Zellic

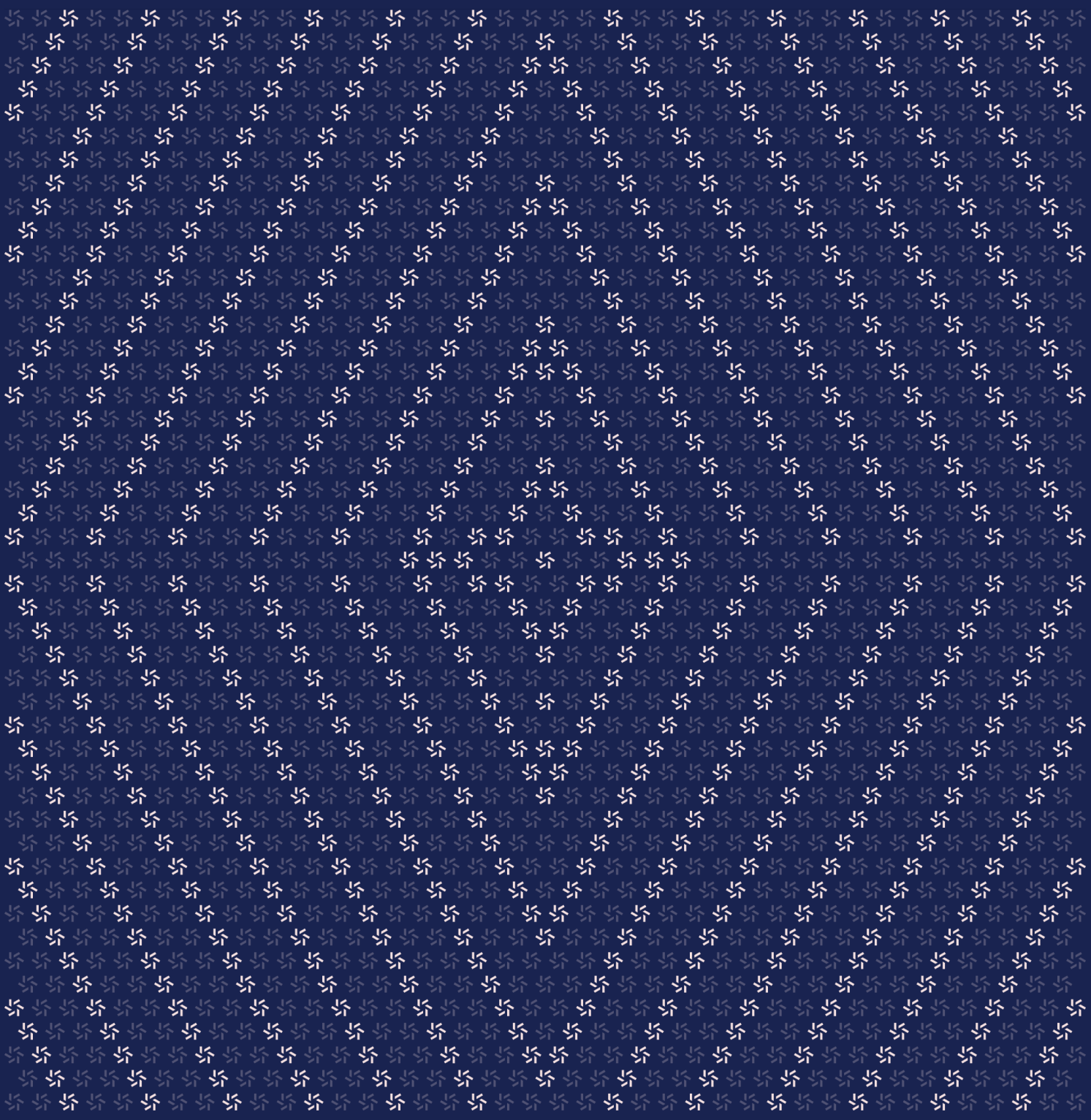
Prepared for
Hyperbeat contributors
Hyperbeat Foundation

Prepared by
Seunghyeon Kim
Jinseo Kim
Zellic

January 26, 2025

BeatPotV2

Smart Contract Security Assessment



Contents

About Zellic

Zellic is a vulnerability research firm with deep expertise in blockchain security. We specialize in EVM, Move (Aptos and Sui), and Solana as well as Cairo, NEAR, and Cosmos. We review L1s and L2s, cross-chain protocols, wallets and applied cryptography, zero-knowledge circuits, web applications, and more.

Prior to Zellic, we founded the #1 CTF (competitive hacking) team ↗ worldwide in 2020, 2021, and 2023. Our engineers bring a rich set of skills and backgrounds, including cryptography, web security, mobile security, low-level exploitation, and finance. Our background in traditional information security and competitive hacking has enabled us to consistently discover hidden vulnerabilities and develop novel security research, earning us the reputation as the go-to security firm for teams whose rate of innovation outpaces the existing security landscape.

For more on Zellic's ongoing security research initiatives, check out our website zellic.io ↗ and follow @zellic_io ↗ on Twitter. If you are interested in partnering with Zellic, contact us at hello@zellic.io ↗.

1. Overview

1.1. Executive Summary

Zellic conducted a security assessment for Hyperbeat Foundation from December 9th to December 11th, 2025. During this engagement, Zellic reviewed BeatPotV2's code for security vulnerabilities, design issues, and general weaknesses in security posture.

1.2. Goals of the Assessment

In a security assessment, goals are framed in terms of questions that we wish to answer. These questions are agreed upon through close communication between Zellic and the client. In this assessment, we sought to answer the following questions:

- Could the jackpot winner selection be manipulated?
- Could an attacker drain the vaults?
- Could an attacker cause a denial of service to the contract?
- Is the implementation of the jackpot math sound?

1.3. Non-goals and Limitations

We did not assess the following areas that were outside the scope of this engagement:

- Front-end components
- Infrastructure relating to the project
- Key custody

Due to the time-boxed nature of security assessments in general, there are limitations in the coverage an assessment can provide.

1.4. Results

During our assessment on the scoped BeatPotV2 contracts, we discovered 11 findings. No critical issues were found. Four findings were of high impact, two were of medium impact, three were of low impact, and the remaining findings were informational in nature.

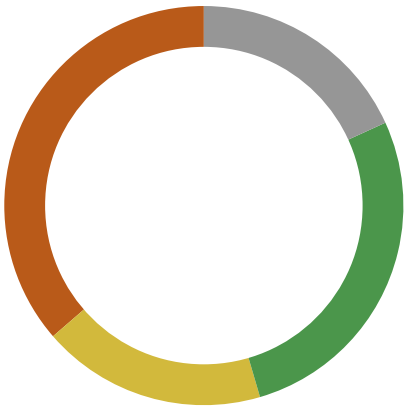
Additionally, Zellic recorded its notes and observations from the assessment for the benefit of Hyperbeat Foundation in the Discussion section (4. ↗).

Based on the number of severe findings uncovered during the audit, it is our opinion that the project is not yet ready for production. We strongly advise a comprehensive reassessment before

deployment to help identify any potential issues or vulnerabilities introduced by necessary fixes or changes. We also recommend adopting a security-focused development workflow, including (but not limited to) augmenting the repository with comprehensive end-to-end tests that achieve 100% branch coverage using any common, maintainable testing framework, thoroughly documenting all function requirements, and training developers to have a security mindset while writing code.

Breakdown of Finding Impacts

Impact Level	Count
Critical	0
High	4
Medium	2
Low	3
Informational	2



2. Introduction

2.1. About BeatPotV2

Hyperbeat Foundation contributed the following description of BeatPotV2:

> Hyperbeat is building infrastructure for on-chain yield, liquidity, and payments powered exclusively by Hyperliquid. Hyperbeat utilizes smart contracts to offer a fully on-chain experience for Accounts management, fiat payments, credit, savings and trading.

2.2. Methodology

During a security assessment, Zellic works through standard phases of security auditing, including both automated testing and manual review. These processes can vary significantly per engagement, but the majority of the time is spent on a thorough manual review of the entire scope.

Alongside a variety of tools and analyzers used on an as-needed basis, Zellic focuses primarily on the following classes of security and reliability issues:

Basic coding mistakes. Many critical vulnerabilities in the past have been caused by simple, surface-level mistakes that could have easily been caught ahead of time by code review. Depending on the engagement, we may also employ sophisticated analyzers such as model checkers, theorem provers, fuzzers, and so on as necessary. We also perform a cursory review of the code to familiarize ourselves with the contracts.

Business logic errors. Business logic is the heart of any smart contract application. We examine the specifications and designs for inconsistencies, flaws, and weaknesses that create opportunities for abuse. For example, these include problems like unrealistic tokenomics or dangerous arbitrage opportunities. To the best of our abilities, time permitting, we also review the contract logic to ensure that the code implements the expected functionality as specified in the platform's design documents.

Integration risks. Several well-known exploits have not been the result of any bug within the contract itself; rather, they are an unintended consequence of the contract's interaction with the broader DeFi ecosystem. Time permitting, we review external interactions and summarize the associated risks: for example, flash loan attacks, oracle price manipulation, MEV/sandwich attacks, and so on.

Code maturity. We look for potential improvements in the codebase in general. We look for violations of industry best practices and guidelines and code quality standards. We also provide suggestions for possible optimizations, such as gas optimization, upgradability weaknesses, centralization risks, and so on.

For each finding, Zellic assigns it an impact rating based on its severity and likelihood. There is no hard-and-fast formula for calculating a finding's impact. Instead, we assign it on a case-by-case basis based on our judgment and experience. Both the severity and likelihood of an issue affect

its impact. For instance, a highly severe issue's impact may be attenuated by a low likelihood. We assign the following impact ratings (ordered by importance): Critical, High, Medium, Low, and Informational.

Zellic organizes its reports such that the most important findings come first in the document, rather than being strictly ordered on impact alone. Thus, we may sometimes emphasize an "Informational" finding higher than a "Low" finding. The key distinction is that although certain findings may have the same impact rating, their *importance* may differ. This varies based on various soft factors, like our clients' threat models, their business needs, and so on. We aim to provide useful and actionable advice to our partners considering their long-term goals, rather than a simple list of security issues at present.

Finally, Zellic provides a list of miscellaneous observations that do not have security impact or are not directly related to the scoped contracts itself. These observations — found in the Discussion (4. ↗) section of the document — may include suggestions for improving the codebase, or general recommendations, but do not necessarily convey that we suggest a code change.

2.3. Scope

The engagement involved a review of the following targets:

BeatPotV2 Contracts

Type	Solidity
Platform	EVM-compatible
Target	BeatPot
Repository	https://github.com/0xhyperbeat/BeatPot ↗
Version	`86ae43e1a4415473073089c431285a14e2fd83e7`
Programs	`BeatPotV2` `BeatPotWithdrawalQueue`

2.4. Project Overview

Zellic was contracted to perform a security assessment for a total of six person-days. The assessment was conducted by two consultants over the course of three calendar days.

Contact Information

The following project manager was associated with the engagement:

Pedro Moura
Engagement Manager
pedro@zellic.io ↗

The following consultants were engaged to conduct the assessment:

Seunghyeon Kim
Engineer
seunghyon@zellic.io ↗

Jinseo Kim
Engineer
jinseo@zellic.io ↗

2.5. Project Timeline

The key dates of the engagement are detailed below.

December 9, 2025	Kick-off call
December 9, 2025	Start of primary review period
December 11, 2025	End of primary review period

3. Detailed Findings

3.1. Selecting winner spends up to eight million gas in the default configuration

Target	BeatPotV2		
Category	Optimization	**Severity**	High
Likelihood	High	**Impact**	High

Description

The BeatPotV2 contract allows players and backers join the lottery game. Players may join the game by purchasing entries per round, and backers may join the game by depositing their tokens to the contract.

When the players win the round (or the funds of backers are not used for the round because the total amount the backers deposited is less than the total amount players entered), one of the players who would solely receive the reward pot would be selected. The winner is randomly chosen with the weighted probability that is proportional to the amount the player entered:

```
function findWinnerFromPlayers(
    uint256 winningEntry
) private view returns (address) {
    uint256 cumulativeEntriesBps = 0;
    for (uint256 i = 0; i < playersByRoundId[roundId].length; i++) {
        address playerAddress = playersByRoundId[roundId][i];
        cumulativeEntriesBps
    += playerEntriesByRoundId[roundId][playerAddress];
        if (winningEntry <= cumulativeEntriesBps) {
            return playerAddress;
        }
    }
    // No winner found, this should never happen
    return fallbackWinner;
}
```

Note that this function runs with the time complexity $O(n)$, where n is the number of players. In the default configuration, which allows a number of players up to 1,500, this function may spend up to eight million gas.

Impact

The callback gas limit must be set above eight million in order to ensure the callback from Pyth Entropy does not revert when players win (the default minimum gas limit is 500,000 in HyperEVM

as of the time of writing). Note that Pyth Entropy does not refund unspent gas.

The maximum gas per block on HyperEVM is 30 million per large block, which the entropy callback transaction may be difficult to fit in.

Recommendations

Consider storing the amount of entries players buy in a more efficient data structure, such as a Fenwick tree.

Remediation

This issue has been acknowledged by Hyperbeat Foundation, and a fix was implemented in commit `feafae82` ↗.

Reviewing the fix, it is noted that the comment of the `_updateBIT` function mentions that `delta` can be negative, which is actually impossible since `delta` is unsigned integer. This issue has been acknowledged by Hyperbeat Foundation.

3.2. Value of shares inflates over time, eventually causing an inoperable deposit function

Target	BeatPotV2		
Category	Business Logic	**Severity**	High
Likelihood	High	**Impact**	High

Description

If the total amount a backer deposits is greater than the total amount players enter for the round, the winner may be the backers or one of the players. Backers equally share the win/loss of the round.

Internally, the BeatPotV2 contract is an ERC-20 contract, and its token represents the amount of share a backer holds; when a backer deposits their underlying tokens to the contract, the contract mints the corresponding shares. Shares can later be withdrawn as the underlying tokens. The win/loss of the round is accumulated to the total deposited amount of tokens, and the value of a share (which would affect the exchange rate between token and share) is calculated as the total deposited amount of tokens divided by the total amount of shares.

The following code implements the deposit function:

```
/// @notice Deposit capital as a backer or add to existing position
/// @dev Capital and exposure rate update immediately; allocation occurs at
    next round
/// @dev Supports fee-on-transfer tokens, refunds excess, floors to entry price
/// @param value Token amount to deposit
function backerDeposit(uint256 value) public {
    // (...)

    uint256 balanceBefore = token.balanceOf(address(this));
    token.safeTransferFrom(msg.sender, address(this), value);
    uint256 balanceAfter = token.balanceOf(address(this));
    uint256 actualReceived = balanceAfter - balanceBefore;
    uint256 flooredValue = (actualReceived / entryPrice) * entryPrice;

    // (...)

    uint256 shares;
    if (backerVaultTotal == 0) {
        shares = flooredValue;
    } else {
```

```
            shares = flooredValue * totalSupply() / backerVaultTotal;
        }
        _mint(msg.sender, shares);
        backerVaultTotal += flooredValue;

        uint256 remainder = actualReceived - flooredValue;
        if (remainder > 0) {
            token.safeTransfer(msg.sender, remainder);
        }

        emit BackerDeposit(msg.sender, flooredValue);
        emit BackerLiquidityTotalUpdated(backerVaultTotal);
    }
```

The initial exechange rate would be 1:1, and this rate may be changed by the result of games. The rate may be reset to 1:1 if all deposited tokens are withdrawn.

As explained in Discussion point 4.2. ↗, the game is slightly biased to backers, resulting in this exchange rate inflating over time. If this rate exceeds the amount to deposit (with decimals), the `backerDeposit` function will mint zero shares because the deposited amount would be truncated during the calculation `flooredValue * totalSupply() / backerVaultTotal`.

A malicious user might accelerate this in the early phase of the contract (specifically when there is no deposit) by depositing a very small amount of tokens and entering the game as a player paying a lot of tokens whose fee would be accumulated to the total deposited amount of tokens. This may or may not be feasible depending on the entry price and the decimal of underlying tokens.

Impact

If this rate exceeds the amount to deposit (with decimals), the `backerDeposit` function will mint zero shares because the deposited amount would be truncated during the calculation `flooredValue * totalSupply() / backerVaultTotal`. Assuming the EV of a backer is 6% per round (based on the calculation in Discussion point 4.2. ↗, it may differ by the ratio between the total amount that is backer deposited and the total amount players paid), `backerVaultTotal / totalSupply()` would exceed 10^{18} after 711 rounds, which may visibly affect the behavior of the contract, such as the `backerDeposit` function minting zero shares.

Recommendations

Consider refactoring the way the deposit vault is implemented.

Remediation

Hyperbeat Foundation notified us that they decided not to fix this issue as of now, considering that it is uncertain that continued active participation, which is necessary to trigger this issue, would

occur over several hundreds of rounds. Hyperbeat Foundation stated that they plan to closely monitor the deployed contract and take appropriate actions if this issue surfaces. Meanwhile, mitigations which limit the impact of this issue were implemented in the following commits:

- 95dcb5e4
- 7ea2aad6
- 1f2be006
- bfad155a

3.3. The `setEntryPrice` function should not be called during the round

Target	BeatPotV2		
Category	Coding Mistakes	**Severity**	High
Likelihood	Medium	**Impact**	High

Description

We observed that the `setEntryPrice` function allows updating the entry price even while a round is currently in progress. The `entryPrice` parameter is a key component in calculating the winning probability range for players. Modifying this value after players have already entered a round alters the probability denominator relative to their purchased entries.

```
function setEntryPrice(uint256 _newEntryPrice) external onlyOwner {
    entryPrice = _newEntryPrice;
}
```

Impact

This could lead to inconsistent winning probabilities for participants within the same round. To ensure a predictable and consistent gaming environment, the entry price should remain constant throughout the duration of a round.

Recommendations

We recommend adding a check to ensure that `setEntryPrice` can only be called when the round is not active or when there are no active players participating in the current round.

Remediation

This issue has been acknowledged by Hyperbeat Foundation, and a fix was implemented in commit a25eb06a ↗.

Reviewing the fix, it is noted that admins will not be able to change the entry price through the `setEntryPrice` function when there is a user who enters the game right after the next round starts. This issue has been acknowledged by Hyperbeat Foundation.

3.4. Withdrawal-logic denial of service by fee-on-transfer tokens

Target	BeatPotV2, BeatPotWithdrawalQueue		
Category	Coding Mistakes	**Severity**	High
Likelihood	Medium	**Impact**	High

Description

The comment of the BeatPotV2 contract mentions that it supports fee-on-transfer (FoT) tokens:

```
// ========== PUBLIC FUNCTIONS ========== //

/// @notice Deposit capital as a backer or add to existing position
/// @dev Capital and exposure rate update immediately; allocation occurs at
    next round
/// @dev Supports fee-on-transfer tokens, refunds excess, floors to entry price
/// @param value Token amount to deposit
function backerDeposit(uint256 value) public {
    // (...)
}
```

However, it fails to correctly account for the transfer tax during the withdrawal distribution process, leading to an insolvency state in the BeatPotWithdrawalQueue contract.

In the `_processWithdrawalsForRound` function, the contract performs the following sequence:

1. It calculates `tokenAmount` based on the shares to be withdrawn.
2. It calls `withdrawalQueue.updateTotalWithdrawalAmount(..., tokenAmount)` to record this amount in the queue's internal ledger.
3. It calls `token.safeTransfer(..., tokenAmount)` to send the funds.

If the token charges a transfer fee (e.g., 1%), the WithdrawalQueue contract receives 99% of `tokenAmount`, but its internal `totalWithdrawalAmount` state variable is updated by 100% of `tokenAmount`.

As a result, the `claimWithdrawal` function will fail to transfer the correct amount of tokens to the user, causing withdrawal insolvency.

Impact

This creates a discrepancy between the actual token balance and the internal accounting. Users claim their withdrawals via `claimWithdrawal`, which calculates their payout based on the inflated internal accounting (`totalWithdrawalAmount`).

The first set of users will successfully withdraw their full expected amounts (effectively draining the 1% deficit from the pool). However, the last set of users will inevitably face a transaction revert (denial of service) when they attempt to claim, because the WithdrawalQueue contract will physically run out of tokens before settling all liabilities.

Recommendations

Adopt the balance-check pattern to handle FoT tokens correctly. The BeatPotV2 contract should measure the balance increase in the WithdrawalQueue contract to determine the actual transferred amount.

Remediation

This issue has been acknowledged by Hyperbeat Foundation. Hyperbeat Foundation notified us that they do not plan to use fee-on-transfer tokens with these contracts even though it is documented that the contract supports fee-on-transfer tokens.

3.5. Minimum amount of backer deposit is not enforced

Target	BeatPotV2		
Category	Coding Mistakes	**Severity**	Medium
Likelihood	High	**Impact**	Medium

Description

The BeatPotV2 contract defines the variable that stores the minimum amount of backer deposit, its setter method, and the relevant error:

```
contract BeatPotV2 is
    Initializable,
    Ownable2StepUpgradeable,
    UUPSUpgradeable,
    IEntropyConsumer,
    ERC20Upgradeable
{
    // (...)
    /// @notice Minimum deposit required for new backers (floored to entry
    price)
    uint256 public minBackerDeposit;
    // (...)
    /// @notice Thrown when new backer deposit below minimum threshold
    error BackerDepositBelowMinimum();
    // (...)
    /// @notice Update minimum backer deposit amount (admin only)
    function setMinBackerDeposit(uint256 _minDeposit) external onlyOwner {
        minBackerDeposit = _minDeposit;
    }
    // (...)
}
```

However, this minimum amount of backer deposit is not enforced anywhere in the contract.

Impact

A backer may deposit lower than the minimum amount of backer deposit set by the admin.

Recommendations

Consider adding a check enforcing this condition.

Remediation

This issue has been acknowledged by Hyperbeat Foundation, and a fix was implemented in commit cda020f8 ↗.

3.6. Withdrawal of fee-on-transfer tokens requires paying a fee twice

Target	BeatPotV2, BeatPotWithdrawalQueue		
Category	Business Logic	**Severity**	Medium
Likelihood	High	**Impact**	Medium

Description

When a backer withdraws their deposit from a contract, the backer invokes the `requestWithdrawal` function of the BeatPotWithdrawalQueue contract, transferring their shares to the BeatPotWithdrawalQueue contract. This contract burns these shares and receives underlying tokens from the BeatPotV2 contract when a round ends. Then the backer may receive their tokens by invoking the `claimWithdrawal` function of the BeatPotWithdrawalQueue contract.

However, if the underlying token is an FoT token, the transfer fee would be deducted twice from the tokens to be withdrawn during the withdrawal process because transferring tokens takes place twice (BeatPotV2 → BeatPotWithdrawalQueue → backer wallet).

Impact

A backer would pay the token-transfer fee more than the one time systematically required.

Recommendations

Consider refactoring the contract so that the withdrawal process involves a single token transfer.

Remediation

This issue has been acknowledged by Hyperbeat Foundation. Hyperbeat Foundation notified us that they do not plan to use fee-on-transfer tokens with these contracts even though it is documented that the contract supports fee-on-transfer tokens.

3.7. The `ProtocolFeeWithdrawal` event is always emitted with a zero amount

Target	BeatPotV2		
Category	Coding Mistakes	**Severity**	Low
Likelihood	High	**Impact**	Low

Description

The following function transfers the accumulated protocol fee to the configured receiver and emits the relevant event:

```
/**
* @notice Withdraw protocol fees to configured fee address (admin
    onlybackerDeposit)
* @dev Requires protocolFeeAddress to be set, resets claimable before transfer
*/
function withdrawProtocolFees() external onlyOwner {
    if (protocolFeeClaimable == 0) revert NoProtocolFeesToWithdraw();

    uint256 transferProtocolFeeAmount = protocolFeeClaimable;
    protocolFeeClaimable = 0;

    if (protocolFeeAddress == address(0)) revert ProtocolFeeAddressNotSet();

    token.safeTransfer(protocolFeeAddress, transferProtocolFeeAmount);

    emit ProtocolFeeWithdrawal(protocolFeeClaimable);
}
```

However, it should be noted that the `protocolFeeClaimable` variable would be zero when it is included in the `ProtocolFeeWithdrawal` event because the code sets the variable to zero.

Impact

The `ProtocolFeeWithdrawal` event will be always emitted with zero amount, which may confuse users and developers about its behavior.

Recommendations

Consider changing the code to properly emit the amount of claimed protocol fees.

Remediation

This issue has been acknowledged by Hyperbeat Foundation, and a fix was implemented in commit a03e2cdf ↗.

3.8. The `RoundExecuted` event is emitted with zero purchased entries when a backer wins

Target	BeatPotV2		
Category	Coding Mistakes	**Severity**	Low
Likelihood	High	**Impact**	Low

Description

When a round is finished, the `RoundExecuted` event is emitted:

```
/// @notice Emitted when round execution completes
/// @param time Timestamp of round completion
/// @param winner Winning player address (address(0) if backers won)
/// @param winningEntry The winning entry number
/// @param prizeAmount Total prize amount awarded
/// @param entriesPurchasedTotalBps  total entries for the round in basis
    points
/// @param roundId Round id
event RoundExecuted(
    uint256 time,
    address winner,
    uint256 winningEntry,
    uint256 prizeAmount,
    uint256 entriesPurchasedTotalBps,
    uint256 roundId
);

// (...)

// Determines a winner, and adjusts backer capital/allocation accordingly
function selectWinnerAndRebalance(bytes32 randomNumber) private {
    // (...)

    // No entries bought
    if (playersByRoundId[roundId].length == 0) {
        emit RoundExecuted(lastRoundEndTime, address(0), 0, backerVaultTotal,
    0, roundId);
        _processWithdrawalsForRound();
        roundId++;
        return;
```

```
    }

    // (...)

    if (playerVaultTotal >= backerVaultTotal) {
        // Round is fully funded by players, so winner gets the player vault
    and backers get the backer vault
        // (...)
        emit RoundExecuted(
            lastRoundEndTime,
            lastWinnerAddress,
            winningEntry,
            prizeAmount,
            totalEntriesByRoundId[roundId],
            roundId
        );
    } else {
        // Round is not fully funded by players, i.e. partially funded by
    backers
        // (...)
        if (winningEntry <= totalEntriesByRoundId[roundId]) {
            // Round is won by a player, so winner gets the backer vault and
    backers get the player vault (but lose the backer vault)
            // (...)
            emit RoundExecuted(
                lastRoundEndTime,
                lastWinnerAddress,
                winningEntry,
                prizeAmount,
                totalEntriesByRoundId[roundId],
                roundId
            );
        } else {
            // Round is won by backers, so backers get both the player vault
    and backer vault
            // (...)
            emit RoundExecuted(
                lastRoundEndTime,
                lastWinnerAddress,
                winningEntry,
                backerVaultTotal,
                0,
                roundId
            );
        }
    }
```

```
    // (...)
}

// (...)
```

When a round is won by backers, the `entriesPurchasedTotalBps` would be always set to zero, even though the total purchased entry may be nonzero.

Impact

The `RoundExecuted` event will be emitted with zero amount of total purchased entries when a backer wins, which may confuse users and developers about its behavior.

Recommendations

Consider changing the code to properly emit the amount of total purchased entries.

Remediation

This issue has been acknowledged by Hyperbeat Foundation, and a fix was implemented in commit `d194a76f` ↗.

3.9. Flawed minimum-fee margin-check logic

Target	BeatPotV2		
Category	Coding Mistakes	**Severity**	Low
Likelihood	Medium	**Impact**	Low

Description

The following code allows the admin to set the total fee rate and the referral fee rate:

```
/**
 * @notice Update referral fee in basis points (admin only)
 * @dev Cannot exceed total fee basis points
 */
function setReferralFeeBps(uint256 _referralFeeBps) external onlyOwner {
    if (_referralFeeBps > feeBps) revert ReferralFeeBpsExceedsFee();
    referralFeeBps = _referralFeeBps;
}

/**
 * @notice Update total fee in basis points (admin only)
 * @dev Maximum 80%, must maintain 5% margin above referral fee
 */
function setFeeBps(uint256 _feeBps) external onlyOwner {
    if (_feeBps > 8000) revert FeeBpsExceedsMaximum();
    if (referralFeeBps + 500 > _feeBps) revert InsufficientFeeBpsMargin();
    feeBps = _feeBps;
}
```

The `setFeeBps` function ensures that `feeBps - referralFeeBps >= 500` and raises an error if it is not the case. However, the `setReferralFeeBps` function does not check this condition.

Impact

If the admin invokes the `setReferralFeeBps` after invoking the `setFeeBps` function, they can make the contract fail to satisfy `feeBps - referralFeeBps >= 500`, which is inconsistent with the apparent business logic.

Recommendations

Consider ensuring the same condition on the `setReferralFeeBps` function.

Remediation

This issue has been acknowledged by Hyperbeat Foundation, and a fix was implemented in commit `545ef375`.

3.10. Unused errors

Target	BeatPotV2		
Category	Code Maturity	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

The `BackerAllocationNotZero`, `BackerExposureRateNotZero`, `BackerIndexNotFound`, `BackerNotActive`, and `MaxBackerLimitReached` errors are defined as the following:

```
contract BeatPotV2 is
    Initializable,
    Ownable2StepUpgradeable,
    UUPSUpgradeable,
    IEntropyConsumer,
    ERC20Upgradeable
{
    // (...)
    /// @notice Thrown when attempting to deactivate backer with non-zero
    allocation
    error BackerAllocationNotZero();
    // (...)
    /// @notice Thrown when attempting to deactivate backer with non-zero
    exposure rate
    error BackerExposureRateNotZero();
    /// @notice Thrown when backer address not found in active list
    error BackerIndexNotFound();
    /// @notice Thrown when operation requires active backer status
    error BackerNotActive();
    // (...)
    /// @notice Thrown when active backer count reaches limit
    error MaxBackerLimitReached();
    // (...)
```

None of these errors are used in the codebase. Considering the comments of each errors do not match the behavior of the contract regarding backer operations, it seems the codebase was refactored to remove the relevant business logic.

Note that the `BackerDepositBelowMinimum` error is also unused. However, we believe that this should be resolved by implementing a check that could raise this error; see Finding 3.5. ↗.

Impact

Unused code may make an impression that such business logic exists in the project, even though that is not the case.

Recommendations

Consider removing these errors from the codebase.

Remediation

This issue has been acknowledged by Hyperbeat Foundation, and a fix was implemented in commit 9e09b6eb ↗.

3.11. Centralization risk

Target	BeatPotV2		
Category	Business Logic	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

The owner may perform privileged operations that may affect important aspects of the business logic. Especially, the owner may

- upgrade the contract without restriction,
- forcefully release the round lock, and
- change the withdrawal-queue address.

Impact

If the owner account is compromised by a malicious attacker, all assets in the contract may be immediately stolen by the attacker.

Recommendations

Consider implementing additional security measures on upgrading the contract and changing the withdrawal-queue address, such as putting the function behind a multi-signature account and timelock.

Also consider adding a sanity check on forcefully releasing the round lock, such as allowing it to be invoked only after a certain period is passed since the last entropy request.

Remediation

This issue has been acknowledged by Hyperbeat Foundation.

4. Discussion

The purpose of this section is to document miscellaneous observations that we made during the assessment. These discussion notes are not necessarily security related and do not convey that we are suggesting a code change.

4.1. The `backerVaultTotal` variable may exceed `backerVaultCap`

The variable `backerVaultTotal` stores the total amount of deposited tokens. There is the variable `backerVaultCap`, which caps the total amount of deposited tokens after the deposit operation:

```
/// @notice Deposit capital as a backer or add to existing position
/// @dev Capital and exposure rate update immediately; allocation occurs at
    next round
/// @dev Supports fee-on-transfer tokens, refunds excess, floors to entry price
/// @param value Token amount to deposit
function backerDeposit(uint256 value) public {
    // (...)
    if (backerVaultTotal + flooredValue > backerVaultCap)
    revert DepositExceedsVaultCap();
    // (...)
    backerVaultTotal += flooredValue;
    // (...)
}
```

However, note that the actual value of `backerVaultTotal` may exceed the `backerVaultCap` because it also increases when 1) the fee paid by players are accumulated and 2) backers win the round.

We believe this behavior does not have to be altered considering it does not harm the business logic of the contract, but we recommend to document this behavior as it could be unintuitive.

4.2. Game is slightly biased towards backers

If the total amount a backer deposits is greater than the total amount that players entered for the round, the winner may be the backers or one of the players. Backers equally share the win/loss of the round.

When a player wins, the backer loses their backer vault but gets the player vault; thus in total, the backer loses `backerVaultAmount - playerVaultAmount`. When the backer wins, the backer keeps their backer vault while winning the player vault; thus in total, the backer wins `playerVaultAmount`. Roughly, the probability the player wins should be `playerVaultAmount / backerVaultAmount` and the probability the backer wins should be `(backerVaultAmount - playerVaultAmount) /`

`backerVaultAmount` in order to make the game fair for both backers and players.

The BeatPotV2 contract implements this by running `getWinningEntry(randomNumber, (backerVaultTotal * 10000 / entryPrice))` and checking whether this value exceeds `totalEntriesByRoundId[roundId]`. However, it should be noted that the game is still slightly unfair to players, mainly because players pay a fee when they enter the game while backers do not.

Assuming one player enters the round with 1 HYPE and one backer deposits 2 HYPE to the round, the expected value of the win for the backer is 0.95 * 59.5% - 1.1 * 40.5% = 0.12 HYPE in this round, which implies that the game is slightly biased towards the backer.

We shared our concern to Hyperbeat Foundation that rational users may not prefer to join the game as a player if their position is financially unfavorable. Hyperbeat Foundation replied that this behavior is intended and they believe the financial structure of the game is sustainable since this probabilistic bias offsets the loss aversion users may feel.

4.3. Entry price on the constructor is the only parameter without the token decimal

The contract fetches the decimal of the underlying token. It is then multiplied to the `_entryPrice` constructor parameter to calculate the value of `entryPrice`:

```
/// @notice Initializes the BeatPot contract (UUPS proxy pattern)
/// @param _entropyAddress Address of the Entropy V2 contract
/// @param _initialOwnerAddress Initial owner (should differ from deployer)
/// @param _token Address of ERC20 token for prizes/deposits
/// @param _entryPrice Price of single entry in token units (without decimals)
function initialize(
    address _entropyAddress,
    address _initialOwnerAddress,
    address _token,
    uint256 _entryPrice
) public initializer {
    //(...)
    token = IERC20(_token);
    tokenDecimals = IERC20Metadata(_token).decimals();

    entryPrice = _entryPrice * (10 ** tokenDecimals);
    // (...)
}
```

While it does not negatively affect how the contract behaves around the `entryPrice` variable, we recommend to modify this behavior so the `_entryPrice` parameter takes the token decimal into account (i.e., `entryPrice = _entryPrice`) because

- the way the contract handles the amount of tokens is not unified over the contract, and

- the initial entry price can only be set to a multiple of one token, which may not make sense for some high-valued tokens.

4.4. The `lastWinnerAddress` variable is unchanged when no players enter the round

The `lastWinnerAddress` variable stores the winner's address of the most recent round:

```
/// @notice Address of the most recent round winner (address(0) if backers won)
address public lastWinnerAddress;
```

Note that this value would be unchanged when no players enter the round.

We believe this behavior does not have to be changed considering it does not harm the business logic of the contract, but we recommend to document this behavior as it could be unintuitive.

5. Threat Model

This provides a full threat model description for various functions. As time permitted, we analyzed each function in the contracts and created a written threat model for some critical functions. A threat model documents a given function's externally controllable inputs and how an attacker could leverage each input to cause harm.

Not all functions in the audit scope may have been modeled. The absence of a threat model in this section does not necessarily suggest that a function is safe.

5.1. Module: BeatPotV2.sol

Function: `backerDeposit(uint256 value)`

This function allows a user to deposit tokens into the backer vault. The user receives shares representing their claim on the vault.

Inputs

- `value`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Must be greater than zero.
  - **Impact**: Amount of tokens to deposit.

Branches and code coverage

Intended branches

- Check `roundLock`.
  - ☑ Test coverage
- Check `value > 0`.
  - ☑ Test coverage
- Transfer tokens from the user.
  - ☑ Test coverage
- Calculate actual amount received (FoT support).
  - ☑ Test coverage
- Floor value to `entryPrice`.
  - ☑ Test coverage
- Check `flooredValue` constraints.
  - ☑ Test coverage
- Calculate shares to mint.

☑ Test coverage

- Mint shares.

☑ Test coverage

- Update `backerVaultTotal`.

☑ Test coverage

- Refund the remainder.

☑ Test coverage

Negative behavior

- Revert if the round is executing (`RoundCurrentlyExecuting`).

☑ Negative test

- Revert if the value is 0 (`InvalidDepositAmount`).

☑ Negative test

- Revert if the floored value is less than the entry price (`DepositBelowEntryPrice`).

☑ Negative test

- Revert if the cap is exceeded (`DepositExceedsVaultCap`).

☑ Negative test

Function call analysis

- `this.token.balanceOf(address(this))`
  - **What is controllable?** None directly. The value depends on the token contract's internal state.
  - **If the return value is controllable, how is it used and how can it go wrong?** It is used to calculate the actual amount received (supporting FoT tokens). If the token contract is malicious, it could report false balances.
  - **What happens if it reverts, reenters or does other unusual control flow?** If it reverts, the deposit transaction fails.
- `SafeERC20.safeTransferFrom(this.token, msg.sender, address(this), value)`
  - **What is controllable?** The `value` is controlled by the caller. The `msg.sender` is the source of funds.
  - **If the return value is controllable, how is it used and how can it go wrong?** N/A (standard ERC-20 transfer).
  - **What happens if it reverts, reenters or does other unusual control flow?** The transaction reverts if the transfer fails.
- `this.token.balanceOf(address(this))`
  - **What is controllable?** None directly. The value depends on the token

contract's internal state.
    - **If the return value is controllable, how is it used and how can it go wrong?** It is used to calculate the actual amount received. If the token contract is malicious, it could report false balances.
    - **What happens if it reverts, reenters or does other unusual control flow?** If it reverts, the deposit transaction fails.
- `this.totalSupply() -> this._getERC20Storage()`
    - **What is controllable?** None. The supply is managed by the contract logic.
    - **If the return value is controllable, how is it used and how can it go wrong?** It is used to calculate the share exchange rate.
    - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `this._mint(msg.sender, shares) -> this._update(address(0), account, value) -> this._getERC20Storage()`
    - **What is controllable?** The recipient (`msg.sender`) is fully controlled by the caller. The amount of `shares` is derived from the deposit.
    - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
    - **What happens if it reverts, reenters or does other unusual control flow?** If minting logic fails (e.g., hooks), the transaction reverts.
- `SafeERC20.safeTransfer(this.token, msg.sender, remainder)`
    - **What is controllable?** The recipient (`msg.sender`) is fully controlled by the caller as the refund target.
    - **If the return value is controllable, how is it used and how can it go wrong?** This action refunds excess tokens.
    - **What happens if it reverts, reenters or does other unusual control flow?** The transaction reverts if the refund fails.

Function: `burnShares(address backerAddress, uint256 sharesAmount)`

This is a restricted function that allows the WithdrawalQueue to burn shares from a backer's account upon withdrawal-request processing.

Inputs

- `backerAddress`
    - **Control**: The WithdrawalQueue contract.
    - **Constraints**: Must hold shares.
    - **Impact**: Reduces supply and user balance.
- `sharesAmount`

- **Control**: The WithdrawalQueue contract.
- **Constraints**: Must be less than or equal to the user balance.
- **Impact**: Amount of shares to burn.

Branches and code coverage

Intended branches

- Call `_burn`.
  - ☑ Test coverage

Negative behavior

- The caller must be `withdrawalQueue` (enforced by `onlyWithdrawalQueue` modifier).
  - ☑ Negative test

Function call analysis

- `this._burn(backerAddress, sharesAmount) -> this._update(account, address(0), value) -> this._getERC20Storage()`
  - **What is controllable?** `backerAddress` is passed by the trusted WithdrawalQueue contract.
  - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.

Function: `constructor()`

This function disables initializers to prevent the implementation contract from being initialized, as per UUPS-pattern best practices.

Branches and code coverage

Intended branches

- Disable initializers.
  - ☑ Test coverage

Negative behavior

- Cannot be initialized.

☑ Negative test

Function: `distributeBackerFeesToBackers()`

This function distributes collected backer fees to the backer vault. If a protocol-fee threshold is met, a portion is set aside for the protocol.

Branches and code coverage

Intended branches

- If `backerVaultTotal` is 0, the fees go to `playerVaultTotal`.
  - ☑ Test coverage
- Deduct protocol fee if `protocolFeeThreshold` is met.
  - ☑ Test coverage
- Add remaining fees to `backerVaultTotal`.
  - ☑ Test coverage

Function: `distributePlayerVaultToBackers()`

This function moves the entire player vault balance to the backer vault. It is used when backers win the round.

Branches and code coverage

Intended branches

- Add `playerVaultTotal` to `backerVaultTotal`.
  - ☑ Test coverage
- Reset `playerVaultTotal` to 0.
  - ☑ Test coverage

Function: `entropyCallback(uint64 sequenceNumber, address, byte[32] randomNumber)`

This is a callback function invoked by the Entropy contract to deliver the requested random number. This function verifies the request and executes the round logic to select a winner.

Inputs

- `sequenceNumber`
  - **Control**: Fully controlled by the caller (Entropy protocol).
  - **Constraints**: Must match `currentSequenceNumber`.
  - **Impact**: Identifies the specific randomness request.
- `<unnamed>`
  - **Control**: N/A.
  - **Constraints**: None.
  - **Impact**: None.
- `randomNumber`
  - **Control**: Fully controlled by the caller (Entropy protocol).
  - **Constraints**: Must be 32 bytes of randomness.
  - **Impact**: Determines the winner of the round.

Branches and code coverage

Intended branches

- Emit the `EntropyResult` event.
  - ☑ Test coverage
- Check `roundLock` (must be true).
  - ☑ Test coverage
- Check `sequenceNumber` matches.
  - ☑ Test coverage
- Call `selectWinnerAndRebalance`.
  - ☑ Test coverage
- Unlock the round, and reset the sequence number.
  - ☑ Test coverage

Negative behavior

- Revert if the round is not locked (`RoundExecutionNotInProgress`).
  - ☑ Negative test
- Revert if the sequence number mismatches (`SequenceNumberMismatch`).
  - ☑ Negative test
- The caller is not the entropy contract (handled by `IEntropyConsumer` inheritance).
  - ☑ Negative test

Function: `executeRound(bytes32 userRandomNumber)`

This function initiates the round-execution process by requesting randomness from the Entropy protocol. This function locks the round to prevent further actions until checks are complete.

Inputs

- `userRandomNumber`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None (any 32-byte value).
  - **Impact**: Mixed with provider entropy to ensure randomness.

Branches and code coverage

Intended branches

- Check if the round duration has elapsed.
  - ☑ Test coverage
- Check if the round is already executing (locked).
  - ☑ Test coverage
- Lock the round.
  - ☑ Test coverage
- Calculate the entropy fee, and check `msg.value`.
  - ☑ Test coverage
- Refund excess ETH.
  - ☑ Test coverage
- Request entropy validation.
  - ☑ Test coverage
- Store the sequence number.
  - ☑ Test coverage
- Emit `RoundExecutionRequested`.
  - ☑ Test coverage

Negative behavior

- Revert if `block.timestamp` < `lastRoundEndTime` + duration (`RoundDurationNotElapsed`).
  - ☑ Negative test

- Revert if `roundLock` is true (`RoundCurrentlyExecuting`).
  - ☑ Negative test
- Revert if `msg.value` is less than the fee (`InsufficientEntropyFee`).
  - ☑ Negative test
- Revert if the refund fails (`RefundFailed`).
  - ☑ Negative test

Function call analysis

- `this.getEntropyFee() -> this.entropy.getFeeV2(this.entropyProvider, this.callbackGasLimit)`
  - **What is controllable?** `callbackGasLimit` is controlled by the admin. The fee returned is controlled by the Entropy contract.
  - **If the return value is controllable, how is it used and how can it go wrong?** The returned fee determines the `msg.value` requirement. If the Entropy contract returns an excessively high fee, it could prevent execution.
  - **What happens if it reverts, reenters or does other unusual control flow?** If the fee-calculation reverts, the round execution cannot proceed.
- `this.getEntropyFee() -> this.entropy.getFeeV2()`
  - **What is controllable?** The fee returned is controlled by the Entropy contract.
  - **If the return value is controllable, how is it used and how can it go wrong?** Similar to the above — an invalid fee could block execution.
  - **What happens if it reverts, reenters or does other unusual control flow?** Execution reverts.
- `this.entropy.requestV2{value: fee}(this.entropyProvider, userRandomNumber, this.callbackGasLimit)`
  - **What is controllable?** `userRandomNumber` is fully controlled by the caller. `callbackGasLimit` is controlled by the admin.
  - **If the return value is controllable, how is it used and how can it go wrong?** The call returns a `sequenceNumber`. If the Entropy contract is malicious, it could duplicate sequence numbers or manipulate the process, but we assume the Entropy protocol is secure.
  - **What happens if it reverts, reenters or does other unusual control flow?** Using `requestV2` protects against reentrancy. Reverts will cancel the round execution.

Function: `findWinnerFromPlayers(uint256 winningEntry)`

This function iterates through the list of players for the round to find which player owns the winning entry number.

Inputs

- `winningEntry`
  - **Control**: Internal (from `getWinningEntry`).
  - **Constraints**: Must be between `1` and `totalEntries`.
  - **Impact**: Determines the winning address.

Branches and code coverage

Intended branches

- Iterate players.
  - ☑ Test coverage
- Accumulate entries.
  - ☑ Test coverage
- Return player if cumulative entries are greater than or equal to the winning entry.
  - ☑ Test coverage
- Return fallback winner if loop completes (safety fallback).
  - ☑ Test coverage

Function: `forceReleaseRoundLock()`

This is an emergency function to unlock the round if entropy callback fails or gets stuck.

Branches and code coverage

Intended branches

- Set `roundLock` to false.
  - ☑ Test coverage
- Reset `currentSequenceNumber`.
  - ☑ Test coverage

Negative behavior

- The caller must be the owner.
  - ☑ Negative test

Function: `getEntropyFee()`

This function calculates the fee required by the Entropy protocol to request randomness.

Branches and code coverage

Intended branches

- If `callbackGasLimit > 0`, use it to calculate the fee.
  - ☑ Test coverage
- Otherwise, use the default fee calculation.
  - ☑ Test coverage

Function: `getEntropy()`

This function returns the address of the Entropy contract used for randomness.

Branches and code coverage

Intended branches

- Return the entropy contract address.
  - ☑ Test coverage

Function: `getWinningEntry(byte[32] rawRandomNumber, uint256 max)`

This function calculates the winning entry number from the random seed using modulo arithmetic.

Inputs

- `rawRandomNumber`
  - **Control**: Fully controlled by the caller (Entropy protocol).
  - **Constraints**: Must be 32 bytes of randomness.
  - **Impact**: Determines the outcome.
- `max`
  - **Control**: Internal state (total entries).
  - **Constraints**: Must be greater than zero.
  - **Impact**: Upper bound for the winning number.

Branches and code coverage

Intended branches

- Return (random % max) + 1.
  - ☑ Test coverage

Function: `initialize(address _entropyAddress, address _initialOwnerAddress, address _token, uint256 _entryPrice)`

This function initializes the contract state, setting up the owner, token, entropy provider, and default configuration parameters.

Inputs

- `_entropyAddress`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Must be a valid Entropy V2 contract address.
  - **Impact**: Source of randomness for the lottery.
- `_initialOwnerAddress`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Should be a valid address (not zero address).
  - **Impact**: Full control over administrative functions.
- `_token`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Must be a compliant ERC-20 token.
  - **Impact**: The asset used for betting and prizes.
- `_entryPrice`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Must be greater than zero.
  - **Impact**: Determines the cost of a single lottery entry.

Branches and code coverage

Intended branches

- Initialize Ownable, UUPS, and ERC-20.
  - ☑ Test coverage
- Set entropy provider and defaults.

- ☑ Test coverage
- Set token and decimals.
  - ☑ Test coverage
- Set round configuration (duration, price, limits).
  - ☑ Test coverage

Negative behavior

- Cannot be called more than once (ensured by `initializer`).
  - ☑ Negative test

Function call analysis

- `this.__Ownable_init(_initialOwnerAddress) -> this.__Ownable_init_unchained(initialOwner) -> this._transferOwnership(initialOwner) -> this._getOwnableStorage()`
  - **What is controllable?** Initial owner address.
  - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters or does other unusual control flow?** Initialization fails.
- `this.__Ownable_init(_initialOwnerAddress) -> this.__Ownable_init_unchained(initialOwner) -> this._transferOwnership(initialOwner) -> this._getOwnable2StepStorage()`
  - **What is controllable?** N/A.
  - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `this.__ERC20_init("BeatPotV2", "BPV2") -> this.__ERC20_init_unchained(name_, symbol_) -> this._getERC20Storage()`
  - **What is controllable?** The name and symbol are hardcoded.
  - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `this.entropy.getDefaultProvider()`
  - **What is controllable?** The return value (default provider) is determined by the Entropy contract logic.
  - **If the return value is controllable, how is it used and how can it go wrong?** It

sets the `entropyProvider`. If the Entropy contract is malicious or compromised, it could return a provider that manipulates randomness.
    - **What happens if it reverts, reenters or does other unusual control flow?** If this call reverts, contract initialization fails.
- `IERC20Metadata(_token).decimals()`
    - **What is controllable?** The decimals value is controlled by the implementation of the provided `_token` contract.
    - **If the return value is controllable, how is it used and how can it go wrong?** It scales the `entryPrice`. If a malicious token returns an extremely large value, it could cause mathematical overflows or set an unintended price.
    - **What happens if it reverts, reenters or does other unusual control flow?** If this call reverts, contract initialization fails.

Function: `purchaseEntries(address referrer, uint256 value, address recipient)`

This function allows a user to purchase lottery entries. It supports gifting entries to a recipient.

Inputs

- `referrer`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: Cannot be `msg.sender`.
    - **Impact**: Receives the referral fee.
- `value`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: Must suffice for at least one entry.
    - **Impact**: Number of entries purchased.
- `recipient`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: None.
    - **Impact**: Who gets the entries.

Branches and code coverage

Intended branches

- Check `allowPurchasing` and `roundLock`.
    - ☑ Test coverage

- Check `referrer != msg.sender`.
  - ☑ Test coverage
- Transfer tokens.
  - ☑ Test coverage
- Resolve `playerAddress` (recipient or sender).
  - ☑ Test coverage
- Call `_processEntryPurchase`.
  - ☑ Test coverage
- Call `_calculateFees` and `_updateFeeTotals`.
  - ☑ Test coverage
- Update `playerVaultTotal`.
  - ☑ Test coverage
- Refund the remainder.
  - ☑ Test coverage
- Emit `PlayerEntryPurchase`.
  - ☑ Test coverage

Negative behavior

- Revert if purchasing is disabled (`PurchasingNotAllowed`).
  - ☑ Negative test
- Revert if value is 0 (`InvalidPurchaseAmount`).
  - ☑ Negative test
- Revert if the round is locked (`RoundCurrentlyExecuting`).
  - ☑ Negative test
- Revert if it is self-referral (`CannotSelfReferral`).
  - ☑ Negative test

Function call analysis

- `this.token.balanceOf(address(this))`
  - **What is controllable?** None directly.
  - **If the return value is controllable, how is it used and how can it go wrong?** It is used to calculate the actual received amount.
  - **What happens if it reverts, reenters or does other unusual control flow?** Revert.

- `SafeERC20.safeTransferFrom(this.token, msg.sender, address(this), value)`
    - **What is controllable?** The `value` and the source of funds (`msg.sender`) are fully controlled by the caller.
    - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
    - **What happens if it reverts, reenters or does other unusual control flow?** Revert.
- `this.token.balanceOf(address(this))`
    - **What is controllable?** None directly. The value depends on the token contract's internal state.
    - **If the return value is controllable, how is it used and how can it go wrong?** It is used to calculate the actual amount received. If the token contract is malicious, it could report false balances.
    - **What happens if it reverts, reenters or does other unusual control flow?** If it reverts, the purchase transaction fails.
- `SafeERC20.safeTransfer(this.token, msg.sender, remainder)`
    - **What is controllable?** The refund recipient (`msg.sender`) is fully controlled by the caller.
    - **If the return value is controllable, how is it used and how can it go wrong?** Determines where excess tokens are sent.
    - **What happens if it reverts, reenters or does other unusual control flow?** Revert.

Function: `selectWinnerAndRebalance(byte[32] randomNumber)`

This function determines the winner of the round and distributes the prize. It handles two main cases:

1. When the player wins, the player gets the pot (or backer vault if partially funded).
2. When the backers win (if partially funded), the backers keep the pot.

This function also handles fee distribution and processes withdrawals.

Inputs

- `randomNumber`
    - **Control**: Fully controlled by the caller (Entropy protocol).
    - **Constraints**: Must be 32 bytes.
    - **Impact**: Randomness source.

Branches and code coverage

Intended branches

- Refund when there are no players.
  - ☑ Test coverage
- Distribute backer fees.
  - ☑ Test coverage
- When fully funded by players, the winner gets `playerVaultTotal`.
  - ☑ Test coverage
- When partially funded (backers participated) and the player wins, the winner gets `backerVaultTotal`.
  - ☑ Test coverage
- When partially funded (backers participated) and the player wins, the backers get `playerVaultTotal` (via `distributePlayerVaultToBackers`).
  - ☑ Test coverage
- When partially funded (backers participated) and the backers win, the backers get `playerVaultTotal`.
  - ☑ Test coverage
- Reset round state (e.g., `allFeesTotal`).
  - ☑ Test coverage
- Process withdrawals (`_processWithdrawalsForRound`).
  - ☑ Test coverage
- Increment `roundId`.
  - ☑ Test coverage

Function: `setAllowPurchasing(bool _allow)`

This function toggles whether entry purchasing is allowed.

Inputs

- `_allow`
  - **Control**: Fully controlled by the caller (Admin).
  - **Constraints**: None.
  - **Impact**: Halts or enables gameplay.

Branches and code coverage

Intended branches

- Update `allowPurchasing`.
  - ☑ Test coverage

Negative behavior

- The caller must be the owner.
  - ☑ Negative test

Function: `setBackerVaultCap(uint256 _cap)`

This function sets the maximum capacity for the backer vault.

Inputs

- `_cap`
  - **Control**: Fully controlled by the caller (admin).
  - **Constraints**: None.
  - **Impact**: Limits backer exposure and deposits.

Branches and code coverage

Intended branches

- Update `backerVaultCap`.
  - ☑ Test coverage

Negative behavior

- The caller must be owner.
  - ☑ Negative test

Function: `setCallbackGasLimit(uint32 _gasLimit)`

This function sets the gas limit for the entropy callback.

Inputs

- `_gasLimit`
  - **Control**: Fully controlled by the caller (admin).
  - **Constraints**: None.
  - **Impact**: Entropy-callback success probability.

Branches and code coverage

Intended branches

- Update `callbackGasLimit`.
  - ☑ Test coverage

Negative behavior

- The caller must be the owner.
  - ☑ Negative test

Function: `setEntryPrice(uint256 _newEntryPrice)`

This function updates the price for a single lottery entry.

Inputs

- `_newEntryPrice`
  - **Control**: Fully controlled by the caller (admin).
  - **Constraints**: None strictly enforced.
  - **Impact**: Cost of participation.

Branches and code coverage

Intended branches

- Update `entryPrice`.
  - ☑ Test coverage

Negative behavior

- The caller must be the owner.
  - ☑ Negative test

Function: `setFallbackWinner(address _fallbackWinner)`

This function sets the address to receive funds if no winner can be found (should not happen in normal operations).

Inputs

- `_fallbackWinner`
  - **Control**: Fully controlled by the caller (admin).
  - **Constraints**: None.
  - **Impact**: Emergency-fund recipient.

Branches and code coverage

Intended branches

- Update `fallbackWinner`.
  - ☑ Test coverage

Negative behavior

- The caller must be the owner.
  - ☑ Negative test

Function: `setFeeBps(uint256 _feeBps)`

This function updates the total fee percentage.

Inputs

- `_feeBps`
  - **Control**: Fully controlled by the caller (admin).
  - **Constraints**: Max 80% — the margin must be greater than the referral fee.
  - **Impact**: Revenue and backer incentives.

Branches and code coverage

Intended branches

- Check the max constraint.

☑ Test coverage

- Check the margin constraint.

☑ Test coverage

- Update `feeBps`.

☑ Test coverage

Negative behavior

- Revert if greater than 80% (`FeeBpsExceedsMaximum`).

☑ Negative test

- Revert if there is insufficient margin (`InsufficientFeeBpsMargin`).

☑ Negative test

- The caller must be the owner.

☑ Negative test

Function: `setMinBackerDeposit(uint256 _minDeposit)`

This function sets the minimum deposit amount for backers.

Inputs

- `_minDeposit`
  - **Control**: Fully controlled by the caller (admin).
  - **Constraints**: None.
  - **Impact**: Deposit barriers.

Branches and code coverage

Intended branches

- Update `minBackerDeposit`.

☑ Test coverage

Negative behavior

- The caller must be the owner.

☑ Negative test

Function: `setPlayerLimit(uint256 _playerLimit)`

This function sets the maximum number of players per round.

Inputs

- `_playerLimit`
  - **Control**: Fully controlled by the caller (admin).
  - **Constraints**: None.
  - **Impact**: Scalability limits.

Branches and code coverage

Intended branches

- Update `playerLimit`.
  - ☑ Test coverage

Negative behavior

- The caller must be the owner.
  - ☑ Negative test

Function: `setProtocolFeeAddress(address _protocolFeeAddress)`

This function sets the address where protocol fees are sent.

Inputs

- `_protocolFeeAddress`
  - **Control**: Fully controlled by the caller (admin).
  - **Constraints**: None.
  - **Impact**: Protocol revenue destination.

Branches and code coverage

Intended branches

- Update `protocolFeeAddress`.
  - ☑ Test coverage

Negative behavior

- The caller must be the owner.
  - ☑ Negative test

Function: `setProtocolFeeThreshold(uint256 _protocolFeeThreshold)`

This function sets the threshold of backer fees required to trigger protocol-fee deduction.

Inputs

- `_protocolFeeThreshold`
  - **Control**: Fully controlled by the caller (admin).
  - **Constraints**: None.
  - **Impact**: When the protocol takes a cut.

Branches and code coverage

Intended branches

- Update `protocolFeeThreshold`.
  - ☑ Test coverage

Negative behavior

- The caller must be the owner.
  - ☑ Negative test

Function: `setReferralFeeBps(uint256 _referralFeeBps)`

This function updates the referral-fee percentage.

Inputs

- `_referralFeeBps`
  - **Control**: Fully controlled by the caller (admin).
  - **Constraints**: Must be less than or equal to `feeBps`.
  - **Impact**: Referral incentives.

Branches and code coverage

Intended branches

- Check the constraint.
  - ☑ Test coverage
- Update `referralFeeBps`.
  - ☑ Test coverage

Negative behavior

- Revert if `_referralFeeBps > feeBps` (`ReferralFeeBpsExceedsFee`).
  - ☑ Negative test
- The caller must be the owner.
  - ☑ Negative test

Function: `setRoundDurationInSeconds(uint256 _newDuration)`

This function updates the duration of each lottery round.

Inputs

- `_newDuration`
  - **Control**: Fully controlled by the caller (admin).
  - **Constraints**: None.
  - **Impact**: Round frequency.

Branches and code coverage

Intended branches

- Update `roundDurationInSeconds`.
  - ☑ Test coverage

Negative behavior

- The caller must be the owner.
  - ☑ Negative test

Function: `setWithdrawalQueue(address _withdrawalQueue)`

This function sets the address of the withdrawal-queue contract.

Inputs

- `_withdrawalQueue`
  - **Control**: Fully controlled by the caller (admin).
  - **Constraints**: None (assumed valid).
  - **Impact**: Withdrawal processing.

Branches and code coverage

Intended branches

- Update `withdrawalQueue`.
  - ☑ Test coverage

Negative behavior

- The caller must be the owner.
  - ☑ Negative test

Function: `userLiquidityBalance(address user)`

This function calculates the underlying token value of a user's shares in the backer vault.

Inputs

- `user`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None.
  - **Impact**: Return value target.

Branches and code coverage

Intended branches

- Calculate share value based on `backerVaultTotal` and `totalSupply`.
  - ☑ Test coverage

- Return 0 if the supply is zero.

  ☑ Test coverage

Function: `withdrawPrizes()`

This function allows a player to withdraw their accumulated prize winnings.

Branches and code coverage

Intended branches

- Check `prizesClaimable > 0`.

  ☑ Test coverage
- Emit the event.

  ☑ Test coverage
- Reset `prizesClaimable` to 0.

  ☑ Test coverage
- Transfer tokens.

  ☑ Test coverage

Negative behavior

- Revert if there are no prizes (`NoPrizesToWithdraw`).

  ☑ Negative test

Function call analysis

- `SafeERC20.safeTransfer(this.token, msg.sender, transferAmount)`
  - **What is controllable?** The recipient (`msg.sender`) is fully controlled by the caller.
  - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters or does other unusual control flow?** If the transfer reverts, the withdrawal fails and no state change occurs.

Function: `withdrawProtocolFees()`

This function allows the admin to withdraw accumulated protocol fees to the configured protocol-fee address.

Branches and code coverage

Intended branches

- Check `protocolFeeClaimable > 0`.
  - ☑ Test coverage
- Check `protocolFeeAddress` is set.
  - ☑ Test coverage
- Reset `protocolFeeClaimable`.
  - ☑ Test coverage
- Transfer fees.
  - ☑ Test coverage
- Emit the event.
  - ☑ Test coverage

Negative behavior

- Revert if there are no fees (`NoProtocolFeesToWithdraw`).
  - ☑ Negative test
- Revert if the address is not set (`ProtocolFeeAddressNotSet`).
  - ☑ Negative test
- The caller must be the owner.
  - ☑ Negative test

Function call analysis

- `SafeERC20.safeTransfer(this.token, this.protocolFeeAddress, transferProtocolFeeAmount)`
  - **What is controllable?** The recipient (`protocolFeeAddress`) is controlled by the admin.
  - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters or does other unusual control flow?** If the transfer reverts, the withdrawal fails.

Function: `withdrawReferralFees()`

This function allows a referrer to withdraw their accumulated referral fees.

Branches and code coverage

Intended branches

- Check `referralFeesClaimable > 0`.
  - ☑ Test coverage
- Emit the event.
  - ☑ Test coverage
- Reset `referralFeesClaimable` to ‘0.
  - ☑ Test coverage
- Transfer tokens.
  - ☑ Test coverage

Negative behavior

- Revert if there are no fees (`NoReferralFeesToWithdraw`).
  - ☑ Negative test

Function call analysis

- `SafeERC20.safeTransfer(this.token, msg.sender, transferAmount)`
  - **What is controllable?** The recipient (`msg.sender`) is fully controlled by the caller.
  - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters or does other unusual control flow?** If the transfer reverts, the withdrawal fails.

Function: `_calculateFees(uint256 usedAmount, address referrer)`

This function calculates the breakdown of fees (total, referral, and backer) for a given amount.

Inputs

- `usedAmount`
  - **Control**: Fully controlled by the caller (indirectly via purchase).
  - **Constraints**: None.
  - **Impact**: Base for fee calculation.
- `referrer`

- **Control**: Fully controlled by the caller.
- **Constraints**: None.
- **Impact**: Determines if referral fee is nonzero.

Branches and code coverage

Intended branches

- Calculate the total fee.
  - ☑ Test coverage
- Calculate the referral fee (if referrer exists).
  - ☑ Test coverage
- Calculate the backer fee (remainder).
  - ☑ Test coverage

Function: `_processEntryPurchase(uint256 actualReceived, address playerAddress)`

This function calculates entries purchased based on amount received, updates player stats, and handles player limit logic.

Inputs

- `actualReceived`
  - **Control**: Fully controlled by the caller (via transfer).
  - **Constraints**: Must be greater than or equal to `entryPrice`.
  - **Impact**: Determines entry count.
- `playerAddress`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None.
  - **Impact**: Owner of entries.

Branches and code coverage

Intended branches

- Calculate `entryCount`.
  - ☑ Test coverage

- Check internal `entryCount > 0`.
  - ☑ Test coverage
- Check `playerLimit` if there is a new player and add to the list.
  - ☑ Test coverage
- Calculate `entriesPurchasedBps` (fee deducted).
  - ☑ Test coverage
- Update player entries and total entries.
  - ☑ Test coverage

Negative behavior

- Revert if `entryCount` is 0 (`InsufficientAmountForEntry`).
  - ☑ Negative test
- Revert if player limit has been reached for the new player (`MaxPlayerLimitReached`).
  - ☑ Negative test

Function: `_processWithdrawalsForRound()`

This function processes queued withdrawals for the current round by transferring the proportional share of the backer vault to the withdrawal-queue contract.

Branches and code coverage

Intended branches

- Check `withdrawalQueue.withdrawalShares > 0`.
  - ☑ Test coverage
- Calculate the withdrawal token amount.
  - ☑ Test coverage
- Update withdrawal-queue total amount.
  - ☑ Test coverage
- Transfer tokens to the withdrawal queue.
  - ☑ Test coverage
- Deduct from `backerVaultTotal`.
  - ☑ Test coverage

Function: `_updateFeeTotals(uint256 allFeeAmount, uint256 referralFeeAmount, uint256 backerFeeAmount, address referrer)`

This function updates the global fee accumulators and the specific referrer's claimable balance.

Inputs

- `allFeeAmount`
  - **Control**: Internal calculation.
  - **Constraints**: None.
  - **Impact**: Increases `allFeesTotal`.
- `referralFeeAmount`
  - **Control**: Internal calculation.
  - **Constraints**: None.
  - **Impact**: Increases referrer's balance.
- `backerFeeAmount`
  - **Control**: Internal calculation.
  - **Constraints**: None.
  - **Impact**: Increases `backerFeesTotal`.
- `referrer`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None.
  - **Impact**: Recipient of the referral fee.

Branches and code coverage

Intended branches

- Update `allFeesTotal`.
  - ☑ Test coverage
- Update `referralFeesClaimable` (if referrer exists).
  - ☑ Test coverage
- Update `backerFeesTotal`.
  - ☑ Test coverage

5.2. Module: BeatPotWithdrawalQueue.sol

Function: `claimWithdrawal(uint256 _roundId)`

This function allows a user to claim their tokens after a round has completed and withdrawals are processed.

Inputs

- `_roundId`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Must be less than the current round ID.
  - **Impact**: Which round to claim from.

Branches and code coverage

Intended branches

- Check that the round has completed.
  - ☑ Test coverage
- Check that the withdrawal has not already been claimed.
  - ☑ Test coverage
- Mark as withdrawn.
  - ☑ Test coverage
- Calculate token amount (proportional).
  - ☑ Test coverage
- Transfer tokens.
  - ☑ Test coverage
- Emit the event.
  - ☑ Test coverage

Negative behavior

- Revert if the round has not completed (`RoundNotCompleted`).
  - ☑ Negative test
- Revert if the withdrawal has already been claimed (`WithdrawalAlreadyClaimed`).
  - ☑ Negative test

Function call analysis

- `SafeERC20.safeTransfer(this.token, msg.sender, tokenAmount)`
  - **What is controllable?** The recipient (`msg.sender`) is fully controlled by the caller.
  - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters or does other unusual control flow?** If transfer fails, the claim reverts.

Function: `constructor(address _beatpot)`

This function initializes the withdrawal queue with the BeatPot contract address.

Inputs

- `_beatpot`
  - **Control**: Fully controlled by the caller (deployer).
  - **Constraints**: Must be a valid BeatPotV2 address.
  - **Impact**: Determines the authorized caller and token.

Branches and code coverage

Intended branches

- Set `beatpot`.
  - ☑ Test coverage
- Set `token`.
  - ☑ Test coverage

Function: `requestWithdrawal(uint256 _sharesAmount)`

This function allows a backer to request withdrawal of their liquidity shares. Shares are transferred to this contract until processed.

Inputs

- `_sharesAmount`
  - **Control**: Fully controlled by the caller.

- **Constraints**: Must be greater than zero.
- **Impact**: Amount of shares to withdraw.

Branches and code coverage

Intended branches

- Check `amount > 0`.
  - ☑ Test coverage
- Update `withdrawalRequest` for user.
  - ☑ Test coverage
- Update `withdrawalShares` for the round.
  - ☑ Test coverage
- Transfer shares from the user to this contract.
  - ☑ Test coverage
- Emit the event.
  - ☑ Test coverage

Negative behavior

- Revert if the amount is zero (`InvalidSharesAmount`).
  - ☑ Negative test

Function call analysis

- `SafeERC20.safeTransferFrom(IERC20(address(this.beatpot)), msg.sender, address(this), _sharesAmount)`
  - **What is controllable?** The source of funds (`msg.sender`) and amount (`_sharesAmount`) are fully controlled by the caller.
  - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters or does other unusual control flow?** If the transfer fails, the request reverts.

Function: `updateTotalWithdrawalAmount(uint256 _roundId, uint256 _amount)`

This function updates the total withdrawal amount for a round and burns the corresponding shares held by this contract. It is called by BeatPotV2.

Inputs

- `_roundId`
  - **Control**: Fully controlled by the caller (BeatPotV2).
  - **Constraints**: None.
  - **Impact**: Round identification.
- `_amount`
  - **Control**: Fully controlled by the caller (BeatPotV2).
  - **Constraints**: None.
  - **Impact**: Total tokens allocated for withdrawal.

Branches and code coverage

Intended branches

- Burn shares held by this contract.
  - ☑ Test coverage
- Update `totalWithdrawalAmount`.
  - ☑ Test coverage
- Emit the event.
  - ☑ Test coverage

Negative behavior

- The caller must be the BeatPot contract (`onlyBeatPot`).
  - ☑ Negative test

Function call analysis

- `this.beatpot.burnShares(address(this), this.withdrawalShares[_roundId])`
  - **What is controllable?** `_roundId` is controlled by `BeatPotV2` (the caller).
  - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters or does other unusual control flow?** If burning fails, the update reverts.

6. Assessment Results

During our assessment on the scoped BeatPotV2 contracts, we discovered 11 findings. No critical issues were found. Four findings were of high impact, two were of medium impact, three were of low impact, and the remaining findings were informational in nature.

We often observe a high number of findings in projects undergoing rapid development. Our recommendation to the Hyperbeat Foundation team is to implement a security-focused development workflow. This includes augmenting the codebase with a comprehensive test suite to ensure proper behavior under real-world conditions. For example, we believe that the Finding 3.5. ↗ and 3.9. ↗ should have been caught through the testing process during the development.

It is our opinion that the Finding 3.1. ↗, 3.3. ↗ and 3.4. ↗ could be caught if the test suite involved executing multiple business logics under the environment close to production. We recommend to deploy the contract to HyperEVM testnet for several weeks to validate the intended behavior under real conditions. Also, consider expanding test coverage by implementing multioperation test scenarios, rather than relying on single-action tests.

6.1. Disclaimer

This assessment does not provide any warranties about finding all possible issues within its scope; in other words, the evaluation results do not guarantee the absence of any subsequent issues. Zellic, of course, also cannot make guarantees about any code added to the project after the version reviewed during our assessment. Furthermore, because a single assessment can never be considered comprehensive, we always recommend multiple independent assessments paired with a bug bounty program.

For each finding, Zellic provides a recommended solution. All code samples in these recommendations are intended to convey how an issue may be resolved (i.e., the idea), but they may not be tested or functional code. These recommendations are not exhaustive, and we encourage our partners to consider them as a starting point for further discussion. We are happy to provide additional guidance and advice as needed.

Finally, the contents of this assessment report are for informational purposes only; do not construe any information in this report as legal, tax, investment, or financial advice. Nothing contained in this report constitutes a solicitation or endorsement of a project by Zellic.