list rather than the expected one.

Recommended Mitigation: Ensure that all pending VRF requests are fulfilled before allowing any updates to the prize list.

Linea: Acknowledged. Acceptable behavior.

Cyfrin: Acknowledged.

7.3.4 Assembly blocks could benefit from `"memory-safe"` annotation

Description: When hashing request data in `Spin::_hashParticipation` and `Spin::_hashClaim`, inline assembly is used to efficiently compute the hash:

```
assembly {
    let mPtr := mload(0x40)
    mstore(
        mPtr,
        0x4635ca970da82693e235d3cdaa3678d42c6824330c48b4135f080d655e54da78 //
        ↪  keccak256("ClaimRequest(address user,uint256 expirationTimestamp,uint64 nonce,uint32
        ↪  prizeId)")
    )
    mstore(add(mPtr, 0x20), _user)
    mstore(add(mPtr, 0x40), _expirationTimestamp)
    mstore(add(mPtr, 0x60), _nonce)
    mstore(add(mPtr, 0x80), _prizeId)
    claimHash := keccak256(mPtr, 0xa0)
}
```

To improve compiler optimizations, consider adding a `memory-safe` annotation to the assembly block:

```
+ assembly ("memory-safe") {
```

Since the assembly block only accesses memory after the free memory pointer (`0x40`), this annotation poses no risk and can allow the Solidity compiler to apply additional optimizations, improving gas efficiency.

Linea: Fixed in commit `b4aaffc`

Cyfrin: Verified.