Zellic

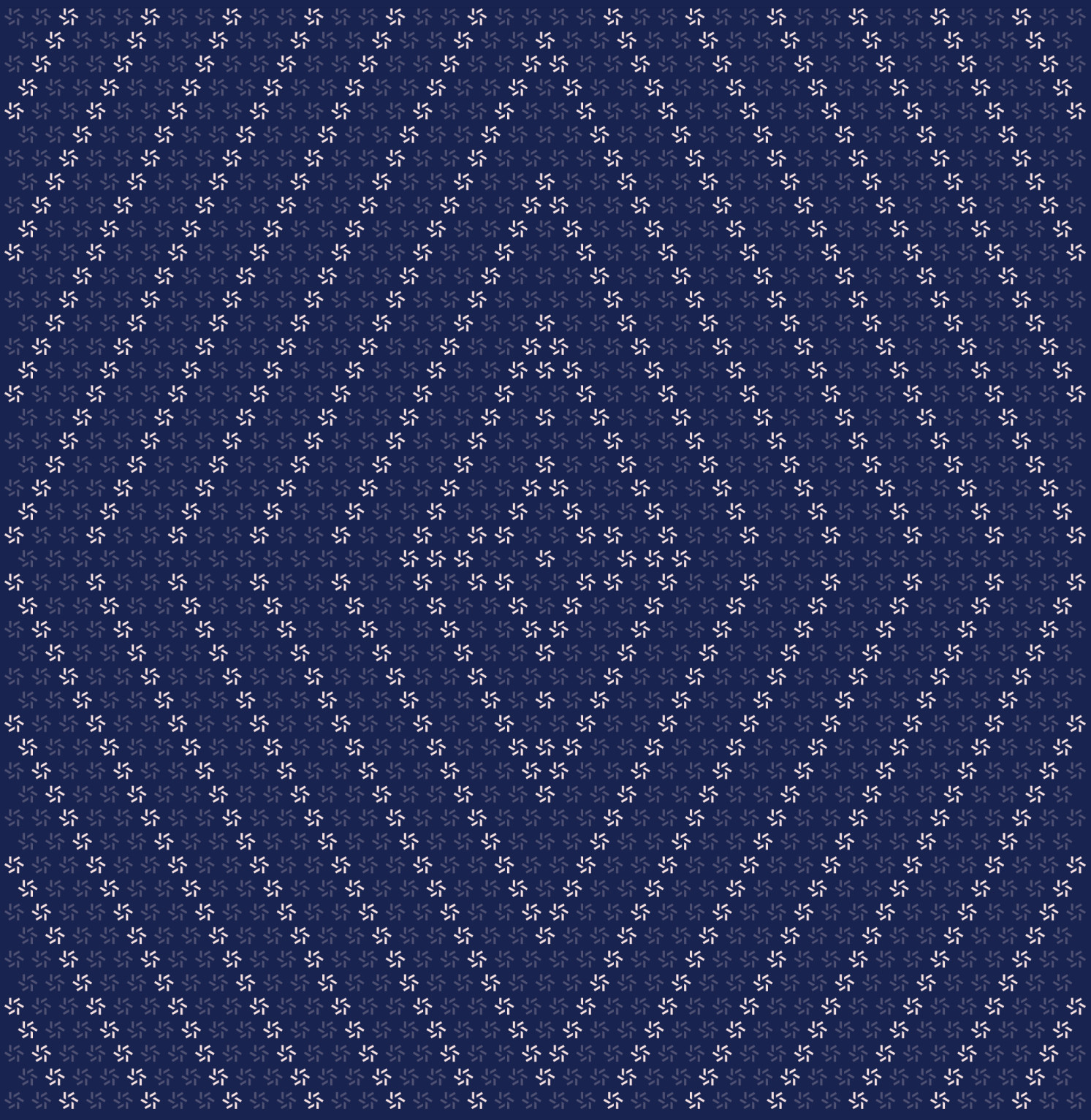
Prepared for
King koala
Sanguine Labs LTD

Prepared by
Kamensec
Juchang Lee
Zellic

October 21, 2024

Yeet

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 Sanguine Labs LTD from October 8th to October 12th, 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:

- Do general ERC-4626 bugs exist?
- Is the xKDK locked-token reward management appropriate?
- Could rewards be lost in the case of incorrect reinvestment in swap data?
- Are Circuit Breaker tokens syncing properly?

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 eight findings. Three critical issues were found. One was of high impact, one was of medium impact, two were of low impact, and the remaining finding was informational in nature.

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

Breakdown of Finding Impacts

Impact Level	Count
Critical	3
High	1
Medium	1
Low	2
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.

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:

Yeet Contracts

Type	Solidity
Platform	EVM-compatible
Target	contracts
Repository	https://github.com/0xKingKoala/contracts ↗
Version	`a93bcdf0dac7d094ced1d147ef09de70d75aceed`

Programs	`StakeV2.sol`
	`contracts/MoneyBrinter.sol`
	`contracts/VaultCircuitBreaker.sol`
	`contracts/Zapper.sol`
	`contracts/eip7265/src/core/CircuitBreaker.sol`
	`contracts/eip7265/src/static/Structs.sol`
	`contracts/eip7265/src/utils/LimiterLib.sol`

2.4. Project Overview

Zellic was contracted to perform a security assessment for a total of 1.8 person-weeks. The assessment was conducted by two consultants over the course of four and a half calendar days.

Contact Information

The following project managers were 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:

Kamensec
Engineer
dimitri@zellic.io

Juchang Lee
Engineer
lee@zellic.io

2.5. Project Timeline

The key dates of the engagement are detailed below.

October 9, 2024	Kick-off call
October 9, 2024	Start of primary review period
October 16, 2024	End of primary review period

3. Detailed Findings

3.1. MoneyBrinter susceptible to stealth deposit

Target	MoneyBrinter.sol		
Category	Coding Mistakes	**Severity**	Critical
Likelihood	High	**Impact**	Critical

Description

The MoneyBrinter contract is a yield-bearing ERC-4626 vault. Due to missing defenses, it is possible to manipulate the exchange rate of shares to underlying assets during deposits to cause loss of user funds.

When depositing, the conversion of underlying assets to shares behaves as follows:

```
function _convertToShares(uint256 assets, Math.Rounding rounding)
    internal view virtual returns (uint256) {
    return assets.mulDiv(totalSupply() + 10 ** _decimalsOffset(),
    totalAssets() + 1, rounding);
}
```

If the `assets * (totalSupply() + 10 ** _decimalOffset) < totalAsset + 1`, this will lead to cases where the resultant `shares == 0` while `safeTransferFrom(_asset, caller, address(this), assets)` succeeds. Assets will be transferred from the depositor, while the depositor is not credited any ownership of future yield.

MoneyBrinter partially prevents direct deposits to the vault by only counting assets as `_totalAssets = IPlugin(beradromeFarmPlugin).balanceOf(address(this))`. However, the `BeradromeFarmPlugin` provides `depositFor`↗, allowing deposits on behalf of a user. This allows us to arbitrarily inflate the denominator of the share calculation. The first depositor can purchase one share, subsequently donating sufficient assets to the Beradrome farm and deflating subsequent depositors. The attacker can eventually claim their donation back as they are the only shareholder.

The following test case illustrates the behavior described:

```
function testDepositInflationAttack(uint256 depositAmount) public {
        depositAmount = bound(depositAmount, 1, 10e6 ether);
        uint256 expectedShares = vault.previewDeposit(depositAmount);

        fundUser(bob, depositAmount);
        approveToVault(bob, depositAmount);
        console.log("vault.totalSupply()", vault.totalSupply());
```

```
        console.log("asset.balanceOf(address(vault))",
    asset.balanceOf(address(vault)));
        vm.startPrank(alice);
        asset.approve(address(vault.beradromeFarmPlugin()), depositAmount
    * 2);
        IPlugin(vault.beradromeFarmPlugin()).depositFor(address(vault),
    depositAmount * 2);
        vm.stopPrank();
        console.log("vault.totalSupply()", vault.totalSupply());
        console.log("asset.balanceOf(address(vault))",
    asset.balanceOf(address(vault)));

        depositIntoVaultAndVerify(bob, depositAmount, expectedShares, true,
    ""); // Reverts due to 0 shares being minted
    }
```

Impact

First depositors are able to steal all funds from subsequent depositors.

Recommendations

We recommend using `_decimalOffset` overrides as a way to minimize the likelihood of this issue being exploited. Alternative strategies could involve an internally tracked deposits variable that prevents assets from being inflated through stealth deposits, and the use of a trusted first depositor.

Remediation

This issue has been acknowledged by Sanguine Labs LTD, and a fix was implemented in commit `26fd3cb4` ↗.

3.2. Lack of user access control in StakeV2

Target	StakeV2.sol		
Category	Coding Mistakes	**Severity**	Critical
Likelihood	High	**Impact**	Critical

Description

The Manager contract in StakeV2.sol does not have access control. The functions `addManager` and `removeManager` lack any access control, allowing arbitrary addresses to be registered or removed as a manager.

```
function addManager(address _manager) external override {
    require(!managers[_manager], "Manager already exists");
    require(_manager != address(0), "Invalid address");
    managers[_manager] = true;
}

function removeManager(address _manager) external override {
    require(managers[_manager], "Manager does not exist");
    require(_manager != address(0), "Invalid address");
    managers[_manager] = false;
}
```

Impact

In StakeV2, the function `executeRewardDistribution` can be arbitrarily utilized by users registered as managers. This means that any user can execute the reward distribution, potentially leading to a loss of funds.

Recommendations

We recommend adding some access control to ensure that only the owner or existing managers can execute the `addManager` and `removeManager` functions.

Remediation

This issue has been acknowledged by Sanguine Labs LTD, and a fix was implemented in commit `2d4a4596` ↗.

3.3. Formulaic error allows stakers to steal excess rewards

Target	StakeV2.sol		
Category	Coding Mistakes	**Severity**	Critical
Likelihood	High	**Impact**	Critical

Description

The StakingV2 contract tracks total shares staked over reward-bearing epochs. The contract makes formulaic errors due to incorrect units of measurements being used during subtraction of `earned[msg.sender]`. This leads to less rewards being deducted than are withdrawn by the user.

The `StakeV2._verifyAndPrepareClaim()` internal function used to handle internal reward accounting calculates the shares to deduct as follows:

```
uint256 sharesToWithdraw = (amountToWithdraw * rewardIndex)
    / totalVaultShares;
require(sharesToWithdraw <= totalUserVaultShares, "Withdrawal amount exceeds
    available rewards");

uint256 rewardsToClaim = (sharesToWithdraw * rewardIndex) / totalVaultShares;
require(rewardsToClaim <= userReward, "Withdrawal amount exceeds rewards
    earned by the user");
earned[msg.sender] -= rewardsToClaim;
totalVaultShares -= sharesToWithdraw;
```

However, if we attempt to derive the units of measurement for `earned[msg.sender]`,

```
earned[account] = stakingToken * rewardPerStakingToken
earned[account] = reward
```

As we can see on line 127 in `executeRewardDistribution()`, the `totalVaultShares` represents the total reward held by the contract. As such, the subtraction from `earned[msg.sender] < totalVaultShares`.

The following test case illustrates that despite the staking contract not having any `totalVaultShares` (implying full withdrawal of rewards), the only user staked still has rewards remaining.

```
function test_fails_to_decrement_earned() public {
    // Setup
    address staker1 = makeAddr("staker_one");
```

```
    address staker2 = makeAddr("staker_two");

    // 1. stake() with 10 ether
    vm.startPrank(staker1);
    token.mint(address(staker1), 1000 ether);
    token.approve(address(stakeV2), 10 ether);
    stakeV2.stake(10 ether);
    vm.stopPrank();

    // Assert Initial Test State
    uint256 initialStakeBalance = stakeV2.balanceOf(staker1);
    uint256 earnedStaker1 = stakeV2.calculateRewardsEarned(staker1);
    assertEq(initialStakeBalance, 10 ether, "invalid initialStakeBalance");
    assertEq(earnedStaker1, 0, "invalid earnedStaker1");

    // 2. depositReward()
    uint256 rewardAmount = 1 ether;
    vm.deal(address(this), rewardAmount);
    stakeV2.depositReward{value: rewardAmount}();

    // 3. executeRewardDistribution()
    uint256 expectedIslandTokens = 0 ether;
    mockZapper.setReturnValues(expectedIslandTokens, rewardAmount); //
    simulates the zapper logic at a 1:1 ratio for
    vaultsSharesMinted:accumulatedRewards
    uint256 expectedRewardIndex = rewardAmount
    * 1 ether / stakeV2.totalSupply();

    vm.expectEmit(true, true, false, true);
    emit StakeV2.RewardsDistributed(rewardAmount, expectedRewardIndex);

    stakeV2.executeRewardDistribution(
        IZapper.SingleTokenSwap(0, 0, 0, address(0), ""),
        IZapper.SingleTokenSwap(0, 0, 0, address(0), ""),
        IZapper.KodiakVaultStakingParams(address(0), 0, 0, 0, 0, 0,
    address(0)),
        IZapper.VaultDepositParams(address(0), address(0), 0)
    );

    // Assert Intermediate Test State
    assertEq(stakeV2.accumulatedRewards(), 0, "Accumulated rewards should be
    reset to 0");
    assertEq(stakeV2.rewardIndex(), expectedRewardIndex, "Reward index should
    be updated correctly");

    // 4. claimRewardsInNative()
    vm.startPrank(staker1);
```

```
    uint256 expectedTotalUserVaultShares = 1e19;
    uint256 amountToWithdraw = expectedTotalUserVaultShares;

    mockZapper.setZapOutNativeReturn(1 ether); // simulates the zapper logic
    at a 1:1 ratio for vaultsSharesMinted:accumulatedRewards
    stakeV2.claimRewardsInNative(
        amountToWithdraw,
        IZapper.SingleTokenSwap(0, 0, 0, address(0), ""),
        IZapper.SingleTokenSwap(0, 0, 0, address(0), ""),
        IZapper.KodiakVaultUnstakingParams(address(0), 0, 0, address(0)),
        IZapper.VaultRedeemParams(address(0), address(0), 0, 0)
    );
    // vm.stopPrank();

    // Assert Post Test State
    uint256 finalVaultShares = stakeV2.totalVaultShares();
    uint256 finalEarnedStaker1 = stakeV2.calculateRewardsEarned(staker1);
    assertEq(finalVaultShares, 0 ether, "invalid finalVaultShares");
    assertEq(finalEarnedStaker1, 0, "invalid finalEarnedStaker1"); // Reverts
    due to left over rewards
}
```

Impact

Users may claim more rewards than they are eligible for, resulting in a potential loss of funds for those who are slower to claim their rewards.

Recommendations

We recommend revisiting the `_verifyAndClaim()` function's internal math operations, ensuring that the correct units of measurement are adhered to.

Remediation

This issue has been acknowledged by Sanguine Labs LTD, and a fix was implemented in commit 2172fc05↗.

3.4. The `rewardIndex` only increases, leading to eventual inability to claim rewards

Target	StakeV2.sol		
Category	Coding Mistakes	**Severity**	High
Likelihood	Medium	**Impact**	High

Description

The StakingV2 contract uses `rewardIndex`, which acts as an epoch-based rewards tracker. Due to rounding errors in Solidity, it is possible that `rewardIndex` increases to the point where `_verifyAndPrepareClaim()` will revert.

The `rewardIndex` variable is an ever-increasing denominator in the `totalUserVaultShares` calculation on line 348, while `totalVaultShares` (the numerator) can decrease in value over time.

This leads to cases where long-term use of the staking contract may result in `totalUserVaultShares = 0` when `userReward * totalVaultShares < rewardIndex`.

The following test case illustrates cases where the user is claiming full withdrawals, but as `rewardIndex` increases, `_verifyAndPrepareClaim()` eventually reverts with `withdrawal amount exceeds available rewards`.

```
function test_increment_rewardIndex_decrement_totalVaultShares() public {
    // Setup
    address staker1 = makeAddr("staker_one");
    address staker2 = makeAddr("staker_two");

    // 1. stake() with 10 ether
    vm.startPrank(staker1);
    token.mint(address(staker1), 1000 ether);
    token.approve(address(stakeV2), 10 ether);
    stakeV2.stake(10 ether);
    vm.stopPrank();

    // Assert Initial Test State
    uint256 initialStakeBalance = stakeV2.balanceOf(staker1);
    uint256 earnedStaker1 = stakeV2.calculateRewardsEarned(staker1);
    assertEq(initialStakeBalance, 10 ether, "invalid initialStakeBalance");
    assertEq(earnedStaker1, 0, "invalid earnedStaker1");
    uint256 totalIterations = 20;
    for(uint256 x = 0; x < totalIterations; ++x) {
        // 2. depositReward()
```

```
        uint256 rewardAmount = 1 ether;
        vm.deal(address(this), rewardAmount);
        stakeV2.depositReward{value:  rewardAmount}();

        // 3. executeRewardDistribution()
        uint256 expectedIslandTokens = 0 ether;
        mockZapper.setReturnValues(expectedIslandTokens, rewardAmount); //
    simulates the zapper logic at a 1:1 ratio for
    vaultsSharesMinted:accumulatedRewards
        uint256 expectedRewardIndex = (x + 1) * rewardAmount
    * 1 ether / stakeV2.totalSupply();

        vm.expectEmit(true, true, false, true);
        emit StakeV2.RewardsDistributed(rewardAmount, expectedRewardIndex);

        stakeV2.executeRewardDistribution(
            IZapper.SingleTokenSwap(0, 0, 0, address(0), ""),
            IZapper.SingleTokenSwap(0, 0, 0, address(0), ""),
            IZapper.KodiakVaultStakingParams(address(0), 0, 0, 0, 0, 0,
    address(0)),
            IZapper.VaultDepositParams(address(0), address(0), 0)
        );

        // Assert Intermediate Test State
        assertEq(stakeV2.accumulatedRewards(), 0, "Accumulated rewards should
    be reset to 0");
        assertEq(stakeV2.rewardIndex(), expectedRewardIndex, "Reward index
    should be updated correctly");

        // 4. claimRewardsInNative()
        vm.startPrank(staker1);
        uint256 totalUserVaultShares = rewardAmount
    * stakeV2.totalVaultShares() / stakeV2.rewardIndex();
        uint256 amountToWithdraw =  stakeV2.totalVaultShares()
    * stakeV2.totalVaultShares() / stakeV2.rewardIndex(); // @audit withdraw
    total Vault shares

        mockZapper.setZapOutNativeReturn(amountToWithdraw); // simulates the
    zapper logic at a 1:1 ratio for vaultsSharesMinted:accumulatedRewards
        stakeV2.claimRewardsInNative( // REVERT: On the 20th claim this reverts
            amountToWithdraw,
            IZapper.SingleTokenSwap(0, 0, 0, address(0), ""),
            IZapper.SingleTokenSwap(0, 0, 0, address(0), ""),
            IZapper.KodiakVaultUnstakingParams(address(0), 0, 0, address(0)),
            IZapper.VaultRedeemParams(address(0), address(0), 0, 0)
        );
    }
```

```
}
```

Impact

Due to cumulative rounding errors, users with smaller rewards are incapable of withdrawing rewards. Over time, this issue worsens as `rewardIndex` is never decremented.

Recommendations

We recommend simplifying the `_verifyAndPrepareClaim()` function as much as possible.

Stake.sol allowed full, direct withdrawals of exactly all `earned[msg.sender]`. StakeV2.sol uses the exact same metric for `earned[msg.sender]`, which is units of `rewards`, meaning that withdrawals can simply also be in reward unit measurements rather than in some fractional representation that must be converted to rewards and vice versa.

Remediation

This issue has been acknowledged by Sanguine Labs LTD, and a fix was implemented in commit `2172fc05` ↗.

3.5. Manager can leak funds during reward distribution

Target	StakeV2.sol		
Category	Coding Mistakes	**Severity**	Medium
Likelihood	Low	**Impact**	Medium

Description

The StakeV2 contract expects an `onlyManager`-approved account to execute the reward distribution. If the Zapper contract has an approved Kodiak Vault that can be rounded down to zero, it is possible for rewards to be lost to stakers in StakeV2 during distribution.

The `executeRewardDistribution()` function validates the results from zapping as follows:

```
(uint256 amountIslandTokens, uint256 vaultSharesMinted) =
    zapper.zapInNative{value: amountToDistribute}(swap0, swap1, stakingParams,
    vaultParams);
require(amountIslandTokens == 0, "Amount of island tokens must be 0");
totalVaultShares += vaultSharesMinted;
```

This code block conducts checks to ensure island tokens have not been minted; however, it does check that `vaultSharesMinted == 0`.

Impact

Either intentionally or accidentally, if vault shares are rounded down to zero, the StakeV2 contract transfers funds without receiving the rights to the rewards those funds generate.

Manager is a trusted role; therefore, likelihood of this attack is low. However, the total user reward is vulnerable, leading to a higher impact.

Recommendations

Reward distribution should ensure that `vaultSharesMinted != 0` to avoid easy compromises of vault-share manipulation.

Remediation

This issue has been acknowledged by Sanguine Labs LTD, and a fix was implemented in commit `4ef36a6b`.

3.6. Zapper contract leaks excess token balance to OBRouter

Target	Zapper.sol		
Category	Coding Mistakes	**Severity**	Low
Likelihood	Low	**Impact**	Low

Description

The Zapper contract helps manage swaps on the Ooga Booga Router (OBRouter) and liquidity provisions on the Kodiak Vault. The contract makes trust assumptions about the `inputAmount` passed into the router; however, the router breaks those assumptions, causing loss of funds under very specific circumstances.

When a user executes `zapIn()` (as well as any other zap-in flow), lines 194–196 conduct a balance transfer as follows:

```
IERC20(inputToken).safeTransferFrom(
    _msgSender(), address(this), swapToToken0.inputAmount
    + swapToToken1.inputAmount
);
```

This infers that only balances transferred should be available to be swapped. However, the OBRouter (the contract can be found here ↗), attempts to change the balance to the maximum held by the recipient.

```
 // Support rebasing tokens by allowing the user to trade the entire balance
if (tokenInfo.inputAmount == 0) {
    tokenInfo.inputAmount
    = IERC20(tokenInfo.inputToken).balanceOf(msg.sender);
}
IERC20(tokenInfo.inputToken).safeTransferFrom(msg.sender, executor,
    tokenInfo.inputAmount);
```

Impact

If Zapper holds any balance, a user is able to conduct a swap specifying `tokenInfo.inputAmount == 0`. This will not transfer any token from the user to Zapper; however, the trade will succeed, transferring the entire balance of Zapper for use in the OBRouter.

The Zapper contract is not designed to capture and retain token balances of any kind, and hence the

likelihood of this vulnerability is low. The loss of funds is capped at the excess left through normal interaction with the Zapper and OBRouter contracts or direct transfers to Zapper.

Recommendations

If the requirement to transfer max balances of the Zapper contract is unintended, we recommend reverting on `swapTokenInfo.inputAmount == 0`.

Remediation

This issue has been acknowledged by Sanguine Labs LTD, and a fix was implemented in commit 7375a58c ↗.

3.7. Circuit Breaker's unsafe casting results in errors during liquidity tracking

Target	CircuitBreaker.sol		
Category	Coding Mistakes	**Severity**	Low
Likelihood	Low	**Impact**	Low

Description

The CircuitBreaker contract tracks liquidity changes within an interval. Due to unsafe typecasting, the liquidity track can be misrepresented when exceeding values of $2^{255} - 1$.

The `_onTokenOutflow()` function conducts the following record change;

```
limiter.recordChange(-int256(_amount), WITHDRAWAL_PERIOD, TICK_LENGTH);
```

However, casting any integer that exceeds $2^{255} - 1$ will end up changing the amount to a negative number. This will end up incrementing the limiter instead of decrementing.

Impact

It is possible to bypass liquidity changes within an interval by incrementing during large withdrawals. However, this requires token balances in excess of $2^{255} - 1$.

Recommendations

We recommend the use of safe typecasting to ensure math operations are not reversed unexpectedly.

Remediation

This issue has been acknowledged by Sanguine Labs LTD, and a fix was implemented in commit `8e0bf0ce` ↗.

3.8. VaultCircuitBreaker constructor is missing zero-address checks for admin

Target	VaultCircuitBreaker.sol		
Category	Coding Mistakes	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

In the VaultCircuitBreaker contract, when initializing the admin in the constructor, no zero-address check is performed.

```
constructor(
    address _admin,
    uint256 _rateLimitCooldownPeriod,
    uint256 _withdrawlPeriod,
    uint256 _liquidityTickLength
) {
    admin = _admin;
    rateLimitCooldownPeriod = _rateLimitCooldownPeriod;
    WITHDRAWAL_PERIOD = _withdrawlPeriod;
    TICK_LENGTH = _liquidityTickLength;
    isOperational = true;
}
```

Impact

If a zero address is included due to incorrect distribution in the admin address, there is no way to reset it, which could lead to potential issues.

Recommendations

We recommend adding the requisite checks to the constructor.

Remediation

This issue has been acknowledged by Sanguine Labs LTD, and a fix was implemented in commit 8e0bf0ce ↗.

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. Inconsistencies in modifiers can lead to missing checks

Several modifiers have been applied inconsistently, which may lead to future developers mistakenly believing that a specific function has been properly checked. This inconsistency undermines the clarity of the code and poses long-term sustainability risks for future maintenance and development.

This issue has been acknowledged by Sanguine Labs LTD, and a fix was implemented in commit `f5434f3d` ↗.

4.2. CircuitBreaker complexity introduces unnecessary attack surfaces

The CircuitBreaker, aka EIP-7265 ↗, is a contract that handles protocol-wide token-withdrawal circuit-breaking functionality once liquidity exceeds tolerable transfer limits. However, this contract is both erroneous and complicated.

1. It tracks liquidity changes, which does not capture token volatility. This means, as token value decreases, more liquidity is needed to meet the same transfer demands. Likewise, as asset price increases, liquidity transferred will be significantly lower and huge-value transfers are required to hit the circuit's liquidity threshold. Requiring manual intervention to update liquidity params might not be fast enough for volatile tokens to provide meaningful protection against value changes.

2. When the circuit is triggered, if reverts are disabled, `lockedFunds` are tracked. After the `rateLimit` is removed, the locked funds are instantly claimable. Anyone with sufficient capital can hold funds hostage, transferring large amounts in order to trigger the Circuit Breaker. An admin must intervene, but if they remove the rate limit so that some users can withdraw, the offending user that triggered the rate limit can continue to pull the entire balance at the exact same time as all other users. There is no preferential withdrawal to earlier users; as such, there is no defense against massive price manipulation as a result of the sudden withdrawal.

3. There is also no recovery mechanisms for hacked assets. Admin members cannot rescue funds after the locked funds are recorded and redistribute them as intended.

With these points, it is strongly worth considering the need for circuit-breaker functionality, the need for an external contract handling protocol-wide protection, and whether iterating through all trans-

fers after each transfer provides sufficient reward to compensate problems if additional flaws in the circuit-breaker logic are discovered.

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: MoneyBrinter.sol

Function: `compound(address[] swapInputTokens, IZapper.SingleTokenSwap[] swapToToken0, IZapper.SingleTokenSwap[] swapToToken1, IZapper.KodiakVaultStakingParams stakingParams, IZapper.VaultDepositParams vaultStakingParams)`

This function compounds rewards by swapping harvested tokens and staking in the Kodiak Vault, depositing into Beradrome farm.

Inputs

- `swapInputTokens`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Must be equal to the sum of the lengths of `swapToToken0` and `swapToToken1`
  - **Impact**: Information of input-token addresses for swap.
- `swapToToken0`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: N/A.
  - **Impact**: Information of swapping swapInputTokens to token0.
- `swapToToken1`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: N/A.
  - **Impact**: Information of swapping swapInputTokens to token1.
- `stakingParams`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: The `stakingParams.receiver` must be the contract address.
  - **Impact**: Information of staking in Kodiak Vault.
- `vaultStakingParams`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: N/A.
  - **Impact**: Information of depositing into the vault.

Branches and code coverage

Intended branches

- Check if `stakingParams.receiver` is the contract address.
  - ☑ Test coverage
- Check if the length of `swapInputTokens` is equal to the sum of the lengths of `swapToToken0` and `swapToToken1`.
  - ☑ Test coverage
- Zaps in tokens.
  - ☑ Test coverage
- Check if `islandTokensMinted` is greater than `stakingParams.amountSharesMin`.
  - ☑ Test coverage
- Check if `vaultSharesMinted` is zero.
  - ☑ Test coverage

Negative behavior

- The `stakingParams.receiver` is not the contract address.
  - ☐ Negative test
- The length of `swapInputTokens` is not equal to the sum of the lengths of `swapToToken0` and `swapToToken1`.
  - ☐ Negative test
- `islandTokensMinted` is less than `stakingParams.amountSharesMin`.
  - ☐ Negative test
- `vaultSharesMinted` is not zero.
  - ☐ Negative test

Function: `harvestKodiakRewards(address[] previousKodiakRewardTokens)`

This function harvests rewards from the Kodiak rewards contract.

Inputs

- `previousKodiakRewardTokens`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: N/A.
  - **Impact**: Information of addresses that previous Kodiak reward tokens used

5.2. Module: StakeV2.sol

Function: `claimRewardsInNative(uint256 amountToWithdraw, IZapper.SingleTokenSwap swapData0, IZapper.SingleTokenSwap swapData1, IZapper.KodiakVaultUnstakingParams unstakeParams, IZapper.VaultRedeemParams redeemParams)`

This function allows a user to claim their rewards in native Ethereum by performing a zap-out operation.

Inputs

- `amountToWithdraw`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: N/A.
  - **Impact**: The amount of rewards the user intends to claim.
- `swapData0`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: N/A.
  - **Impact**: Information of swapping to token0 during the zap-out operation.
- `swapData1`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: N/A.
  - **Impact**: Information of swapping to token1 during the zap-out operation.
- `unstakeParams`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: N/A.
  - **Impact**: Information of unstake during the zap-out operation.
- `redeemParams`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: N/A.
  - **Impact**: Information of redeem during the zap-out operation.

Branches and code coverage

Intended branches

- Verify amount and `redeemParams` and prepare `redeemParams` to call `zapOutNative`.
  - ☑ Test coverage

Function: `claimRewardsInToken0(uint256 amountToWithdraw, IZapper.SingleTokenSwap swapData, IZapper.KodiakVaultUnstakingParams unstakeParams, IZapper.VaultRedeemParams redeemParams)`

This function allows a user to claim their rewards in token0 by performing a zap-out operation.

Inputs

- `amountToWithdraw`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: N/A.
  - **Impact**: The amount of rewards the user intends to claim.
- `swapData`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: N/A.
  - **Impact**: Information of swap data during the zap-out operation.
- `unstakeParams`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: N/A.
  - **Impact**: Information of unstake during the zap-out operation.
- `redeemParams`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: N/A.
  - **Impact**: Information of redeem during the zap-out operation.

Branches and code coverage

Intended branches

- Verify amount and `redeemParams` and prepare `redeemParams` to call `zapOutToToken0`.
  - ☐ Test coverage

Function: `claimRewardsInToken1(uint256 amountToWithdraw, IZapper.SingleTokenSwap swapData, IZapper.KodiakVaultUnstakingParams unstakeParams, IZapper.VaultRedeemParams redeemParams)`

This function allows a user to claim their rewards in token1 by performing a zap-out operation.

Inputs

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

- **Constraints**: N/A.
- **Impact**: The amount of rewards the user intends to claim.

- `swapData`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: N/A.
  - **Impact**: Information of swap data during the zap-out operation.
- `unstakeParams`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: N/A.
  - **Impact**: Information of unstake during the zap-out operation.
- `redeemParams`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: N/A.
  - **Impact**: Information of redeem during the zap-out operation.

Branches and code coverage

Intended branches

- Verify amount and `redeemParams` and prepare `redeemParams` to call `zapOutToToken1`.
  - ☐ Test coverage

Function: `claimRewardsInToken(uint256 amountToWithdraw, address outputToken, IZapper.SingleTokenSwap swap0, IZapper.SingleTokenSwap swap1, IZapper.KodiakVaultUnstakingParams unstakeParams, IZapper.VaultRedeemParams redeemParams)`

This function allows a user to claim their rewards in outputToken by performing a zap-out operation.

Inputs

- `amountToWithdraw`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: N/A.
  - **Impact**: The amount of rewards the user intends to claim.
- `outputToken`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: N/A.
  - **Impact**: Token to be received after claiming.
- `swap0`
  - **Control**: Fully controlled by the caller.

- **Constraints**: N/A.
- **Impact**: Information swapping token0 to outputToken.

- `swap1`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: N/A.
  - **Impact**: Information swapping token1 to outputToken.
- `unstakeParams`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: N/A.
  - **Impact**: Information of unstake during the zap-out operation.
- `redeemParams`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: N/A.
  - **Impact**: Information of redeem during the zap-out operation.

Branches and code coverage

Intended branches

- Verify amount and `redeemParams` and prepare `redeemParams` to call `zapOut`.
  - ☐ Test coverage

Function: `rageQuit(uint256 index)`

This function is called when unstaking is necessary before the vesting period. It will be called in the dApp.

Inputs

- `index`
  - **Control**: Fully controled by the caller.
  - **Constraints**: N/A.
  - **Impact**: Index of vestings.

Function: `stake(uint256 amount)`

This is the stake amount of token in contract.

Inputs

- `amount`

- **Control**: Fully controlled by the caller.
- **Constraints**: Must be greater than zero.
- **Impact**: Amount of stake.

Branches and code coverage

Intended branches

- Check if amount is greater than zero.
  - ☑ Test coverage
- Receive amount of token from the caller.
  - ☑ Test coverage
- Add `balanceOf` and `totalSupply` as much as amount.
  - ☑ Test coverage

Negative behavior

- Amount is less than zero.
  - ☐ Negative test

Function: `startUnstake(uint256 unStakeAmount)`

A request to add unstaking to vesting reduces only the `balanceOf` and `totalSupply`. The tokens are yet to be sent to the caller.

Inputs

- `unStakeAmount`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Must be greater than zero.
  - **Impact**: Amount of unstake.

Branches and code coverage

Intended branches

- Check if `unStakeAmount` is greater than zero.
  - ☑ Test coverage
- Check if `stakedTimes` has not reached `STAKING_LIMIT`.
  - ☑ Test coverage
- Check if `unStakeAmount` is less than the caller's balance.
  - ☑ Test coverage
- Subtract `balanceOf` and `totalSupply` as much as `unStakeAmount`.

- ☑ Test coverage
- Push vesting request that contains start time, end time, and `unStakeAmount`.
  - ☑ Test coverage

Negative behavior

- The `unStakeAmount` is less than zero.
  - ☐ Negative test
- `stakedTimes` has reached `STAKING_LIMIT`.
  - ☐ Negative test
- The `unStakeAmount` is greater than the caller's balance.
  - ☑ Negative test

Function: `unstake(uint256 index)`

This is the actual unstake function. The unstaking process is carried out, and tokens are subsequently disbursed to the caller.

Inputs

- `index`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: N/A.
  - **Impact**: Index of vestings.

Branches and code coverage

Intended branches

- Check if vesting period has ended.
  - ☑ Test coverage

Negative behavior

- Vesting period has not ended.
  - ☐ Negative test

5.3. Module: Zapper.sol

Function: `zapInNative(SingleTokenSwap swapData0, SingleTokenSwap swapData1, IZapper.KodiakVaultStakingParams stakingParams, IZapper.VaultDepositParams vaultParams)`

This function zaps into the vault using native tokens to swap with token0 and token1.

Inputs

- `swapData0`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: N/A.
  - **Impact**: Information of swapping native token to token0.
- `swapData1`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: N/A.
  - **Impact**: Information of swapping native token to token1.
- `stakingParams`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: `stakingParams.kodiakVault` must be whitelisted.
  - **Impact**: Information of staking in the vault.
- `vaultParams`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: N/A.
  - **Impact**: Information of depositing into the vault.

Branches and code coverage

Intended branches

- Check if `stakingParams.kodiakVault` is added in the whitelist.
  - ☑ Test coverage

Negative behavior

- `stakingParams.kodiakVault` is not in the whitelist.
  - ☑ Negative test

Function: `zapInToken0(SingleTokenSwap swapData, KodiakVaultStakingParams stakingParams, VaultDepositParams vaultParams)`

This zaps into the vault using token0 — swaps token0 for token1.

Inputs

- `swapData`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: N/A.
  - **Impact**: Information of swapping token0 to token1.
- `stakingParams`

- **Control**: Fully controlled by the caller.
- **Constraints**: `stakingParams.kodiakVault` must be whitelisted.
- **Impact**: Information of staking in the vault.

- `vaultParams`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: N/A.
  - **Impact**: Information of depositing into the vault.

Branches and code coverage

Intended branches

- Check if `stakingParams.kodiakVault` is added in the whitelist.
  - ☑ Test coverage

Negative behavior

- `stakingParams.kodiakVault` is not in the whitelist.
  - ☑ Negative test

Function: `zapInToken1(SingleTokenSwap swapData, KodiakVaultStakingParams stakingParams, VaultDepositParams vaultParams)`

This zaps into the vault using token1 — swaps token1 for token0.

Inputs

- `swapData`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: N/A.
  - **Impact**: Information of swapping token1 to token0.
- `stakingParams`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: `stakingParams.kodiakVault` must be whitelisted.
  - **Impact**: Information of staking in the vault.
- `vaultParams`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: N/A.
  - **Impact**: Information of depositing into the vault.

Branches and code coverage

Intended branches

- Check if `stakingParams.kodiakVault` is added in the whitelist.
  - ☑ Test coverage

Negative behavior

- `stakingParams.kodiakVault` is not in the whitelist.
  - ☑ Negative test

Function: `zapInWithMultipleTokens(MultiSwapParams swapParams, KodiakVaultStakingParams stakingParams, VaultDepositParams vaultParams)`

This zaps into the vault using multiple tokens that swap for token0 and token1.

Inputs

- `swapParams`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: N/A.
  - **Impact**: Information of `swapParams` that requires to swap multiple tokens.
- `stakingParams`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: `stakingParams.kodiakVault` must be whitelisted.
  - **Impact**: Information of staking in the vault.
- `vaultParams`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: N/A.
  - **Impact**: Information of depositing into the vault.

Branches and code coverage

Intended branches

- Check if `stakingParams.kodiakVault` is added in the whitelist.
  - ☑ Test coverage

Negative behavior

- `stakingParams.kodiakVault` is not in the whitelist.
  - ☑ Negative test

Function: `zapInWithoutSwap(KodiakVaultStakingParams stakingParams, VaultDepositParams vaultParams)`

This zaps into the vault using token0 and token1 from the caller without swapping.

Inputs

- `stakingParams`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: `stakingParams.kodiakVault` must be whitelisted.
  - **Impact**: Information of staking in the vault.
- `vaultParams`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: N/A.
  - **Impact**: Information of depositing into the vault.

Branches and code coverage

Intended branches

- Check if `stakingParams.kodiakVault` is added in the whitelist.
  - ☑ Test coverage
- Receive token0 and token1 from the caller.
  - ☑ Test coverage

Negative behavior

- `stakingParams.kodiakVault` is not in the whitelist.
  - ☐ Negative test

Function: `zapIn(address inputToken, SingleTokenSwap swapToToken0, SingleTokenSwap swapToToken1, KodiakVaultStakingParams stakingParams, VaultDepositParams vaultParams)`

This zaps into the vault using a whitelisted token that swaps for token0 and token1.

Inputs

- `inputToken`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: N/A.
  - **Impact**: Whitelisted token that swaps for token0 and token1.
- `swapToToken0`

- **Control**: Fully controlled by the caller.
- **Constraints**: N/A.
- **Impact**: Information of swapping inputToken to token0.

- `swapToToken1`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: N/A.
  - **Impact**: Information of swapping inputToken to token1.
- `stakingParams`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: `stakingParams.kodiakVault` must be whitelist.
  - **Impact**: Information of staking in the vault.
- `vaultParams`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: N/A.
  - **Impact**: Information of depositing into the vault.

Branches and code coverage

Intended branches

- Check if `stakingParams.kodiakVault` is added in the whitelist.
  - ☑ Test coverage
- Receive amount that requires to swap inputToken.
  - ☑ Test coverage

Negative behavior

- `stakingParams.kodiakVault` is not in the whitelist.
  - ☑ Negative test

Function: `zapOutNative(address receiver, SingleTokenSwap swapData0, SingleTokenSwap swapData1, IZapper.KodiakVaultUnstakingParams unstakeParams, IZapper.VaultRedeemParams redeemParams)`

This zaps out to get native token.

Inputs

- `receiver`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: N/A.
  - **Impact**: Address to receive output native token.
- `swapData0`

    - **Control**: Fully controlled by the caller.
    - **Constraints**: N/A.
    - **Impact**: Information of swapping token0 to native token.
- `swapData1`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: N/A.
    - **Impact**: Information of swapping token1 to native token.
- `unstakeParams`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: N/A.
    - **Impact**: Information of unstaking from the Kodiak Vault.
- `redeemParams`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: N/A.
    - **Impact**: Information of redeeming from the vault.

Branches and code coverage

Intended branches

- Check if the amount of token0 received by Zapper is not zero.
    - ☑ Test coverage
- Check if the amount of token1 received by Zapper is not zero.
    - ☑ Test coverage
- Swap token0 and token1 to native token.
    - ☑ Test coverage
- Transfer native token to the receiver.
    - ☑ Test coverage

Function: `zapOutToToken0(address receiver, SingleTokenSwap swapData, KodiakVaultUnstakingParams unstakeParams, VaultRedeemParams redeemParams)`

This zaps out to get token0.

Inputs

- `receiver`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: N/A.
    - **Impact**: Address to receive token00.
- `swapData`

    - **Control**: Fully controlled by the caller.
    - **Constraints**: N/A.
    - **Impact**: Information of swap token1 to token0.
- `unstakeParams`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: N/A.
    - **Impact**: Information of unstaking from the Kodiak Vault.
- `redeemParams`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: N/A.
    - **Impact**: Information of redeeming from the vault.

Branches and code coverage

Intended branches

- Check if the amount of token0 received by Zapper is not zero.
    - ☑ Test coverage
- Check if the amount of token1 received by Zapper is not zero.
    - ☑ Test coverage
- Transfer token0 to the receiver.
    - ☑ Test coverage
- Transfer remaining token1 to the caller.
    - ☑ Test coverage

Function: `zapOutToToken1(address receiver, SingleTokenSwap swapData, KodiakVaultUnstakingParams unstakeParams, VaultRedeemParams redeemParams)`

This zaps out to get token1.

Inputs

- `receiver`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: N/A.
    - **Impact**: Address to receive token1.
- `swapData`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: N/A.
    - **Impact**: Information of swapping token0 to token1.
- `unstakeParams`

- **Control**: Fully controlled by the caller.
- **Constraints**: N/A.
- **Impact**: Information of unstaking from the Kodiak Vault.

- `redeemParams`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: N/A.
  - **Impact**: Information of redeeming from the vault.

Branches and code coverage

Intended branches

- Check if the amount of token0 received by Zapper is not zero.
  - ☑ Test coverage
- Check if the amount of token1 received by Zapper is not zero.
  - ☑ Test coverage
- Transfer token1 to the receiver.
  - ☑ Test coverage
- Transfer remaining token0 to the caller.
  - ☑ Test coverage

Function: `zapOut(address outputToken, address receiver, SingleTokenSwap swap0, SingleTokenSwap swap1, KodiakVaultUnstakingParams unstakeParams, VaultRedeemParams redeemParams)`

This zaps out of a vault using a token0 or token1 swap to get outputToken.

Inputs

- `outputToken`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: N/A.
  - **Impact**: Token to be received after zap-out.
- `receiver`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: N/A.
  - **Impact**: Address to receive outputToken.
- `swap0`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: N/A.
  - **Impact**: Information of swapping token0 to outputToken.
- `swap1`

    - **Control**: Fully controlled by the caller.
    - **Constraints**: N/A.
    - **Impact**: Information of swapping token1 to outputToken.
- `unstakeParams`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: N/A.
    - **Impact**: Information of unstaking from the Kodiak Vault.
- `redeemParams`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: N/A.
    - **Impact**: Information of redeeming from the vault.

Branches and code coverage

Intended branches

- Check if the amount of token0 received by Zapper is not zero.
    - ☑ Test coverage
- Check if the amount of token1 received by Zapper is not zero.
    - ☑ Test coverage
- Swap token0 and token1 to outputToken.
    - ☑ Test coverage
- Transfer outputToken to receiver.
    - ☑ Test coverage

6. 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 eight findings. Three critical issues were found. One was of high impact, one was of medium impact, two were of low impact, and the remaining finding was informational in nature.

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.