Zellic

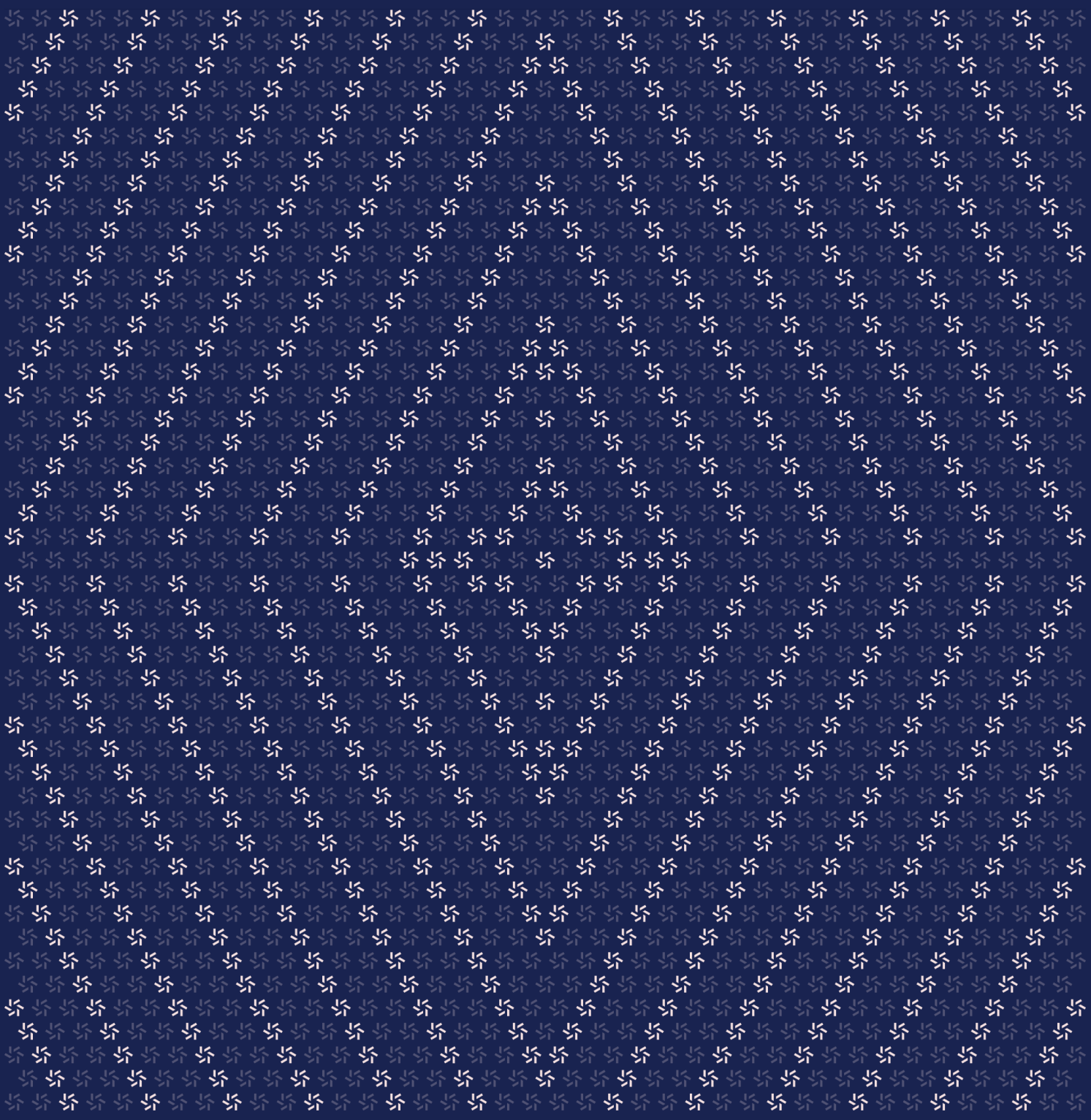
Prepared for
King Koala
Sanguine Labs LTD

Prepared by
Katerina Belotskaia
Jinseo Kim
Zellic

September 6, 2024

Yeet

Smart Contract Security Assessment



Contents

About Zellic **4**

1. Overview **4**

1.1. Executive Summary 5

1.2. Goals of the Assessment 5

1.3. Non-goals and Limitations 5

1.4. Results 5

2. Introduction **6**

2.1. About Yeet 7

2.2. Methodology 7

2.3. Scope 9

2.4. Project Overview 9

2.5. Project Timeline 10

3. Detailed Findings **10**

3.1. NFT boosting can be applied multiple times with one NFT 11

3.2. Incorrect calculation of the minimum amount for Yeet 14

3.3. Unoptimized `getClaimableAmount` function 16

3.4. Bias of the `draftWinners` function in the Yeetback contract 18

3.5. Precision loss in `getDistribution` resulting in revert 20

3.6. Incorrect comment 22

4. Threat Model **23**

4.1. Module: Reward.sol 24

4.2. Module: Yeet.sol 26

5. Assessment Results **29**

5.1. Disclaimer 30

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 Sanguine Labs LTD from August 26th to August 28th, 2024. During this engagement, Zellic reviewed Yeet'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:

- Is there some bias in the drawing function in the Yeetback contract?
- Is the reward function correctly implemented?
- Is the pot value properly calculated?

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

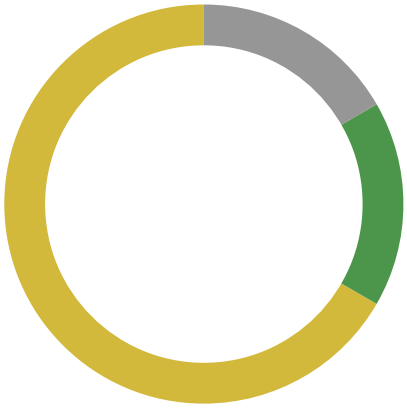
Due to the time-boxed nature of security assessments in general, there are limitations in the coverage an assessment can provide.

1.4. Results

During our assessment on the scoped Yeet contracts, we discovered six findings. No critical issues were found. Four findings were of medium impact, one was of low impact, and the remaining finding was informational in nature.

Breakdown of Finding Impacts

Impact Level	Count
Critical	0
High	0
Medium	4
Low	1
Informational	1



2. Introduction

2.1. About Yeet

Sanguine Labs LTD contributed the following description of Yeet:

> Yeet is a gamified DeFi protocol in the Berachain ecosystem with no dominant game theoretic strategy. Players can win or lose money in a variety of different ways, and employ multiple types of tactics whilst playing.

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.

2.3. Scope

The engagement involved a review of the following targets:

Yeet Contracts

Type	Solidity
Platform	EVM-compatible
Target	contracts
Repository	https://github.com/0xKingKoala/contracts ↗
Version	`526e3e503079913de4189b99dfea41f2a973d419`

Programs	`Reward.sol`
	`Yeet.sol`
	`YeetBack.sol`

2.4. Project Overview

Zellic was contracted to perform a security assessment for a total of four and a half 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:

Jacob Goreski
Engagement Manager
jacob@zellic.io ↗

Chad McDonald
Engagement Manager
chad@zellic.io ↗

The following consultants were engaged to conduct the assessment:

Katerina Belotskaia
Engineer
kate@zellic.io ↗

Jinseo Kim
Engineer
jinseo@zellic.io ↗

2.5. Project Timeline

The key dates of the engagement are detailed below.

August 26, 2024	Kick-off call
August 26, 2024	Start of primary review period
August 28, 2024	End of primary review period

3. Detailed Findings

3.1. NFT boosting can be applied multiple times with one NFT

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

Description

When a user invokes the `yeet` function, they can optionally provide the `tokenIds` parameter, which indicates the list of the NFTs of the user. If a user has NFTs, they can receive the boost for their reward. The multiplier of the boost increases, depending on the number of NFTs. The following code implements this feature:

```
/// @notice this function is used to yeet with tokenIds (NFTs)
function yeet(uint256[] memory tokenIds) external payable {
    _yeet(tokenIds);
}

/// @notice yeet is the main function of the game, users yeet the native token
    in the pot
/// @param tokenIds the tokenIds of the NFTs the user owns, used to calculate
    the boost
function _yeet(uint256[] memory tokenIds) internal {
    // (...)

    uint256 boostedValue = getBoostedValue(msg.sender, valueToPot, tokenIds);
    rewardsContract.addYeetVolume(msg.sender, boostedValue);
    stakingContract.depositReward{value: valueToStakers}();
}

// (...)

/// @notice getBoostedValue is a function that returns the boosted value of a
    yeet based on how many NFTs the user owns
/// @param sender the user that yeeted
/// @param value the value of the yeet
/// @param tokenIds the tokenIds of the NFTs the user owns
/// @return uint256 the boosted value
function getBoostedValue(address sender, uint256 value, uint256[]
    memory tokenIds) public view returns (uint256) {
    uint256 nftBoost = getNFTBoost(sender, tokenIds);
    return value + (value * nftBoost) / 10000;
```

```
}

// (...)

/// @notice getNFTBoost is a function that returns the NFT boost of a user
    based on how many NFTs the user owns
function getNFTBoost(address owner, uint256[] memory tokenIds)
    public view returns (uint256) {
    if (yeetardsNFTsAddress == address(0)) {
        return 0;
    }
    INFTContract nftContract = INFTContract(yeetardsNFTsAddress);

    uint256 balance = tokenIds.length;
    for (uint i = 0; i < balance; i++) {
        // make sure the user is the owner of all the tokensIds
        uint256 tokenId = tokenIds[i];
        if (nftContract.ownerOf(tokenId) != owner) {
            revert UserDoesNotOwnNFTs(owner, tokenId);
        }
        if (!nftContract.isEligibleForBoost(tokenId)) {
            revert NFTNotEligibleForBoost(tokenId);
        }
    }

    if (balance > nftBoostLookup.length - 1) {
        return nftBoostLookup[nftBoostLookup.length - 1];
    }

    return nftBoostLookup[balance];
}
```

However, it is not checked that an NFT appears only once in the given parameter `tokenIds`.

Impact

A malicious user can provide the `tokenIds` parameter, which includes one NFT the user owns, multiple times, in order to maximize their boost without having several NFTs.

Recommendations

Consider checking if all elements of the given `tokenIds` parameter are unique.

Remediation

This issue has been acknowledged by Sanguine Labs LTD, and a fix was implemented in commit `7e5e936d`.

3.2. Incorrect calculation of the minimum amount for Yeet

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

Description

The `_minimumYeetPoint` function claculates the minimum amount of BERA necessary for a Yeet to be successful. The function is called in the `yeet` function to ensure that the amount of BERA sent is greater than or equal to the minimum amount. Each Yeet adds an entry for the Yeetback raffle. The use of `MINIMUM_YEET_POINT` ensures that a substantial amount of BERA is sent for an entry to be added.

```
function _minimumYeetPoint(uint256 totalPot) private view returns (uint256) {
    if (totalPot == 0) {
        return MINIMUM_YEET_POINT;
    }

    if(isBoostrapPhase()) {
       return MINIMUM_YEET_POINT;
    }

    return totalPot / POT_DIVISION;
}
```

However, despite the name of the variable `MINIMUM_YEET_POINT`, the minimum amount for a Yeet can be less than this variable.

If `BOOSTRAP_PHASE_DURATION` is set to zero, which is the default configuration, the initial amount of BERA sent should be `MINIMUM_YEET_POINT`. All the following Yeets should be at least `MINIMUM_YEET_POINT / POT_DIVISION`. However, if an attacker sends BERA to the fallback or receive function, the attacker can add 1 WEI to `totalPot`.

```
/// @notice Add a fallback function to accept BERA
fallback() external payable {
    //Sucks to be you ;)
    potToWinner += msg.value;
    emit Yeetard(msg.sender, block.timestamp, potToWinner,
    _minimumYeetPoint(potToWinner), roundNumber);
}
```

```
/// @notice Add a receive function to accept BERA
receive() external payable {
    //Sucks to be you ;)
    potToWinner += msg.value;
    emit Yeetard(msg.sender, block.timestamp, potToWinner,
    _minimumYeetPoint(potToWinner), roundNumber);
}
```

This causes the following Yeets to require considerably less BERA than `MINIMUM_YEET_POINT / POT_DIVISION`. Following users only have to pay 1 WEI to add a Yeet (because the `addYeetVolume` requires the value to be nonzero).

Impact

An attacker can obtain a number of entries in the Yeetback raffle using a small amount of BERA.

Recommendations

In the `_minimumYeetPoint` function, ensure the minimum amount for a Yeet is greater than or equal to the `MINIMUM_YEET_POINT` variable.

Remediation

This issue has been acknowledged by Sanguine Labs LTD, and a fix was implemented in commit 3944c308 ↗.

3.3. Unoptimized `getClaimableAmount` function

Target	Reward		
Category	Optimization	**Severity**	Medium
Likelihood	High	**Impact**	Medium

Description

The `getClaimableAmount` function returns the total reward amount that can be claimed from the Reward contract at the moment (including the amount already claimed):

```
function getClaimableAmount(address user) public view returns (uint256) {
    // (...)

    for (uint256 epoch = lastClaimedForEpoch[user] + 1; epoch < currentEpoch;
    epoch++) {
        // (...)
    }

    return totalClaimable;
}
```

The initial value of `lastClaimedForEpoch[user]` is zero. If a user has never claimed their reward before, this function iterates from the very first epoch to the current epoch, which can be inefficient if a number of epochs elapsed.

Note that the `claim` function invokes the `getClaimableAmount` function:

```
function claim() external {
    uint amountEarned = getClaimableAmount(msg.sender);
    require(amountEarned != 0, "Nothing to claim");
    require(token.balanceOf(address(this)) >= amountEarned, "Not enough tokens
    in contract");

    lastClaimedForEpoch[msg.sender] = currentEpoch - 1; // This should be the
    fix.
    token.transfer(msg.sender, amountEarned);
    emit Rewarded(msg.sender, amountEarned, block.timestamp);
}
```

Impact

The `getClaimableAmount` and `claim` function will spend a lot of gas if a number of epochs has elapsed. For instance, after three years from the creation of the Reward contract, a new user will have to spend around nine million gas in order to invoke these functions.

Recommendations

Consider optimizing the gas usage of the `getClaimableAmount` function.

Remediation

This issue has been acknowledged by Sanguine Labs LTD, and a fix was implemented in commit `551ed0a7`↗.

3.4. Bias of the `draftWinners` function in the Yeetback contract

Target	Yeetback		
Category	Business Logic	**Severity**	Medium
Likelihood	Medium	**Impact**	Medium

Description

For each round, the Yeetback contract randomly draws 10 winners (with replacement) whom the Yeetback pot for the round is evenly distributed to.

The following code is the `draftWinners` function, which samples 10 winners:

```
function draftWinners(uint256 randomNumber, uint256 round) private {
    uint256 potValue = potForRound[round];
    uint256 nrOfYeets = yeetsInRound[round].length;
    uint256 randomNumberCopy = randomNumber;
    uint16 nrOfWinners = 10;

    uint256 winnings = potValue / nrOfWinners;
    amountToWinners[round] = winnings;

    for (uint8 i; i < nrOfWinners; i++) {
        uint16 smallNumbers = uint16(randomNumberCopy & 0xFFFF);
        randomNumberCopy >>= 16;

        // Seed a new random number with some entropy from the large random
    number
        uint randomDataNumber = uint(keccak256(abi.encodePacked(smallNumbers,
    nrOfYeets)));
        uint winningYeetIndex = randomDataNumber % nrOfYeets; // index of the
    winning yeet
        address winnerAddress = yeetsInRound[round][winningYeetIndex];

        // Update amountToWinners and amountOfWins
        amountOfWins[round][winnerAddress] += 1;

        emit YeetbackWinner(round, winnerAddress, winnings);
    }
}
```

Each sampling is done by slicing two-byte data from the given random number, hashing the con-

catenation of the above two-byte slice and the number of Yeets in this round and calculating the hash number modulo the number of Yeets in this round.

In conclusion, 10 two-byte slices of the random number are respectively hashed with the number of Yeets to decide each winner of a sampling. This means that the contribution of randomness from the random number is only a two-byte size, which is insufficient for the law of large numbers to be applied.

Impact

The odds to win the Yeetback raffle is biased under the current algorithm. For instance, we have observed that, assuming there are 100 Yeets in the pot, the winning rate of the 63rd Yeet is 0.91% and the winning rate of the 87th Yeet is 1.10%.

An attacker can exploit this bias to unfairly maximize their expected return on the Yeetback contract.

Recommendations

Consider changing the winner-selection algorithm to remove the bias.

Remediation

This issue has been acknowledged by Sanguine Labs LTD, and a fix was implemented in commit `edf75dfb` ↗.

3.5. Precision loss in `getDistribution` resulting in revert

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

Description

When a user invokes the `yeet` function with their funds, the function distributes the funds to multiple pots and taxes.

```
function _yeet(uint256[] memory tokenIds) internal {
    // (...)

    (
        uint valueToPot,
        uint valueToYeetback,
        uint valueToStakers,
        uint publicGoods,
        uint teamRevenue
    ) = getDistribution(msg.value);

    // (...)
}

function getDistribution(uint256 yeetAmount) public view returns (uint256,
    uint256, uint256, uint256, uint256) {
    uint256 scale = gameSettings.SCALE();

    uint valueAfterTax = (yeetAmount / scale) * (scale - TAX_PER_YEET);
    uint valueToYeetBack = (yeetAmount / scale) * (YEETBACK_PERCENTAGE);
    uint valueToPot = (yeetAmount / scale) * (scale - YEETBACK_PERCENTAGE
    - TAX_PER_YEET);
    uint tax = yeetAmount - valueAfterTax;

    uint256 valueToStakers = (tax / scale) * TAX_TO_STAKERS;
    uint256 publicGoods = (tax / scale) * TAX_TO_PUBLIC_GOODS;
    uint256 teamRevenue = (tax / scale) * TAX_TO_TREASURY;

    require(valueToPot + valueToYeetBack + valueToStakers + publicGoods
    + teamRevenue == yeetAmount, "Yeet: Distribution error");
```

```
        return (valueToPot, valueToYeetBack, valueToStakers, publicGoods,
        teamRevenue);
    }
```

The `getDistribution` function tries to ensure that the sum of the distributed amounts is equal to the input amount. However, it should be noted that parts of the tax are calculated with rounding down. Because the dust amount caused by this rounding down is not handled, unless the `tax` is divisible by the `scale`, `getDistribution` reverts.

Impact

The `getDistribution` function reverts if the `tax` is not divisible by the `scale`. This prevents the `yeet` function from being invoked with specific amounts.

Recommendations

Consider removing the check mentioned above. We believe that the dust lost with this precision loss is negligible, and the potential configuration misses are checked when the configuration is changed.

Remediation

This issue has been acknowledged by Sanguine Labs LTD, and a fix was implemented in commit `a29e6dbf` ↗.

3.6. Incorrect comment

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

Description

The Reward contract distributes the predefined amount of tokens to the reward beneficiaries. The following code defines the amount of tokens to be distributed:

```
/// @notice The duration of each distribution period
uint256 private constant DISTRIBUTION_CHANGE = 7 days;
/// @notice The amount of tokens to be distributed in the first epoch
uint256 private constant STARTING_TOKEN_COUNT = 216_227 * 10 ** 18;
/// @notice The length of each epoch in seconds
uint256 private constant EPOCH_LENGTH = 1 days;

// (...)

constructor(IERC20 _token, RewardSettings _settings)
    OnlyYeetContract(msg.sender) {
    // (...)
    epochRewards[currentEpoch] = STARTING_TOKEN_COUNT / (DISTRIBUTION_CHANGE
    / EPOCH_LENGTH);
    // (...)
}
```

The comment of the `STARTING_TOKEN_COUNT` explains that `216_227 * 10 ** 18` is the amount of tokens to be distributed in the first epoch. However, it is actually the amount for the first distribution period, and the amount of tokens to be distributed in the first epoch will be 1/7th.

Impact

This finding documents the inconsistency between the code and the comment.

Recommendations

Consider revising the comment.

Remediation

This issue has been acknowledged by Sanguine Labs LTD, and a fix was implemented in commit `d8de4eb2` ↗.

4. 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.

4.1. Module: Reward.sol

Function: `addYeetVolume(address user, uint256 amount)`

This is available only for the `yeetContract` address. Before executing the main logic of this function, the epoch will be updated if the current epoch has ended. The function increases the `userYeetVolume` by the `amount` for the provided `user` address and for the current epoch. Additionally, the `totalYeetVolume` is updated by the provided `amount`.

Inputs

- `user`
  - **Control**: N/A.
  - **Constraints**: Cannot be zero address.
  - **Impact**: The address of the user-added volume.
- `amount`
  - **Control**: N/A.
  - **Constraints**: Must be greater than zero.
  - **Impact**: The amount of volume.

Branches and code coverage (including function calls)

Intended branches

- `userYeetVolume` is updated properly.
  - ☑ Test coverage
- `totalYeetVolume` is updated properly.
  - ☑ Test coverage
- If `_shouldEndEpoch` is true, the epoch is updated.
  - ☑ Test coverage

Negative behavior

- `amount == 0`.
  - ☐ Test coverage
- User is zero address.

- ☐ Test coverage

Function call analysis

- `_shouldEndEpoch()`
  - **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?** Returns true if `block.timestamp` is greater than or equal to the `currentEpochEnd` — otherwise, it returns false. If the current epoch ends, the `_endEpoch` function is triggered.
- `_endEpoch()`
  - **What is controllable?** N/A.
  - **Argument control?**: 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?**: The function updates the epoch contract state; the `currentEpoch` is incremented, the `currentEpochStart` is updated using the current `currentEpochEnd`, and after that, the `currentEpochEnd` is set to the `currentEpochStart + EPOCH_LENGTH`. Also, this function updates the `epochRewards` for the new epoch.

Function: `claim()`

This function allows the caller to claim earned rewards. To get the reward, the caller should add YEET volume before. The caller will receive their part of the reward, corresponding to their part of the total YEET. The amount earned is calculated using the `getClaimableAmount` function, which gets `userYeetVolume` and `totalYeetVolume` for each epoch since the last claimed epoch for the caller to calculate the user's part of the reward for each epoch.

Inputs

- `msg.sender`
  - **Control**: N/A.
  - **Constraints**: If the caller hasn't added any YEET since the last `lastClaimedForEpoch`, this function will revert.
  - **Impact**: The reward claimer.

Branches and code coverage (including function calls)

Intended branches

- The reward for the first user's claim is calculated properly.
  - ☑ Test coverage
- The reward for the claim of any user but the first is calculated properly and does not take into account the previously claimed reward.
  - ☑ Test coverage

Negative behavior

- Caller has never added YEET volume.
  - ☑ Test coverage
- The second `claim` call after a successful claim is reverted.
  - ☐ Test coverage

Function call analysis

- `getClaimableAmount(msg.sender)`
  - **What is controllable?** N/A.
  - **If the return value is controllable, how is it used and how can it go wrong?** Return the earned reward for the `msg.sender` since the last `lastClaimedForEpoch`.
  - **What happens if it reverts, reenters or does other unusual control flow?**: N/A.
- `token.transfer(msg.sender, amountEarned);`
  - **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?** Transfer reward tokens to the caller.

4.2. Module: Yeet.sol

Function: `claim()`

The function allows the winner to claim the reward. If `msg.sender` does not have any reward, the function will be reverted.

Branches and code coverage

Intended branches

- The caller owns the reward and receives it successfully.
  - ☑ Test coverage

Negative behavior

- Caller does not have a reward.
  - ☐ Negative test
- After a successful claim, the next double-claim call is failed.
  - ☐ Negative test

Function: `_yeet(uint256[] tokenIds)`

This function allows the caller to `yeet` the native token to participate in the game. This function can be called with empty `tokenIds` and with the `tokenIds` array that is controlled by the caller. The `tokenIds` array contains the NFTs owned by the user, which allows them get the additional boost for the provided native token amount.

Inputs

- `tokenIds`
  - **Control**: Full control.
  - **Constraints**: User should be the owner of provided `tokenIds`.
  - **Impact**: The NFTs owned by the user determines the NFT boost that will be added to the provided native token amount.

Branches and code coverage

Intended branches

- The updated `publicGoodsAmount`, `treasuryRevenueAmount`, `potToYeetback`, and `potToWinner` values correspond to the expected values.
  - ☐ Test coverage
- The state of the `rewardsContract` is updated properly.
  - ☐ Test coverage
- The expected amount of tokens are deposited to the `stakingContract`.
  - ☐ Test coverage

Negative behavior

- The `tokenIds` contains duplicates.
  - ☐ Negative test
- The caller does not own the provided `tokenIds`.
  - ☐ Negative test

Function call analysis

- `this._minimumYeetPoint(potToWinner)`

- **What is controllable?** N/A.
- **If the return value is controllable, how is it used and how can it go wrong?** The function returns the minimum amout of tokens needed to yeet; it can be the amount determined by the global `MINIMUM_YEET_POINT` if `potToWinner` is zero or if the `isBoostrapPhase` returns true. Otherwise, it will be `potToWinner / POT_DIVISION`. The issue can occur in the case where the bootstrap phase is skipped and the `potToWinner` is not zero, but this value is very small, such that the return value will be less than `MINIMUM_YEET_POINT`. Accordingly, the user will be able to perform the function while contributing a small amount of funds.
- **What happens if it reverts, reenters or does other unusual control flow?** N/A.

- `this.getDistribution(msg.value)`
  - **What is controllable?** `msg.value`.
  - **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?** This function splits the provided funds into fractional parts according to the current settings; it returns the value to the pot, to the Yeetback, to the stakers, to public goods, and to the treasury. This function will revert if the sum of the parts is not equal to the original `msg.value`.
- `yeetback.addYeetsInRound(roundNumber, msg.sender)`
  - **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?** The `roundNumber` should not be zero.
- `getBoostedValue(msg.sender, valueToPot, tokenIds)`
  - **What is controllable?** `tokenIds`.
  - **If the return value is controllable, how is it used and how can it go wrong?** If a user provides the duplicate of token IDs, they will be counted as unique IDs.
  - **What happens if it reverts, reenters or does other unusual control flow?** The function will revert if the caller does not own a token ID or if the token ID is `isEligibleForBoost`. If the `yeetardsNFTsAddress` address is zero, the token IDs will not be used at all.
- `rewardsContract.addYeetVolume(msg.sender, boostedValue)`
  - **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?** No problems.
- `stakingContract.depositReward{value: valueToStakers}()`
  - **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.

5. Assessment Results

At the time of our assessment, the reviewed code was not deployed to the Berachain Mainnet.

During our assessment on the scoped Yeet contracts, we discovered six findings. No critical issues were found. Four findings were of medium impact, one was of low impact, and the remaining finding was informational in nature.

5.1. 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.