CANTINA

Riskiit

Security Review

Cantina Managed review by:

R0bert, Lead Security Researcher
Cryptara, Security Researcher

February 27, 2025

Contents

1 Introduction

1.1 About Cantina

Cantina is a security services marketplace that connects top security researchers and solutions with clients. Learn more at cantina.xyz

1.2 Disclaimer

Cantina Managed provides a detailed evaluation of the security posture of the code at a particular moment based on the information available at the time of the review. While Cantina Managed endeavors to identify and disclose all potential security issues, it cannot guarantee that every vulnerability will be detected or that the code will be entirely secure against all possible attacks. The assessment is conducted based on the specific commit and version of the code provided. Any subsequent modifications to the code may introduce new vulnerabilities that were absent during the initial review. Therefore, any changes made to the code require a new security review to ensure that the code remains secure. Please be advised that the Cantina Managed security review is not a replacement for continuous security measures such as penetration testing, vulnerability scanning, and regular code reviews.

1.3 Risk assessment

Severity	Description
Critical	*Must* fix as soon as possible (if already deployed).
High	Leads to a loss of a significant portion (>10%) of assets in the protocol, or significant harm to a majority of users.
Medium	Global losses <10% or losses to only a subset of users, but still unacceptable.
Low	Losses will be annoying but bearable. Applies to things like griefing attacks that can be easily repaired or even gas inefficiencies.
Gas Optimization	Suggestions around gas saving practices.
Informational	Suggestions around best practices or readability.

1.3.1 Severity Classification

The severity of security issues found during the security review is categorized based on the above table. Critical findings have a high likelihood of being exploited and must be addressed immediately. High findings are almost certain to occur, easy to perform, or not easy but highly incentivized thus must be fixed as soon as possible.

Medium findings are conditionally possible or incentivized but are still relatively likely to occur and should be addressed. Low findings a rare combination of circumstances to exploit, or offer little to no incentive to exploit but are recommended to be addressed.

Lastly, some findings might represent objective improvements that should be addressed but do not impact the project's overall security (Gas and Informational findings).

2 Security Review Summary

Ludex simplifies blockchain integration for game developers, allowing players to compete using on-chain assets via the Challenge protocol, all without needing blockchain expertise.

From Feb 15th to Feb 16th the Cantina team conducted a review of riskiit on commit hash ce101c21. The team identified a total of **21** issues:

Issues Found

Severity	Count	Fixed	Acknowledged
Critical Risk	3	3	0
High Risk	2	2	0
Medium Risk	8	6	2
Low Risk	1	1	0
Gas Optimizations	1	1	0
Informational	6	6	0
Total	**21**	**19**	**2**

3 Findings

3.1 Critical Risk

3.1.1 Missing receive Function Preventing Native Liquidity Deposits

Severity: Critical Risk

Context: *(No context files were provided by the reviewer)*

Description: The contract does not implement a `receive` function, which prevents it from directly accepting native token (ETH) deposits outside of function calls. If there is a need to inject liquidity to maintain solvency, such as fulfilling withdrawals or covering liabilities, the absence of this function makes it impossible to send native tokens directly to the contract using a simple transfer (`send`, `transfer`, or `call`).

Without a `receive` function, any direct ETH transfer to the contract will result in a **revert**, restricting the ability to fund the protocol efficiently.

Recommendation: The contract should implement a `receive` function to accept direct native token deposits when needed. This will ensure that the protocol remains solvent and can receive liquidity injections when required. If additional logic is needed (such as tracking deposits), a `fallback` function can also be used with appropriate handling of `msg.value`.

Ludex Labs: Fixed in PR 47.

Cantina Managed: Fixed by adding the `receive` function.

3.1.2 Incorrect Native Token Handling in Deposit Function

Severity: Critical Risk

Context: *(No context files were provided by the reviewer)*

Description: The `deposit` function incorrectly handles native token deposits due to a misunderstanding of Ethereum's low level `call` mechanism. Instead of relying on `msg.value` to capture the sent amount, the function attempts to send an arbitrary `amount` from the contract to itself using a low level `.call{}`. However, the contract does not implement a `receive` or `fallback` function, causing all such transactions to revert.

This issue leads to several critical problems:

1. Guaranteed reversion: Since the contract lacks a `receive` or `fallback` function, any attempt to send native funds using `.call{}` will fail.
2. Incorrect fund transfer logic: The use of `.call{}` suggests an attempt to transfer funds from the contract's balance rather than properly crediting the depositor via `msg.value`, leading to an ineffective deposit mechanism.
3. Potential Value Manipulation: If the logic were functional, it could allow a user to specify an arbitrary `amount` to withdraw from the contract instead of using their own deposited funds, enabling unintended withdrawals.

Proof of Concept:

```
function testDepositAlwaysRevertForNativeToken() public {
    riskiit.deposit{value: 1 ether}(
        address(0),
        100 ether
    );
}
```

Output:

```
Failing tests:
Encountered 1 failing test in test/RiskiitV1_t.t.sol:RiskiitV1_1Test
[FAIL: TransferFailed()] testDepositAlwaysRevertForNativeToken() (gas: 28163)
```

Recommendation: The `deposit` function should directly rely on `msg.value` for native token deposits instead of attempting to send funds to itself. The logic should strictly enforce that `msg.value` matches the expected deposit amount.

Ludex Labs: Fixed in PR 47.

Cantina Managed: Fix ok.

3.1.3 Unbalanced odd-rewards ratio

Severity: Critical Risk

Context: *(No context files were provided by the reviewer)*

Description: The contract's `entropyCallback` function incorrectly assigns the color GREEN to the number 36, despite 36 being a RED number in American roulette. This discrepancy occurs because the function checks `number == 36` and assigns it to green. Given that the contract appears to be using an American-style roulette format, which includes zero (0) and double zero (00), there should be 38 pockets instead of 37.

The core issue stems from an incorrect modulus operation and improper number range handling. Currently, the contract's logic does not correctly differentiate between a European-style roulette (which has 37 numbers, including a single zero) and an American-style roulette (which has 38 numbers, including both zero and double zero). If the roulette system is intended to have 38 numbers, then the modulus operation should use 38, and GREEN should only be assigned to `number == 0` or `number == 37` (representing 00).

Alternatively, if the system uses a 37-pocket roulette, the modulus should be `37`, which ensures that `37 % 37 == 0`, naturally handling the GREEN case without requiring additional checks. However, the current condition `|| number == 36` is incorrect and should be removed regardless, as **36 is RED, not GREEN.

On the other hand, the winning odds to rewards ratio is totally broken as:.

- Green pockets: 0, 36 (2 total).
- Red pockets: 1, 3, 5, 7, 9, 12, 14, 16, 18, 19, 21, 23, 25, 27, 30, 32, 34 (17 total).
- Black pockets: the remaining 18 numbers: 2, 4, 6, 8, 10, 11, 13, 15, 17, 20, 22, 24, 26, 28, 29, 31, 33, 35.

Since there are 37 total pockets (0..36), we have:

- Green probability: 2/37 5.405%.
- Red probability: 17/37 45.946%.
- Black probability: 18/37 48.648%.

The contract sets the following winning multipliers:

- Green: 6x.
- Red: 2x.
- Black: 2x.

We assume a 1-token bet and compute the net expected value for each color:

- Green:
  - Win probability: 2/37.
  - Win payout: 6 x your bet (i.e., +5 net, but 6 total if counting the original stake).
  - Expected return: (2/37)*6 + (35/37) * 0 = 12/37 = 0.3243.
  - Net EV (subtracting your original 1-token stake) = 0.3243 1 = 0.6757 (i.e. 67.57%).
- Red:
  - Win probability: 17/37.
  - Win payout: 2x your bet.
  - Expected return: (17/37)*2 + (20/37) * 0 = 34/37 = 0.9189.
  - Net EV = 0.9189 1 = 0.0811 (i.e. 8.11%).
- Black:

- Win probability: 18/37.
- Win payout: 2 × your bet.
- Expected return: (18/37)*2 + (19/37) * 0 = 35/37 = 0.9730.
- Net EV = 0.9730 1 = 0.027 (i.e. 2.70%).

Observations:

1. Black is actually the best bet with about a 2.70% house edge.
2. Red is significantly worse at about 8.11%.
3. Green is extremely unfavorable at about 67.57%.

Recommendation: The logic in `entropyCallback` should be corrected based on the intended roulette format. If the game follows the American roulette format, then the modulus should be 38, and `number == 37` should be checked for double zero. If the game follows the European roulette format, then the modulus should be 37, and the `|| number == 36` condition should be removed entirely, as it mistakenly classifies a red number as green. However, the latter case can also be used for American roulette format as the `37 % 37` will also result to `0`, those removing `|| number == 36` could be a single required change to accept the format.

On the other hand, consider recalculating or rebalancing the payout multipliers according to your chosen layout (37 or 38 pockets). As shown above, having 2 Green pockets in a 37-slot wheel skews the probabilities severely, resulting in extremely high house edges for Green and inconsistent edges for Red vs. Black. Adjust the distribution or the paytable (e.g. 2× for Red/Black, 35× for single-zero Green, etc.) to achieve your desired odds.

Ludex Labs: Fixed in PR 48.

Cantina Managed: Fix ok. However, the best bets are RED and BLACK now over GREEN. This is because choosing RED or BLACK (2x multiplier) yields a 5.26% house edge, consistent with an American roulette. The 2x multiplier is appropriately adjusted to the winning probability, incorporating a standard house advantage. On the other hand, choosing GREEN (14x multiplier currently) results in a 26.32% house edge, much higher than the 5.26%. A fair multiplier would be 19x (EV = 0), and a standard roulette multiplier would be 18x (house edge = 5.26%). The 14x multiplier is too low, making the GREEN bet less favorable than the others.

3.2 High Risk

3.2.1 Insecure Migration Logic Allowing Preemptive Registration and State Loss

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

Description: The `batchMigratePlayers` function is designed to transfer player data from an old contract but will skip any player that is already registered in the new system if they either have points or their username is already taken. This creates a critical issue where trusted users who should have their historical data migrated could lose their old state entirely if they or someone else registers before the migration is performed.

Since registration can be performed by anyone, an attacker could front-run the migration process by registering all usernames from the previous contract before `batchMigratePlayers` executes. This would prevent rightful users from reclaiming their previous state and could result in the loss of historical balances, rankings, and referral benefits. Furthermore, users who register with a referral will not have zero points, meaning they will also be skipped during migration, further exacerbating the issue.

If a malicious actor front-runs the migration and takes ownership of previous usernames, affected players may have no way to recover their original data, causing permanent loss of progress and disrupting the integrity of the migration.

Recommendation: The `batchMigratePlayers` function should be protected by a `whenPaused` modifier, ensuring that no new registrations or referrals can occur before the migration is completed. Additionally, all public functions that allow user registration or modifications to the `players` mapping, such as `registerPlayer` and `addReferral`, should have a `whenNotPaused` modifier to prevent preemptive actions that could interfere with migration.

To ensure a secure migration process, the contract should start in a paused state upon deployment, preventing any user interaction until the migration is completed. Once the migration is finalized, the contract can be unpaused to allow normal operation, ensuring that all players are properly transitioned with their previous state intact.

Ludex Labs: Fixed in PR 46.

Cantina Managed: Fix ok. The code is now preventing any action until the contract is unpaused. The initial state of the contract is paused.

3.2.2 Unrestricted Username Registration and Sybil Attack Risk

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

Description: The `registerPlayer` function in the contract does not enforce unique registrations per `msg.sender`, allowing an attacker to register multiple usernames with the same address. This causes several problems:

1. Username Sniping & Frontrunning: A malicious actor can preemptively register desirable usernames before legitimate users can do so, blocking access to those names.
2. Spam Registration: Without restrictions, the attacker can flood the system with numerous registrations, preventing others from acquiring common or desirable usernames.
3. Referral Farming: If referral-based incentives exist, an attacker could exploit the lack of controls to generate multiple accounts and extract undue rewards.
4. Lack of Age Verification: The function does not validate the `isOldEnough` parameter, allowing ineligible users to register without restrictions.

Proof of Concept:

```
function testRegisterPlayerSameSenderAndSameUsername() public {
    // Allowed from the same msg.sender
    vm.startPrank(attacker);
    riskiit.registerPlayer("alice", true, "");
    riskiit.registerPlayer("bob", true, "");
    riskiit.registerPlayer("charlie", true, "");
    vm.stopPrank();

    // Will revert
    vm.prank(alice);
    riskiit.registerPlayer("alice", true, "");
}
```

Recommendation: To mitigate these issues, the contract should enforce strict validation for username uniqueness per `msg.sender`. Implementing a backend-signed verification mechanism can help ensure each `username` is registered only by its rightful owner while preventing automated spam registrations. Additionally, age verification should be cryptographically validated rather than relying on an unchecked boolean input.

Ludex Labs: Fixed in PR 49.

Cantina Managed: Fix ok. In order to register, users must now provide a valid `ageSigner` signature that will be given after passing the KYC process.

3.3 Medium Risk

3.3.1 Registration Manipulation in Referral System allows points inflation

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

Description: The `registerPlayer` function is vulnerable to exploitation through automated smart contract interactions. There are no restrictions preventing contract-based registrations, which allows attackers to manipulate the referral system and artificially inflate points. The attack leverages the following weaknesses:

1. Automated Multi-Account Registration -- The attacker can programmatically generate new addresses, whether EOAs or contract-deployed proxies, to repeatedly register under a referral.
2. Gas-Optimized Exploitation -- By using `CREATE2`, an attacker can deploy self-destructing contracts that perform the registration and then remove traces to minimize costs.
3. Unrestricted Referral Usage -- Since there is no limit on how many times a referral can be used, an attacker can continuously earn points by creating new fake accounts.
4. Blacklist Dependency -- While the contract implements a blacklist system, it requires active monitoring and manual intervention, which is inefficient and reactive rather than preventive.

Proof of Concept:

```
contract RegisterPlayerHandler {
    RiskiitV1_1 internal riskiit;

    constructor(RiskiitV1_1 _riskiit) {
        riskiit = _riskiit;
    }

    function registerPlayer(string calldata _username, bool _isOldEnough, string calldata _referrer) external {
        riskiit.registerPlayer(_username, _isOldEnough, _referrer);
    }
}

function testRegerralMultipleRegistrationsInfinitePoints() public {

    vm.startPrank(attacker);
    riskiit.registerPlayer("attacker", true, "");
    vm.stopPrank();

    // With create2 create infinite addresses
    for (uint i = 0; i < 10; i++) {
        // create random string with i
        string memory username = string(abi.encodePacked("random", i));
        new RegisterPlayerHandler(riskiit).registerPlayer(username, true, "attacker");
        console.log("Attacker points: %d", riskiit.getPlayerPoints(attacker));
    }

    // This will succeed as the attacker has inflated its points via
    // fake referrals and registrations
    assertEq(riskiit.getPlayerPoints(attacker), 1000);
}
```

Output:

```
[PASS] testRegerralMultipleRegistrationsInfinitePoints() (gas: 4133870)
Logs:
  Attacker points: 100
  Attacker points: 200
  Attacker points: 300
  Attacker points: 400
  Attacker points: 500
  Attacker points: 600
  Attacker points: 700
  Attacker points: 800
  Attacker points: 900
  Attacker points: 1000
```

Recommendation: To prevent abuse, the contract should restrict registration mechanisms to prevent automated interactions. Implementing a backend-signed authorization system for username registration ensures that only verified users can register, mitigating mass account creation. Additionally, referral usage should have a reasonable limit to prevent infinite loops of self-referrals, and further filtering mechanisms like KYC verification or off-chain analysis should be considered for high-value incentives.

Ludex Labs:

1. For the points reward on referral, the points made from referrals are so small that it actually makes more sense economically to farm points via spinning. The value of the referrals is really only the percentage of loss. So we don't view it as too much of an issue.

2. For the backend-signed auth system, we implemented a backend signing mechanism on user verify age, PR 49.

Cantina Managed: Fix is insufficient. The current solution of a backend-signed system will not prevent this. The same proof of concept will still work as the signature does only check the `msg.sender` and not the `username` or `referral` during registration.

3.3.2 Missing User-Provided Random Number Leading to Potential Bias

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

Description: The contract calls `entropy.requestWithCallback`, but instead of supplying a user-generated random number as required by the Entropy protocol, it uses a predefined variable named `choice`. This choice-based approach does not contribute any actual entropy to the final random number and deviates from the intended commit-reveal mechanism. The Entropy protocol follows an optimized commit-reveal scheme where the final randomness is derived from the combination of the provider's pre-committed random number and a user-provided random input (`xU`). By failing to provide `xU`, the contract makes the final randomness entirely dependent on the provider, allowing potential manipulation.

The risk of this approach is that a malicious or compromised provider could selectively withhold their reveals or rotate their commitments to bias the outcome in their favor. Since the final random number is computed as `r = hash(xi, xU)`, excluding `xU` removes the user's ability to influence the result, weakening the protocol's fairness guarantees. Additionally, providers who can observe transactions in the mempool may strategically adjust their commitments based on anticipated outcomes, further exacerbating the problem.

Recommendation: The contract should ensure that a properly randomized `userRandomNumber` is generated on-chain and passed into `entropy.requestWithCallback` instead of using `choice`. This will uphold the protocol's security guarantees, ensuring that as long as either the user or provider is honest, the resulting number remains unpredictable and fair. Additionally, monitoring mechanisms should be implemented to detect and blacklist providers exhibiting suspicious behavior, such as selectively withholding reveals or attempting to manipulate randomness by exploiting the absence of a user commitment.

Ludex Labs: Fixed in commit b306a970.

Cantina Managed: Fix is insufficient. The proposed solution does not use a random number parameter but rather relies on blockchain data to generate a pseudorandom number. The documentation does not state how strong the randomness should be, but current implementation does follow a better approach as per the protocol documentation. However, the value provided is not random as the used data can be known beforehand.

3.3.3 Inconsistent Pausing Mechanism Leading to Potential Fund Loss

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

Description: The `entropyCallback` function is currently restricted by the `whenNotPaused` modifier, meaning it cannot execute if the contract is paused between a spin request and the callback. Since the user already pays the entropy fee upfront, a pause occurring before the callback execution will cause the function to revert, effectively resulting in a loss of funds without completing the action.

Although the contract includes a recovery mechanism through the `refundGame` function, which allows the admin to issue refunds, this solution relies on manual intervention and does not guarantee timely compensation. There is also a consideration that `refundGame` could be callable by the original game creator, but this functionality would need to be reassessed for security implications before being implemented.

Recommendation: The `whenNotPaused` modifier should be reconsidered for `entropyCallback`, as preventing its execution after a user has already paid the entropy fee can lead to an unfair loss of funds. If pausing the callback is still necessary, alternative solutions should be explored, such as allowing delayed execution once the contract is unpaused. Additionally, an active monitoring system should be in place to detect such edge cases and ensure timely refunds if required.

Ludex Labs: Fixed in commit ff57744b.

Cantina Managed: Fix is insufficient. The `whenNotPaused` modifier was removed.

3.3.4 Re-Entrancy Vulnerability potentially Leading to Contract Deadlock and Permanent Insolvency

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

Description: The `refundGame` function is vulnerable to a re-entrancy attack, which can be exploited to manipulate `totalOnLine` and cause an underflow. This happens because `payable(game.player).call{value: refundAmount}` is executed before updating critical state variables, allowing an attacker to re-enter the contract and invoke functions that modify `totalOnLine` in an unintended way.

Since `refundGame` also uses unchecked arithmetic on `totalOnLine`, an attacker can reduce this value below expected limits, leading to contract insolvency and a permanent deadlock. When `totalOnLine` becomes an enormous value due to underflow, it triggers integrity checks that will always fail, preventing any further gameplay or withdrawals. This creates a situation where no admin action, including `removeLiquidity`, can resolve the issue, effectively bricking the contract forever.

The proof-of-concept (PoC) showcases the worst-case scenario by modifying `ethReferralAmount` directly, demonstrating how the issue can be artificially induced to illustrate the contract's failure state.

Proof of Concept:

```
bool isReceiveCalled = false;

function test_refund_reentrancy_underflow_deadlock() public {

    stdstore
        .target(address(riskiit))
        .sig("ethReferralAmount(address)")
        .with_key(address(this))
        .checked_write(1 ether);

    assertEq(riskiit.ethReferralAmount(address(this)), 1 ether);

    // give riskiit contract liquidity
    vm.deal(address(riskiit), 2 ether);

    riskiit.deposit{value: 2.00 ether}(address(0), 2 ether);

    // on line should be 2
    assertEq(riskiit.getTotalOnLine(), 2 ether);

    uint256 fees = riskiit.getEntropyFee();

    uint64 seq = riskiit.requestSpin{value: fees}(
        RiskiitV1_1.SpinSide.Black,
        2.00 ether
    );

    riskiit.refundGame(seq);

    console.log("Total on line: %d", riskiit.getTotalOnLine());

    // give riskiit contract a lot of liquidity that should be enough to cover
    // the solvency check
    vm.deal(address(riskiit), 10000 ether);

    riskiit.deposit{value: 0.1 ether}(address(0), 0.1 ether);

    ////////////////////////////////////
    // REVERT //////////////////////////
    ////////////////////////////////////
    // play another game, with small bet, this will always revert as the `totalOnLine` is
    // a huge number causing the integrity check to fail for insufficient funds although we have deposited
    ↪  `10000` ether enought to cover the spin liquidity.
    riskiit.requestSpin{value: fees}(
        RiskiitV1_1.SpinSide.Black,
        0.1 ether
    );
}


receive() external payable {
    if (!isReceiveCalled) {
        isReceiveCalled = true;
        riskiit.redeemReferral(address(this));
    }
}
```

Recommendation: The `refundGame` function must be protected with a non-reentrant modifier to prevent external calls from re-entering the contract and modifying critical state variables during execution. Additionally, `totalOnLine` arithmetic should be carefully checked and handled safely to prevent underflows.

To mitigate situations where the contract becomes permanently locked due to an integrity failure, an upgradeability mechanism should be implemented, allowing administrators to migrate the contract to a secure version without requiring full liquidity migration. This would provide a last-resort safeguard against catastrophic failures.

Ludex Labs: Fixed in commit 94bf20a.

Cantina Managed: Fix ok. The `nonReentrant` modifier was added.

3.3.5 Unrestricted Gameplay and Deposits Without Registration

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

Description: The contract allows players to request spins and deposit funds without first registering. This creates a fundamental inconsistency, as the system assumes that only registered players who meet the age requirement should be able to participate. Without enforcing a registration check:

- Players can bypass the registration process and interact with the game directly, ignoring potential restrictions such as the age requirement.
- Since only registered players are expected to meet the `isOldEnough` condition, allowing unregistered users to play means that age restrictions are effectively unenforced.
- The system may rely on player data stored during registration for tracking purposes. If unregistered users are allowed to play, this could result in incorrect tracking of rewards, losses, or other metrics.

Recommendation: The contract should verify that a user is registered before allowing them to deposit or request spins. Since the registration process already includes an age check, enforcing registration as a prerequisite ensures that only eligible players can participate. This can be done by requiring a mapping lookup that confirms the `msg.sender` is registered before proceeding with gameplay or deposits.

Ludex Labs: Fixed in commit af74519.

Cantina Managed: Fix ok.

3.3.6 Incorrect Liquidity Check in requestToken Could Lead to Insolvency

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

Description: The contract's `requestToken` function currently checks liquidity using the condition `totalOnLine + betAmount * _multiplier`, but it does not properly account for the `entropyFee`, which is used as the value for the callback. This creates a misleading assessment of available liquidity since `entropyFee` will be spent in the next transaction and should not be considered part of the contract's solvency. As a result, the protocol might allow bets that exceed the actual available balance, leading to situations where user payouts cannot be fulfilled. If this happens, the contract may become insolvent, breaking the invariant that the contract balance should always be equal to or greater than `totalOnLine`. In edge cases, this could result in unintended reverts or discrepancies in fund distribution, disrupting the protocol's integrity.

Recommendation: The liquidity check should be adjusted to exclude `entropyFee` from the solvency calculation to ensure that the contract always retains enough funds to cover outstanding liabilities. The correct comparison should be `totalOnLine + betAmount * _multiplier > address(this).balance - entropyFee`, preventing situations where available liquidity is overestimated and keeping the protocol solvent.

Ludex Labs: Fixed in commit d37d6ab2.

Cantina Managed: Fix ok. For ETH bets, the current code explicitly subtracts `entropyFee` in the liquidity check.

3.3.7 Incorrect Accounting in forceWithdraw Leading to Liquidity Miscalculation

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

Description: The `forceWithdraw` function does not correctly update `tokenTotalOnLine` or `totalOnLine` after executing a forced withdrawal. This leads to inaccurate tracking of funds, as the withdrawn amount remains counted as part of the available liquidity even though it has already been transferred out of the contract.

By failing to decrement `tokenTotalOnLine` and `totalOnLine`, the protocol incorrectly assumes that more funds are available than actually exist. This can lead to protocol insolvency, where winnings or withdrawals that should be covered by on-chain liquidity may be approved when the contract does not actually hold sufficient funds. Since the system still considers withdrawn amounts as "active" liquidity, future payout calculations will rely on incorrect values, potentially resulting in failed transactions or unintended insolvency.

Recommendation: The `forceWithdraw` function should explicitly decrease `tokenTotalOnLine` and `totalOnLine` by the withdrawn amount to maintain accurate liquidity tracking. Ensuring these values are correctly updated prevents the contract from overestimating its available funds, safeguarding against insolvency risks and maintaining the integrity of liquidity management.

Ludex Labs: Fixed in PR 51.

Cantina Managed: Fix ok.

3.3.8 Default Provider's `endSequenceNumber` Can Lead to Permanent DoS and Sequence Collisions with New Providers

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

Description: In the Pyth Entropy contract, each provider has a finite hash-chain length defined by `endSequenceNumber`. Specifically:

```
uint64 assignedSequenceNumber = providerInfo.sequenceNumber;
if (assignedSequenceNumber >= providerInfo.endSequenceNumber)
    revert EntropyErrors.OutOfRandomness();
providerInfo.sequenceNumber += 1;
```

- As the `RiskItV1_1` contract always uses the same "default provider," eventually `providerInfo.sequenceNumber` will reach `endSequenceNumber`, causing any new requests to revert with `OutOfRandomness()`. This triggers a permanent DoS on new spins unless the `RiskItV1_1` contract updates its provider.
- If then the contract updates to a different provider, it must ensure its `sequenceNumber` starts higher than any previously used sequence number. Otherwise, `RiskItV1_1` might collide with existing `games[sequenceNumber]` reverting with `GameAlreadyResolved` if it sees a repeated `sequenceNumber` from a different provider. Because `RiskItV1_1` indexes games solely by the `sequenceNumber` and doesn't store "(provider, sequenceNumber)" as a composite key, reusing the same sequence number from a new provider leads to conflicts or reverts.

Recommendation: Extend or rotate the provider's hash chain before reaching `endSequenceNumber`. The provider can call `register` with a fresh chain or use a rotation mechanism to avoid the `OutOfRandomness` revert.

Use a unique game key that includes both the provider's address and the sequence number in the `games` mapping, e.g. `games[keccak256(provider, seqNum)]`. That way, different providers can reuse `sequenceNumbers` without colliding.

Ludex Labs: Fixed in PR 50.

Cantina Managed: Fix ok. A unique game key that includes both the provider's address and the sequence number is used now in the `games` mapping.

3.4 Low Risk

3.4.1 Users With 0 Points Can Re-Register and Overwrite Their Profile

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

Description: In the `registerPlayer` function, the contract prevents re-registration only if `players[msg.sender].points != 0`. However, if the user's registration ends with 0 points which happens when a referral is not provided, they remain eligible to call `registerPlayer` again. As a result, a single address can register multiple times with different usernames, overwriting its own Player record each time (provided points stays at zero).

Recommendation: Consider using a separate boolean (e.g. `isRegistered`) that marks the caller's address as registered once it successfully calls the `registerPlayer` function.

Ludex Labs: Fixed in commit 4711e5ba.

Cantina Managed: Fix ok.

3.5 Gas Optimization

3.5.1 Redundant Storage for Native and Token Balances

Severity: Gas Optimization

Context: *(No context files were provided by the reviewer)*

Description: The contract currently maintains two separate storage variables: `tokenTotalOnLine` for ERC-20 tokens and `totalOnLine` for native tokens (ETH). This introduces unnecessary redundancy in tracking on-chain liquidity, as both serve a similar purpose of keeping record of the funds currently in play. Keeping them separate increases storage costs, adds complexity to accounting logic, and increases the likelihood of inconsistencies if updates are not properly synchronized between the two variables.

Since native tokens (ETH) are typically represented in Solidity using `address(0)`, the `tokenTotalOnLine` mapping could be extended to track native tokens under `address(0)` instead of maintaining a separate `totalOnLine` variable. By consolidating these balances into a single storage structure, the contract can improve gas efficiency, simplify logic, and reduce the risk of mismatches in liquidity calculations.

Recommendation: Instead of using a separate `totalOnLine` variable, the contract could store native token balances in `tokenTotalOnLine[address(0)]`. This unification would allow a single, consistent method for tracking all funds in play, improving code maintainability and reducing storage costs while ensuring accurate accounting for both native and ERC-20 tokens.

Ludex Labs: Fixed in PR 55.

Cantina Managed: Fix ok. The code is now using a single storage mapping for tracking the `totalOnLine` for all tokens.

3.6 Informational

3.6.1 Unnecessary Use of `public` Instead of `external` in Functions

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: The contract currently declares functions as `public` when they should be `external`. In Solidity, a `public` function can be called both internally (from within the contract) and externally (by external accounts or contracts). However, if a function is never intended to be used internally, using `public` is unnecessary and can introduce unintended behaviors, including:

1. Increased Gas Costs — `public` functions can be accessed internally, which increases gas usage when the function is mistakenly called from within the contract instead of performing a direct external call.
2. Code Maintainability Issues — Using `public` when `external` is intended can lead to confusion and accidental internal calls, potentially causing logic errors.
3. Unintended Internal Execution — Developers may inadvertently call the function internally, leading to unexpected behaviors if state modifications depend on external execution.

Recommendation: To improve clarity and efficiency, functions that are never meant to be used internally should be explicitly marked as `external`. This ensures they can only be called from outside the contract, preventing internal misuse and reducing gas consumption. Functions that need internal execution should remain `public` or be explicitly refactored with an internal helper function if required.

Ludex Labs: Fixed in PR 53.

Cantina Managed: Fix ok.

3.6.2 Unchecked Balance Leading to Underflow

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: The function responsible for handling spin requests contains a potential underflow issue due to the line `playerDeposits[msg.sender][address(0)] -= neededTransfer;`. If the player's deposit balance is lower than `neededTransfer`, the subtraction will underflow, causing a revert. While Solidity 0.8+ includes automatic underflow protection, allowing the compiler to handle this issue results in a less user-friendly error message and prevents developers from implementing a more informative revert condition.

If a user attempts to request a spin without sufficient deposited funds, the function will fail without a clear explanation, which can lead to confusion. This could also make it harder to debug issues related to incorrect balance tracking, especially when interacting with front-end applications or external services.

Proof of Concept:

```
function test_spin_totalOnLine() public {
    // give riskiit contract liquidity
    vm.deal(address(riskiit), 3 ether);

    vm.startPrank(alice);

    riskiit.deposit{value: 2.00 ether}(address(0), 2 ether);

    // on line should be 2
    assertEq(riskiit.totalOnLine(), 2 ether);

    // set mock fee
    mockEntropy.setFee(0.01 ether);

    riskiit.requestSpin{value: 1.00 ether}(
        RiskiitV1_1.SpinSide.Black,
        3.00 ether
    );
}
```

Recommendation: Before performing the subtraction, the contract should explicitly check whether `playerDeposits[msg.sender][address(0)]` is at least equal to `neededTransfer`. If the balance is insufficient, a custom revert message should be used to provide a clear and informative error. This ensures better user feedback and prevents unexpected failures due to underflows.

Ludex Labs: Fixed in PR 56.

Cantina Managed: Fix ok. A check was added with a custom revert error.

3.6.3 Lack of Upgradeability

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: The contract currently lacks an upgradeability mechanism, meaning that in the event of a severe exploit or logic failure, the only available recovery option is liquidity migration, which can be inefficient, slow, or even impossible if the contract becomes locked due to an integrity failure. Without an upgrade path, any major issue—such as insolvency caused by incorrect accounting, underflows, or broken liquidity calculations—could render the entire system permanently unusable, with no direct way to patch the contract while preserving user funds.

Recommendation: To ensure long-term stability and security, the contract should be designed with an upgradeability mechanism, such as a proxy pattern (e.g., OpenZeppelin's Transparent Upgradeable Proxy) or an emergency governance-controlled upgrade system. This would allow necessary patches to be deployed without requiring full liquidity migration, protecting both user funds and protocol integrity.

Ludex Labs: Fixed in commit b8881116.

Cantina Managed: Fix ok. The code is now deployed with upgradeable dependencies and using a proxy.

3.6.4 Incompatibility with non-standard ERC20 tokens

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: The `RiskiitV1_1` contract uses direct calls to `transfer` and `transferFrom` functions of ERC20 tokens without proper checks on the return values or safeguards against non-standard implementations. This approach assumes all ERC20 tokens strictly follow the standard interface, which is not always the case.

Recommendation: Consider using SafeERC20.safeTransfer() instead of `transfer()` to perform ERC20 transfers.

Ludex Labs: Fixed in PR 52.

Cantina Managed: Fix ok.

3.6.5 Unnecessary Explicit Initialization of Local Variables

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: Within the `registerPlayer` function, the code explicitly initializes local variables to their default values:

```
address referrer = address(0);
uint256 points = 0;
```

Since Solidity automatically zero-initializes local variables of those types, this is redundant. Explicitly setting them to `address(0)` slightly increases the gas usage.

Recommendation: Remove explicit assignments to zero, letting Solidity's default initialization handle them. For example:

```
address referrer; // no need = address(0)
uint256 points;   // no need = 0
```

Ludex Labs: Fixed in PR 57.

Cantina Managed: Fix ok.

3.6.6 Redundant Balance/Allowance Checks Before transferFrom

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: In the `requestTokenSpin` function, the code explicitly checks:

```
if (
    tokenContract.allowance(msg.sender, address(this)) <
    neededTransfer
) revert InsufficientAllowance();

if (tokenContract.balanceOf(msg.sender) < neededTransfer)
    revert InsufficientBalance();
```

before calling `tokenContract.transferFrom`. However, ERC20 tokens will revert if transferFrom is called without sufficient allowance or balance. Therefore, these checks are redundant because the transfer itself will fail in those scenarios.

Recommendation: Consider removing these explicit checks. On the other hand, use SafeERC20.safeTransfer() instead of `transfer()` to perform ERC20 transfers.

Ludex Labs: Fixed in PR 52.

Cantina Managed: Fix ok.