Making Web3 Space Safe for Everyone

KALOS

Rodeo Finance

Security Assessment

Published on : 26 Jul. 2023
Version v2.1

Security Report Published by KALOS

v2.1 26 Jul. 2023

Auditor : Jade Han

Found issues

Severity of Issues	Findings	Resolved	Acknowledged	Comment
Critical	5	5	-	-
High	2	1	1	-
Medium	-	-	-	-
Low	-	-	-	-
Tips	1	1	-	-

TABLE OF CONTENTS

ABOUT US

Making Web3 Space Safer for Everyone

KALOS is a flagship service of HAECHI LABS, the leader of the global blockchain industry. We bring together the best Web2 and Web3 experts. Security Researchers with expertise in cryptography, leaders of the global best hacker team, and blockchain/smart contract experts are responsible for securing your Web3 service.

We have secured the most well-known web3 services including 1inch, SushiSwap, Klaytn, Badger DAO, SuperRare, Netmarble, Klaytn and Chainsafe. We have secured $60B crypto assets on over 400 main-nets, Defi protocols, NFT services, P2E, and Bridges.

KALOS is the only blockchain technology company selected for the Samsung Electronics Startup Incubation Program in recognition of our expertise. We have also received technology grants from the Ethereum Foundation and Ethereum Community Fund.

Inquiries: audit@kalos.xyz

Website: https://kalos.xyz

Executive Summary

Purpose of this report

This report was prepared to audit the security of the contracts developed by the Rodeo team. KALOS conducted the audit focusing on whether the system created by the Rodeo team is soundly implemented and designed as specified in the published materials, in addition to the safety and security of the contracts.

In detail, we have focused on the following.

- Project availability issues like Denial of Service.
- Storage variable access control.
- Function access control
- saved asset freeze
- saved asset theft
- Yield calculation manipulate
- Unhandled Exception

Codebase Submitted for the Audit

The code used in this audit can be found on GitHub (https://github.com/RodeoFi/rodeo)

The last commit of the code used for this Audit is `525cc3ef7a9fe709b25ae549544142332c4eb343`.

Audit Timeline

Date	Event
2023/06/19	Audit Initiation
2023/07/20	Delivery of v2.0 report.
2023/07/26	Delivery of v2.1 report

Findings

KALOS found 5 Critical, 2 High, 0 medium and 0 Low severity issues. There are 1 Tips issues explained that would improve the code's usability or efficiency upon modification.

Severity	Issue	Status
Critical	TraderJoe Internal TWAP Flaw Issue	(Resolved - v2.0)
Critical	Predictability of Earnings Function Enables Sandwich Trading	(Acknowledged - v2.1)
Critical	OracleTWAP Manipulation	(Resolved - v2.0)
TIPS	Inability to Change UniswapV3 TWAP Duration Issue	(Resolved - v2.0)
Critical	OracleBalancer5050ETH Manipulation Issue	(Resolved - v2.0)
TIPS	Chainlink Oracle Wrong Usage Issue	(Resolved - v2.1)
Critical	Lack of Verification in WithdrawLP Function of LiquidityMining Contract	(Resolved - v2.0)
High	Rate Manipulation Vulnerability in StrategyJoe	(Resolved - v2.0)

OVERVIEW

Protocol overview

Rodeo is a leveraged farming protocol initially deployed to Arbitrum.
The rodeo includes lending pools, liquidations, vetted strategy adapters, auxiliary tools and Liquidity Mining Contracts.

LiquidityMining

The LiquidityMining contract implements a liquidity mining program, incentivizing users to provide liquidity to a pool of tokens. SushiSwap's MiniChef inspires it. The contract allows users to deposit tokens into different pools, and in return, they receive rewards in the form of another token. The rewards are distributed based on the user's share of the total liquidity in each pool.
The contract supports multiple pools, each with its allocation point determining the proportion of rewards it receives. Users can deposit and withdraw tokens, and they have the option to lock their deposits for a specified duration to receive additional rewards. An early withdrawal fee is imposed on users who withdraw their funds before the lock period ends.
The contract tracks user balances and rewards using a UserInfo struct. It calculates the amount of boost applied to each user's deposit based on their lock duration and LP token balance. The contract also handles reward calculations and distribution based on the time that has passed since the last reward update.
Overall, the LiquidityMining contract allows users to earn rewards by providing liquidity to different pools, encouraging participation, and incentivizing long-term deposits.

PartnerProxy

It is a contract that allows certain addresses to execute function calls on behalf of the contract owner.
The contract has a "setExec" function that allows the owner to grant or revoke execution permission to specific addresses.
A "call" function enables the execution of arbitrary calls to external contracts specified by the "tar" address parameter. It also allows the caller to send a value along with the call.
The contract has a "pull" function that allows the contract owner to transfer any ERC20 tokens held by the contract to specific address.
Additionally, an "approve" function allows the contract owner to approve the transfer of any ERC20 tokens held by the contract to another address.

The contract uses a utility library named "Util" and imports an interface for ERC20 tokens. It also emits a custom error called "CallReverted" which is used if a call to an external contract fails.

OracleBalancer5050ETH

The `OracleBalancer5050ETH` contract is a simple oracle that calculates the price of a token in terms of ETH. It is designed to be used with a Balancer pool that contains two tokens with a 50/50 weight distribution. This oracle relies on another oracle (`ethOracle`) to get the ETH price, and it uses the token balances from the Balancer vault to calculate the token price.
The contract constructor initializes the contract by setting the Balancer vault, pool, ETH oracle, and indices of the tokens in the pool. The `decimals` function returns the number of decimal places for the token price. The `latestAnswer` function calculates the current token price by dividing the ETH price by the token price and returns it as an `int256` value.

OracleTWAP

The `OracleTWAP` contract in the provided code implements a Time-Weighted Average Price (TWAP) oracle. It retrieves price data from an external Oracle contract and calculates the average price over a specific time interval.
The contract allows multiple authenticated addresses to update the prices and control the execution of certain functions. The `currentPrice` function retrieves the latest price from the oracle and adjusts it based on the decimal precision.
The `update` function can only be executed by an authenticated address, and it updates the prices array by storing the current price at the appropriate index. It also updates the last timestamp.
The `latestAnswer` function calculates and returns the average price based on the stored prices array. It checks if the price is stale, ensuring it's within the acceptable time range.
The contract emits events for price updates and file changes.

OracleUniswapV2Eth

The provided code is a smart contract called "OracleUniswapV2Eth" designed to provide price data for tokens on the Uniswap V2 decentralized exchange with ETH. Upon deployment, the contract requires the addresses of the Uniswap V2 pair, the WETH token, and an external ETH oracle.
The main functionality of this contract is the "latestAnswer" function, which calculates the latest price of the token in terms of ETH. It achieves this by obtaining the token reserves from the Uniswap V2 pair, adjusting the decimal values of the reserves, and then calculating the price using the external ETH oracle.
Additionally, the contract stores important information, such as the addresses of the tokens being traded, their respective decimal values, and the contract's own default decimals value.

In summary, the "OracleUniswapV2Eth" contract serves as an oracle, providing real-time price data for tokens on the Ethereum blockchain using the Uniswap V2 decentralized exchange and an external ETH oracle. It enables users to fetch accurate token prices relative to ETH, facilitating various decentralized finance (DeFi) applications and financial services.

OracleUniswapV2Usdc

The provided code is a smart contract called "OracleUniswapV2Usdc" that acts as an oracle for obtaining the price of a token relative to USDC on the Uniswap V2 decentralized exchange.
The contract initializes with the addresses of a Uniswap V2 pair and USDC token. It then defines the addresses of token0 and token1 of the pair. The contract implements a "decimals" function that returns the number of decimals for the price set to 18. The "latestAnswer" function retrieves the latest price by fetching the reserves of the pair and performing calculations based on the token0 and token1 values. The result is returned as an int256.
Overall, this contract provides a straightforward way to calculate the price of a token using USDC with Uniswap V2.

Multisig

The Multisig contract is smart, allowing multiple owners to manage and execute transactions collectively. It implements a threshold mechanism requiring a certain number of owner confirmations before a transaction can be executed. The contract also provides functionality to add or remove owners, set proposers and executors, and set transaction delay.
The contract keeps track of transaction details such as time, target address, value, data, and execution status. Owners can confirm transactions and either execute or cancel them. The contract calls the target address with the specified value and data if a transaction is executed. If it fails, the transaction is reverted.
Overall, the Multisig contract enables secure and decentralized management of transactions by multiple owners, providing transparency and accountability in the execution process.

StrategyCamelot

The `StrategyCamelot` contract is a strategy contract that interacts with the CamelotNFTPool. It implements the `Strategy` interface and inherits from it. The main purpose of this contract is to manage assets and perform various operations such as minting, burning, earning, staking, and unstaking.
The contract initializes by receiving the addresses of the StrategyHelper, CamelotNFTPoolFactory, and Pool contracts. It retrieves the necessary token addresses from the factory and stores them. It also sets the strategy's name based on the tokens' symbols in the pool.

Overall, the `StrategyCamelot` contract provides a function for managing assets in the CamelotNFTPool and performing various actions based on the strategy's logic.

StrategyJoe

The "StrategyJoe" contract is a strategy contract designed to interact with TraderJoe v2.1. It implements the Strategy interface and inherits from it. The primary objective of this contract is to optimize token exchange rates and enable LP token management through minting and burning. It utilizes functions for earning, exiting, and rebalancing to ensure efficient asset management within the protocol.

Upon deployment, the contract requires the addresses of the StrategyHelper, IJoeLBRouter, and IJoeLBPair contracts. It utilizes these addresses to interact with the TraderJoe protocol and execute its strategy effectively. Additionally, the contract sets a name for the strategy based on the symbols of the tokens involved in the trading pair.

In summary, the "StrategyJoe" contract acts as a derivative strategy for TraderJoe v2.1, focusing on optimizing token exchange rates and providing mechanisms for minting and burning LP tokens. Its functionalities revolve around efficient asset management within the protocol based on its defined strategy.

StrategyPlutusPlvGlp

The "StrategyPlutusPlvGlp" contract is a strategy contract designed to interact with the PlutusDAO platform and manage assets within the PlutusDAO plvGLP pool. It implements the Strategy interface and inherits from it. The primary objective of this contract is to optimize the token exchange rates and perform various operations such as minting, burning, earning, staking, and unstaking within the PlutusDAO ecosystem.

Upon deployment, the contract receives addresses of various PlutusDAO contracts, including PartnerProxy, IRewardRouter, IGlpManager, IPlutusDepositor, and IPlutusFarm, as well as the address of the USDC token. It utilizes these addresses to interact with the PlutusDAO ecosystem and execute its strategy effectively.

The StrategyPlutusPlvGlp contract utilizes the PlutusDAO ecosystem to optimize asset management and maximize returns. It employs functions for earning yield, depositing assets, and exiting strategies. Additionally, the contract allows setting the exit fee to manage user withdrawals effectively.

In summary, the "StrategyPlutusPlvGlp" contract acts as a derivative strategy for the PlutusDAO plvGLP pool, focusing on optimizing token exchange rates and providing mechanisms for minting and burning LP tokens within the PlutusDAO ecosystem. Its functionalities revolve around efficient asset management and maximizing returns for users in the plvGLP pool.

Scope

```
├── LiquidityMining.sol
├── PartnerProxy.sol
├── oracles
│   ├── OracleBalancer5050ETH.sol
│   ├── OracleTWAP.sol
│   ├── OracleUniswapV2Eth.sol
│   ├── OracleUniswapV2Usdc.sol
├── support
│   ├── Multisig.sol
├── strategies
│   ├── StrategyCamelot.sol
│   ├── StrategyJoe.sol
│   ├── StrategyPlutusPlvGlp.sol
```

We have confirmed that the code deployed on-chain on the audit's completion was the updated code, which included the recommended patches.

Below is the list of verified contract addresses.

LiquidityMining
https://arbiscan.io/address/0xEc7a0f4F39f6f244e7668A98DAaCcBE01E4D63E7

OracleChainlink USDC
https://arbiscan.io/address/0x8E9b3C8fF86A5A2E592F4e32948A9555b8F0cB35
OracleChainlink ETH
https://arbiscan.io/address/0xC3Fa00136EFa7B7c39F6B6f4Ed4fb8de18480712
OracleChainlink wstETH (not USDC)
https://arbiscan.io/address/0x44AD7653c58755F705B297790402D0A5545f7e23
OracleChainlinkETH wstETH
https://arbiscan.io/address/0xC75B29Cfd5244FBf55c5567FbF20a3C2D83c8A80
OracleChainlink rETH (not USDC)
https://arbiscan.io/address/0xE2c1E669a40cf929B8fEE751933E995E9A0C02C2
OracleChainlinkETH rETH
https://arbiscan.io/address/0x09E152B25316e574579bEcC3C760D3730A8B6AC3
OracleChainlink USDT
https://arbiscan.io/address/0xB8B8e4028A8e3DF9657FFCF3F7D16A0C080ea96c

OracleChainlink ARB
https://arbiscan.io/address/0x3AC1bfc26e8c3FcF55e8E41DBb7910b38Da62eBD
OracleChainlink JOE
https://arbiscan.io/address/0x8B2F77e493C79dc297b18F13Ed77A09950603b77
OracleChainlink SUSHI
https://arbiscan.io/address/0x48919E7B6a2595A319886223f4c909f992261604
OracleUniswapV2Pair Camelot RDO/ETH
https://arbiscan.io/address/0xe3bf4b2cf2b94FfE36F4887C684ea8F588c9Ef04

StrategyPlutusPlvGlp PartnerProxy
https://arbiscan.io/address/0xfF1249c81e6614796381e0b9c88a3D080dAD01dF
StrategyPlutusPlvGlp
https://arbiscan.io/address/0x4c38dAEc4059151e0beb57AaD2aF0178273FFf28
StrategyJonesUsdc PartnerProxy
https://arbiscan.io/address/0x5859731D7b7e413A958eA1cDb9020C611b016395

Access Controls

The following risks exist regarding permissions.

- *Authorized actors can set each contract's environment variables (interest rate, coin swap path, slippage, new strategy registration, etc.) so that the assets deposited in the pool/position suffer loss. This may result in the loss of assets of users who have deposited assets in the rodeo.*
- *In the Audit Scope, each Strategy Contract, LiquidityMining, and PartnerProxy allows approved actors to withdraw all assets from the contract.*

The access controls in the contracts are verified using the auth modifier. Authorization in access control is divided into two types: authorized and unauthorized. The authorized privilege is recorded in the exec hash table in address=>bool format.
There are various functions implemented for authorized actors. It is possible to set environment variables that need service operation for each contract.

- LiquidityMining.sol
  In the LiquidityMining contract, authorized users can update critical variables related to the operation of the LiquidityMining contract. They can also add new pools or modify the allocation points, influencing users' interest rewards. Additionally, the "removeUser" function allows these authorized users to remove specific users from the contract forcibly.
- Strategy{N}.sol
  Authorized specific users can configure the paused state, default slippage, and occasionally a few other variables as needed in the Strategy.
- PartnerProxy.sol
  PartnerProxy is a substitute for StrategyPlutusPlvGlp, facilitating the exchange of assets with GLP-related contracts and Plutus DAO-related contracts. Authorized users of this contract can use PartnerProxy to access assets generated through StrategyPlutusPlvGlp directly.

Unlike the previously mentioned contracts, this contract has a separate set of Owners who can designate transactions to be executed by a Multisig wallet. When a transaction is proposed, most Owners must approve it for execution. Apart from Owners, there are also Proposers and Executors. Proposers have the authority to suggest transactions for execution, while Executors can execute proposed transactions once they receive approval from the majority of Owners. Owners have the ultimate power to run any action within the contract.

- Multisig.sol
  Multisig contract holds the authority to modify crucial variables present in almost all contracts it operates. Now, there are three Owners, and if at least two of them approve, Rodeo can exert its influence on other contracts through the Multisig contract.

The authorized accounts set in the contracts of the Rodeo are as follows. (As of 2023/07/26)

[Multisig] - 0xaB7d6293CE715F12879B9fa7CBaBbFCE3BAc0A5a
Owner: (EOA) 0x7D43BA375f060719f636bE1D3CF53Fed1c97abB9
Owner: (EOA) 0x5d52C98D4B2B9725D9a1ea3CcAf44871a34EFB96
Owner: (EOA) 0xa6Ae21Df96fD8e4337f89861904AEfADD3964E79
* Currently, a transaction can be executed if two owners agree.

Proposer: (EOA, ContractCreator) 0x20dE070F1887f82fcE2bdCf5D6d9874091e6FAe9

[LiquidityMining] - 0xEc7a0f4F39f6f244e7668A98DAaCcBE01E4D63E7
exec: (Contract, Multisig) 0xaB7d6293CE715F12879B9fa7CBaBbFCE3BAc0A5a

[StrategyPlutusPlvGlp PartnerProxy] - 0xfF1249c81e6614796381e0b9c88a3D080dAD01dF
exec: (Contract, Multisig) 0xaB7d6293CE715F12879B9fa7CBaBbFCE3BAc0A5a
exec: (Contract, StrategyPlutusPlvGlp) 0x7174aD2d9C836B845ba5611dc5b90740707060eA

[StrategyJonesUsdc PartnerProxy] - 0x5859731D7b7e413A958eA1cDb9020C611b016395
exec: (Contract, Multisig) 0xaB7d6293CE715F12879B9fa7CBaBbFCE3BAc0A5a
exec: (Contract, StrategyJonesUsdc) 0x6A77FEC52D8575AC70F5196F833bd4A6c86c81AE

[StrategyCamelot ETH/USDC] - 0x91308b8d5e2352C7953D88A55D1012D68bF1EfD0
exec: (Contract, InvestorActorStrategyProxy) 0x023be37efd018ce6b2707ea7452012642b6a5000
exec: (Contract, Multisig) 0xab7d6293ce715f12879b9fa7cbabbfce3bac0a5a
exec: (Contract, Investor) 0x8accf43dd31dfcd4919cc7d65912a475bfa60369

[StrategyCamelot ARB/ETH] - 0xA36bBd6494Fd59091898DA15FFa69a35bF0676b0
exec: (Contract, Investor) 0x8accf43dd31dfcd4919cc7d65912a475bfa60369
exec: (Contract, Multisig) 0xab7d6293ce715f12879b9fa7cbabbfce3bac0a5a
exec: (Contract, InvestorActorStrategyProxy) 0x023be37efd018ce6b2707ea7452012642b6a5000

[StrategyJoe ETH/USDC] - 0xDef79690aD84c943525DbA14EB8cb6E82Ae79935
exec: (Contract, Investor) 0x8accf43dd31dfcd4919cc7d65912a475bfa60369
exec: (Contract, Multisig) 0xab7d6293ce715f12879b9fa7cbabbfce3bac0a5a
exec: (Contract, InvestorActorStrategyProxy) 0x023be37efd018ce6b2707ea7452012642b6a5000

[StrategyJoe ARB/ETH] - 0x7012aA8bc7362628Ce560cc42C4705c33D990FA8
exec: (Contract, Investor) 0x8accf43dd31dfcd4919cc7d65912a475bfa60369
exec: (Contract, Multisig) 0xab7d6293ce715f12879b9fa7cbabbfce3bac0a5a
exec: (Contract, InvestorActorStrategyProxy) 0x023be37efd018ce6b2707ea7452012642b6a5000

[StrategyJoe JOE/ETH] - 0x9fcDf795eBaF90197C0B581ce56416050714f545
exec: (Contract, Investor) 0x8accf43dd31dfcd4919cc7d65912a475bfa60369
exec: (Contract, Multisig) 0xab7d6293ce715f12879b9fa7cbabbfce3bac0a5a
exec: (Contract, InvestorActorStrategyProxy) 0x023be37efd018ce6b2707ea7452012642b6a5000

[StrategyPlutusPlvGlp] - 0x4c38dAEc4059151e0beb57AaD2aF0178273FFf28
exec: (Contract, Investor) 0x8accf43dd31dfcd4919cc7d65912a475bfa60369
exec: (Contract, Multisig) 0xab7d6293ce715f12879b9fa7cbabbfce3bac0a5a
exec: (Contract, InvestorActorStrategyProxy) 0x023be37efd018ce6b2707ea7452012642b6a5000

auth() :

- Strategy.sol - StrategyJoe, StrategyCamelot, StrategyPlutusPlvGlp#file(bytes32, uint256)
- Strategy.sol - StrategyJoe, StrategyCamelot, StrategyPlutusPlvGlp#file(bytes32, address)
- Strategy.sol - StrategyJoe, StrategyCamelot, StrategyPlutusPlvGlp#mint(address, uint256, bytes calldata)
- Strategy.sol - StrategyJoe, StrategyCamelot, StrategyPlutusPlvGlp#burn(address, uint256, bytes calldata)
- Strategy.sol - StrategyJoe, StrategyCamelot, StrategyPlutusPlvGlp#exit(address)
- Strategy.sol - StrategyJoe, StrategyCamelot, StrategyPlutusPlvGlp#move(address)
- PartnerProxy.sol - PartnerProxy#call(address, uint256, bytes calldata)
- PartnerProxy.sol - PartnerProxy#setExec(address, bool)
- PartnerProxy.sol - PartnerProxy#pull(address)
- PartnerProxy.sol - PartnerProxy#approve(address, address)
- LiquidityMining.sol - LiquidityMining#file(bytes32, uint256)
- LiquidityMining.sol - LiquidityMining#file(bytes32, address)
- LiquidityMining.sol - LiquidityMining#poolAdd(uint256, address)
- LiquidityMining.sol - LiquidityMining#poolSet(uint256, uint256)
- LiquidityMining.sol - LiquidityMining#removeUser(uint256, address, address)

isOwner() :

- Multisig.sol - Multisig#addOwner(address, uint256)
- Multisig.sol - Multisig#removeOwner(address, uint256)

- Multisig.sol - Multisig#setProposer(address, bool)
- Multisig.sol - Multisig#setExecutor(address, bool)
- Multisig.sol - Multisig#add(address, uint256)
- Multisig.sol - Multisig#confirm(uint256, bool)
- Multisig.sol - Multisig#cancel(uint256)
- Multisig.sol - Multisig#execute(uint256)

isProposer() :

- Multisig.sol - Multisig#add(address, uint256)

isExecutor() :

- Multisig.sol - Multisig#execute(uint256)

FINDINGS

1. TraderJoe Internal TWAP Flaw Issue

ID: Rodeo-2-1
Severity: Critical
Type: Logic Error
Difficulty: Low
File: contracts/src/strategies/StrategyJoe.sol

Issue

The following code snippet is part of the OracleTraderJoe code.

```
// OracleTraderJoe.sol
function latestAnswer() external view returns (int256) {
    address tok = pair.getTokenX();
    if (tok == weth) {
        tok = pair.getTokenY();
    }
    (uint256 cumId1,,) = pair.getOracleSampleAt(uint40(block.timestamp));
    (uint256 cumId0,,) = pair.getOracleSampleAt(uint40(block.timestamp - twapPeriod));
    uint24 avgId = uint24(uint256(cumId1 - cumId0) / twapPeriod);
    uint256 price = FullMath.mulDiv(pair.getPriceFromId(avgId), 1e18, 1 << 128);
    if (pair.getTokenX() == weth) {
        price = 1e18 * 1e18 / price;
    }
    price = price * (10 ** IERC20(tok).decimals()) / 1e18;
    return int256(price) * ethOracle.latestAnswer() / int256(10 ** ethOracle.decimals());
}
```

https://github.com/rodeofi/rodeo/blob/525cc3ef7a9fe709b25ae549544142332c4eb343/contracts/src/oracles/OracleTraderJoe.sol

Looking at the code, TraderJoe utilizes the getOracleSampleAt function to fetch Twap-related elements. Let me explain how these elements are calculated.

Below is the code snippet from LBPair.sol for token swapping.

```
// LBPair.sol
/**
     * @notice Swap tokens iterating over the bins until the entire amount is swapped.
     * Token X will be swapped for token Y if `swapForY` is true, and token Y for token X if `swapForY` is false.
     * This function will not transfer the tokens from the caller, it is expected that the tokens have already been
     * transferred to this contract through another contract, most likely the router.
     * That is why this function shouldn't be called directly, but only through one of the swap functions of a router
```

```
     * that will also perform safety checks, such as minimum amounts and slippage.
     * The variable fee is updated throughout the swap, it increases with the number of bins
crossed.
     * The oracle is updated at the end of the swap.
     * @param swapForY Whether you're swapping token X for token Y (true) or token Y for
token X (false)
     * @param to The address to send the tokens to
     * @return amountsOut The encoded amounts of token X and token Y sent to `to`
     */
function swap(bool swapForY, address to) external override nonReentrant returns (bytes32
amountsOut) {
    ...
    while (true) {
        // token swap
        ...
    }

    if (amountsOut == 0) revert LBPair__InsufficientAmountOut();

    _reserves = reserves.sub(amountsOut);
    _protocolFees = protocolFees;

    parameters = _oracle.update(parameters, activeId); // TWAP Oracle Update!
    _parameters = parameters.setActiveId(activeId);

    if (swapForY) {
        amountsOut.transferY(_tokenY(), to);
    } else {
        amountsOut.transferX(_tokenX(), to);
    }
}
```

https://github.com/traderjoe-xyz/joe-v2/blob/0e422c2d6f3f2095ac04c7e9efde9c9ebd57689b/src/LBPair.sol

Below is the code snippet from OracleHelper.sol that pertains to the Oracle update.

```
// OracleHelper.sol
function update(Oracle storage oracle, bytes32 parameters, uint24 activeId) internal returns
(bytes32) {
    uint16 oracleId = parameters.getOracleId();
    if (oracleId == 0) return parameters;

    bytes32 sample = getSample(oracle, oracleId);

    uint40 createdAt = sample.getSampleCreation();
    uint40 lastUpdatedAt = createdAt + sample.getSampleLifetime();

    if (block.timestamp.safe40() > lastUpdatedAt) {
        unchecked {
            (uint64 cumulativeId, uint64 cumulativeVolatility, uint64 cumulativeBinCrossed)
= sample.sg(
                uint40(block.timestamp - lastUpdatedAt),
                activeId,
                parameters.getVolatilityAccumulator(),
                parameters.getDeltaId(activeId)
            );
```

```
            uint16 length = sample.getOracleLength();
            uint256 lifetime = block.timestamp - createdAt;

            if (lifetime > _MAX_SAMPLE_LIFETIME) {
                assembly {
                    oracleId := add(mod(oracleId, length), 1)
                }

                lifetime = 0;
                createdAt = uint40(block.timestamp);

                parameters = parameters.setOracleId(oracleId);
            }

            sample = SampleMath.encode(
                length, cumulativeId, cumulativeVolatility, cumulativeBinCrossed,
uint8(lifetime), createdAt
            );
        }

        setSample(oracle, oracleId, sample);
    }

    return parameters;
}
```

https://github.com/traderjoe-xyz/joe-v2/blob/0e422c2d6f3f2095ac04c7e9efde9c9ebd57689b/src/libraries/OracleHelper.sol

Let me explain the differences between Twap implementations in other protocols.
We will assume that Swap 1 occurs in Block 1 and Swap 2 occurs in Block 2.
In the case of Uniswap, Swap 1 is reflected in Twap at Block 2.

It incorporates the most recent swap in the previous block into Twap.
On the other hand, in Trader Joe, Swap 1 is reflected in Twap at Block 1.
It is immediately incorporated into Twap as soon as the swap occurs.

Another flaw is observed in the OracleHelper code, where once a swap is reflected in Twap, subsequent swaps until the next delay does not affect the Oracle.
Suppose a scenario where someone borrows a large amount of capital through a Flashloan in protocols like AAVE, performs swaps until the desired bin id and recovers the funds.
Only the swaps made until the desired bin id will be reflected in Twap, while the recovery process will not be reflected in Twap.

If an attacker manipulates the Oracle by utilizing Flashloans with each passing delay, they can manipulate the Oracle to desired values, even though it may take some time.

Therefore, in the case of a third-party project utilizing a specific LBPair as a Price Oracle, unintended damages can occur due to flaws in Trader Joe's code rather than the project's code.
While this doesn't directly cause any losses to TraderJoe's assets, it is a highly critical issue from the perspective of Rodeo Finance.

Recommendation

I have concluded that TWAP provided by TraderJoe has a critical flaw. When I informed the TraderJoe team about it, they acknowledged the issue. Still, they stated that it is not an immediate concern for TraderJoe itself and said we don't know when the flawed patch will be finished. Based on this response, I have concluded that using TWAP provided by TraderJoe should be avoided.

Currently, Rodeo Finance relies on the OracleTraderJoe contract to evaluate the value of the JOE Token. Chainlink provides Price Feed for this token, which is recommended to utilize.

Fix Comment

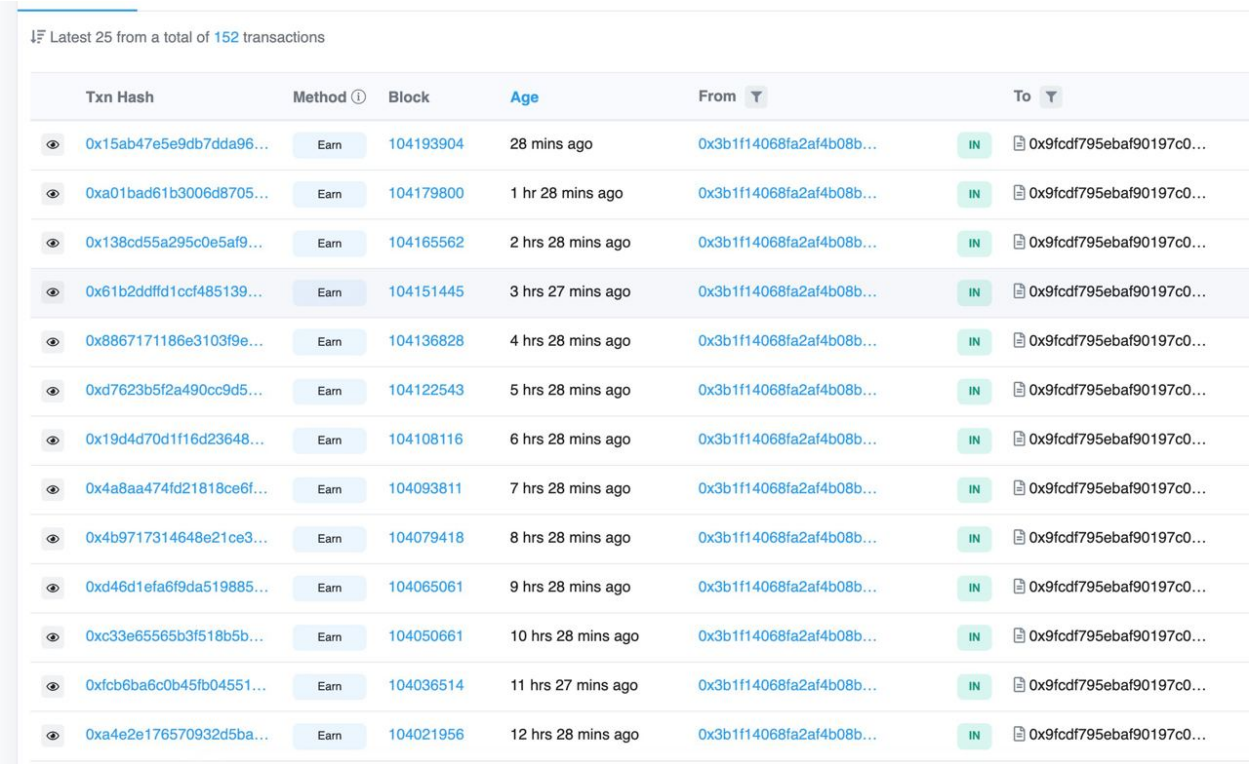
The Rodeo Team stated that they would refrain from using the Twap component provided by TraderJoe until the reported issues in TraderJoe are addressed. Instead, they will aim to utilize Chainlink Oracle to the maximum extent possible.

2. Predictability of Earnings Function Enables Sandwich Trading

ID: Rodeo-2-2
Severity: Critical
Type: Logic Error
Difficulty: Low
File: contracts/src/Strategy.sol

Issue

The Earn function of the already running Strategy Contract can only be called by specified users, and it has been invoked on an hourly basis.
By observing the transaction history on Arbiscan, anyone could ascertain the calling frequency.



Latest 25 from a total of 152 transactions

Txn Hash	Method	Block	Age	From		To
0x15ab47e5e9db7dda96...	Earn	104193904	28 mins ago	0x3b1f14068fa2af4b08b...	IN	0x9fcdf795ebaf90197c0...
0xa01bad61b3006d8705...	Earn	104179800	1 hr 28 mins ago	0x3b1f14068fa2af4b08b...	IN	0x9fcdf795ebaf90197c0...
0x138cd55a295c0e5af9...	Earn	104165562	2 hrs 28 mins ago	0x3b1f14068fa2af4b08b...	IN	0x9fcdf795ebaf90197c0...
0x61b2ddffd1ccf485139...	Earn	104151445	3 hrs 27 mins ago	0x3b1f14068fa2af4b08b...	IN	0x9fcdf795ebaf90197c0...
0x8867171186e3103f9e...	Earn	104136828	4 hrs 28 mins ago	0x3b1f14068fa2af4b08b...	IN	0x9fcdf795ebaf90197c0...
0xd7623b5f2a490cc9d5...	Earn	104122543	5 hrs 28 mins ago	0x3b1f14068fa2af4b08b...	IN	0x9fcdf795ebaf90197c0...
0x19d4d70d1f16d23648...	Earn	104108116	6 hrs 28 mins ago	0x3b1f14068fa2af4b08b...	IN	0x9fcdf795ebaf90197c0...
0x4a8aa474fd21818ce6f...	Earn	104093811	7 hrs 28 mins ago	0x3b1f14068fa2af4b08b...	IN	0x9fcdf795ebaf90197c0...
0x4b9717314648e21ce3...	Earn	104079418	8 hrs 28 mins ago	0x3b1f14068fa2af4b08b...	IN	0x9fcdf795ebaf90197c0...
0xd46d1efa6f9da519885...	Earn	104065061	9 hrs 28 mins ago	0x3b1f14068fa2af4b08b...	IN	0x9fcdf795ebaf90197c0...
0xc33e65565b3f518b5b...	Earn	104050661	10 hrs 28 mins ago	0x3b1f14068fa2af4b08b...	IN	0x9fcdf795ebaf90197c0...
0xfcb6ba6c0b45fb04551...	Earn	104036514	11 hrs 27 mins ago	0x3b1f14068fa2af4b08b...	IN	0x9fcdf795ebaf90197c0...
0xa4e2e176570932d5ba...	Earn	104021956	12 hrs 28 mins ago	0x3b1f14068fa2af4b08b...	IN	0x9fcdf795ebaf90197c0...

We have checked for any potential side effects in on-chain behavior and, upon reviewing multiple strategies, identified a scenario where the assets owned by the StrategyTraderJoe Contract could be compromised.
Below is the code for the Earn function of the StrategyTraderJoe Contract that inherit Strategy contract.

```
function _earn() internal override {
    _burnLP(1e18);
    _rebalance(slippage);
    _mintLP(slippage);
}
```

https://github.com/RodeoFi/rodeo/blob/4e2e6d2ae576660d7c71795bdb69d08220888e1f/contracts/src/strategies/strategyJoe.sol

To briefly explain the code's behavior, it involves burning the LP Tokens owned by the StrategyJoe Contract, adjusting the ratio of two assets through swapping, and then minting LP Tokens again.

Regarding the LP Token minting, it retrieves the current active bin ID of TraderJoe and adds liquidity to that location. In this case, even on Arbitrum, where MEV may not exist, a Front Run Attack is still possible because the attacker knows when the earn function will be executed and the rebalancing will occur. The attacker initiates a swap transaction that moves the bin ID to an abnormal position before the transaction executes the earn function. Then, the transaction that executes the earn function adds liquidity to the bin ID that the attacker desired. Once the liquidity is added, the attacker can swap at a lower price than the original price, thereby gaining economic advantage.

In this issue, StrategyJoe was used as an example for explanation, but it could also impact other strategies that remove deposited liquidity and rebalance during the earn function.

Recommendation

It is recommended to set the execution frequency of the earn function randomly.

Fix Comment

The Rodeo team is currently implementing random earns, and it will be live in the week following the issuance of this report.

3. OracleTwap Manipulation

ID: Rodeo-2-3
Severity: Critical
Type: Logic Error
Difficulty: Low
File: contracts/src/oracles/OracleTWAP.sol

Issue

The critical point of this issue is that the code loses its resilience against temporary liquidity manipulation inherent in TWAP.

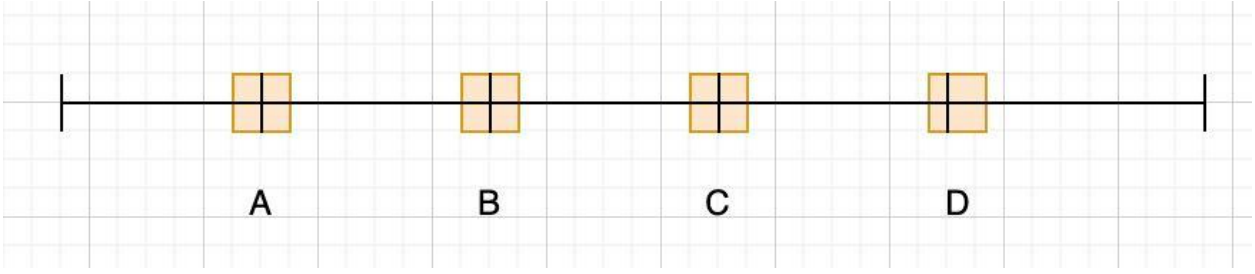
Below code is the OracleTwap code.

```
function latestAnswer() external view returns (int256) {
    require(block.timestamp < lastTimestamp + (updateInterval * 2), "stale price");
    int256 price = (prices[0] + prices[1] + prices[2] + prices[3]) / 4;
    return price;
}

function update() external auth {
    require(block.timestamp > lastTimestamp + updateInterval, "before next update");
    lastIndex = (lastIndex + 1) % 4;
    prices[lastIndex] = currentPrice();
    lastTimestamp = block.timestamp;
    emit Updated(prices[lastIndex]);
}

function currentPrice() public view returns (int256) {
    return oracle.latestAnswer() * 1e18 / int256(10 ** oracle.decimals());
}
```

https://github.com/rodeofi/rodeo/blob/525cc3ef7a9fe709b25ae549544142332c4eb343/contracts/src/oracles/OracleTWAP.sol

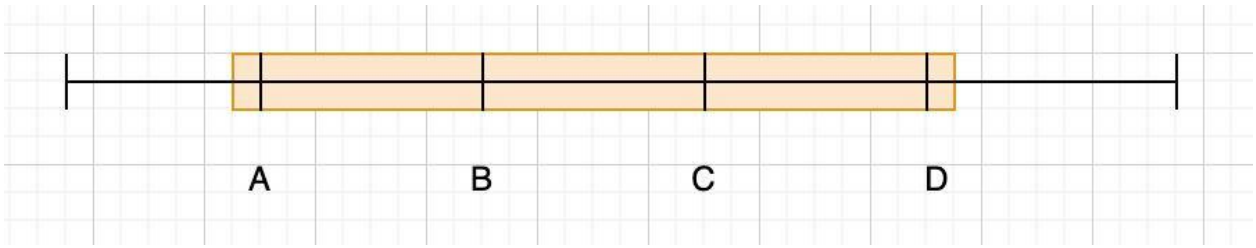
Let's assume the updateInterval is 5 minutes. When the update function is executed every 5 minutes, the Oracle fetches the value at that particular moment and calculates the average.

This can be visually represented as follows:



However, since the prices are retrieved only four times within a 5-minute interval, it does not truly represent a meaningful TWAP.

An accurate and robust TWAP should be represented as shown in the figure below:



Although the code was derived from a UniswapV2 fork, it was modified to remove TWAP-related elements from the pair contract code due to Camelot. However, this code needs to be more secure, and Rodeo should avoid handling pairs with low liquidity that require fetching prices from the Camelot Pair.

Recommendation

It is recommended to discontinue OracleTwap instead of utilizing oracles provided by decentralized exchanges (such as Uniswap V2, Uniswap V3, etc.) that offer Chainlink Price Feed or reliable TWAP without known vulnerabilities. This mitigates potential risks and ensures more secure price data for the system.

Fix Comment

The Rodeo Team acknowledged the flaw in OracleTwap and stated that they will retire the corresponding code.

4. Inability to Change UniswapV3 TWAP Duration Issue

ID: Rodeo-2-4
Severity: Tips
Type: N/A
Difficulty: -
File: contracts/src/oracles/OracleUniswapV3.sol

Issue

The following code is the OracleUniswapV3 code.

```
contract OracleUniswapV3 {
    address public pool;
    address public weth;
    IOracle public ethOracle;
    uint32 public constant twapPeriod = 1800;

    constructor(address _pool, address _weth, address _ethOracle) {
        pool = _pool;
        weth = _weth;
        ethOracle = IOracle(_ethOracle);
    }

    function decimals() external pure returns (uint8) {
        return 18;
    }

    function latestAnswer() external view returns (int256) {
        address tok = IUniswapV3Pool(pool).token0();
        if (tok == weth) {
            tok = IUniswapV3Pool(pool).token1();
        }
        (int24 amt,) = OracleLibrary.consult(pool, twapPeriod);
        uint256 pri = OracleLibrary.getQuoteAtTick(amt, uint128(10) **
IERC20(tok).decimals(), tok, weth);
        return int256(pri) * ethOracle.latestAnswer() / int256(10 ** ethOracle.decimals());
    }
}
```

https://github.com/rodeofi/rodeo/blob/525cc3ef7a9fe709b25ae549544142332c4eb343/contracts/src/oracles/OracleUniswapV3.sol

Upon reviewing the code, it can be observed that the code fetches the TWAP value for any given pair over a period of 1800 seconds (30 minutes), and this value is hardcoded as a constant. The inability to modify this value introduces inflexibility, as 1800 seconds may be suitable for some pairs but not for others.

It is important to allow for the flexibility to adjust this value based on the characteristics of each pair. While 1800 seconds may be appropriate for certain pairs, it may not be suitable for others with different liquidity dynamics or trading patterns.

Recommendation

It is recommended to modify the code to allow for changing the TWAP period.

Fix Comment

The Rodeo Team mentioned that they intentionally set the twap period as a constant to reduce the attack surface. If they need to change the Twap Period, they will create an entirely new Oracle Contract.

5. OracleBalancer5050ETH Manipulation Issue

ID: Rodeo-2-5 Severity: Critical

Type: Logic Error Difficulty: Low

File: contracts/src/oracles/OracleBalancer5050ETH.sol

Issue

The following code is related to OracleBalancer5050ETH.sol.

Upon reviewing the code, it is evident that it directly fetches the token balances from the Balancer Vault and uses them for token valuation.

However, this code is highly vulnerable to temporary manipulation caused by flash loans.

```
function latestAnswer() external view returns (int256) {
    int256 ethPrice = ethOracle.latestAnswer() * 1e18 / int256(10 ** ethOracle.decimals());
    (, uint256[] memory balances,) = vault.getPoolTokens(IBalancerPool(pool).getPoolId());
    int256 tokenPrice = int256(balances[tokenIndex] * 1e18 / balances[wethIndex]);
    return int256(ethPrice * 1e18 / tokenPrice);
}
```

https://github.com/rodeofi/rodeo/blob/525cc3ef7a9fe709b25ae549544142332c4eb343/contracts/src/oracles/OracleBalancer5050ETH.sol

Recommendation

We recommend to use chainlink oracle instead of this OracleBalancer5050ETH.

Fix Comment

The Rodeo Team removed the mentioned Oracle Code from the issue and said they would use Chainlink Oracle instead.

6. Chainlink Oracle Wrong Usage Issue

ID: Rodeo-2-6 Severity: High

Type: N/A Difficulty: Medium

File: contracts/src/oracles/OracleChainlinkETH.sol

Issue

The issue identified pertains to the usage of the OracleChainlinkETH contract.

It currently utilizes the latestAnswer function, which returns a straightforward price.

```
contract OracleChainlinkETH {
    IOracle public oracle;
    IOracle public ethOracle;

    constructor(address _oracle, address _ethOracle) {
        oracle = IOracle(_oracle);
        ethOracle = IOracle(_ethOracle);
    }

    function decimals() external pure returns (uint8) {
        return 18;
    }

    function latestAnswer() external view returns (int256) {
        int256 price = oracle.latestAnswer() * 1e18 / int256(10 ** oracle.decimals());
        return price * ethOracle.latestAnswer() / int256(10 ** ethOracle.decimals());
    }
}
```

https://github.com/rodeofi/rodeo/blob/525cc3ef7a9fe709b25ae549544142332c4eb343/contracts/src/oracles/OracleChainlinkETH.sol

Replacing the latestAnswer function with the latestRoundData is recommended to enhance the security of Chainlink usage.

When implementing the latestRoundData function, the following conditions should be checked:

Ensure that roundId, answer, and updatedAt values are not equal to 0.

Compare the updatedAt value with the current block.timestamp to verify that the data is relatively updated.

Recommendation

To mitigate potential risks and ensure accurate and up-to-date price data, the following recommendations are proposed:

1. Update the OracleChainlinkETH contract to utilize the latestRoundData function instead of the latestAnswer.
2. Implement the necessary checks to ensure roundId, answer, and updatedAt values are not 0.
3. Compare updatedAt with the current block.timestamp to provide the data within an acceptable time frame.

The Rodeo Team needs to consider even the worst-case scenarios while using Chainlink.

Fix Comment

The Rodeo Team has updated the code as recommended by the Kalos team, and the Kalos Team confirmed that the appropriate heartbeat is set in the deployed contract.

7. Lack of Validation in withdraw Function of LiquidityMining Contract

ID: Rodeo-2-7
Severity: Critical
Type: Arithmetic Overflow/Underflow
Difficulty: Low
File: contracts/src/LiquidityMining.sol

Issue

In the LiquidityMining Contract, no code is present in the withdraw function to verify if the amount of withdrawn tokens is equal to or less than the initially deposited amount.

```
function _withdraw(address usr, uint256 pid, uint256 amount, address to) internal {
   PoolInfo memory pool = poolUpdate(pid);
   UserInfo storage info = userInfo[pid][usr];
   require(block.timestamp >= info.lock, "locked");
   _userUpdate(pid, info, 0, 0, 0 - int256(amount));
   emit Withdraw(msg.sender, pid, amount, to);
}
...
function withdrawLp(uint256 pid, uint256 amount, address to) public loop live {
   UserInfo storage info = userInfo[pid][msg.sender];
   _userUpdate(pid, info, 0, 0 - int256(amount), 0);
   lpToken.transfer(to, amount);
   emit WithdrawLp(msg.sender, pid, amount, to);
}
...
function _userUpdate(uint256 pid, UserInfo storage info, uint256 lock, int256 lp, int256
amount) internal {
   int256 prev = int256(getBoostedAmount(info));
   if (amount != 0) info.amount = uint256(int256(info.amount) + amount);
   if (lock > 0) {
       require(info.lock == 0, "already locked");
       info.lock = block.timestamp + min(lock, boostMaxDuration);
       info.boostLock = boostMax * min(lock, boostMaxDuration) / boostMaxDuration;
   }
   if (info.lock > 0 && amount < 0) {
       info.lock = 0;
       info.boostLock = 0;
   }
   if (lp != 0) {
       uint256 value = strategyHelper.value(IPool(address(token[pid])).asset(),
info.amount);
       info.lp = uint256(int256(info.lp) + lp);
       uint256 lpValue = strategyHelper.value(address(lpToken), info.lp);
       if (lpValue > 0 && value > 0) {
           info.boostLp = (lpValue * 1e18 / value) > lpBoostThreshold ? lpBoostAmount : 0;
       } else if (info.boostLp > 0) {
           info.boostLp = 0;
```

```
        }
    }
    int256 next = int256(getBoostedAmount(info));
    info.rewardDebt += (next - prev) * int256(uint256(poolInfo[pid].accRewardPerShare)) /
1e12;
    poolInfo[pid].totalAmount = poolInfo[pid].totalAmount - uint256(prev) + uint256(next);
}
```

https://github.com/rodeofi/rodeo/blob/525cc3ef7a9fe709b25ae549544142332c4eb343/contracts/src/LiquidityMining.sol

This absence of validation poses potential risks, as it may lead to overflow issues or inadequate Token Balance within the contract, making the current code implementation unsafe.

Addressing this vulnerability and including appropriate checks to ensure a secure withdrawal process for deposited tokens is strongly recommended.

Recommendation

We recommend adding code to the withdraw function in the LiquidityMining Contract to validate if the amount of withdrawn tokens is equal to or less than the initially deposited amount.

Fix Comment

The recommended changes have been made. this can verify them at the following link

(https://github.com/rodeofi/rodeo/commit/191b9d2043f487def30bf2ded5cfc634a8310e0e)

8. Rate Manipulation Vulnerability in StrategyJoe

ID: Rodeo-2-8
Severity: High
Type: Logic Error
Difficulty: Medium
File: contracts/src/strategies/StrategyJoe.sol

Issue

StrategyJoe rate calculation code is below.

```
function _amounts() internal view returns (uint256, uint256, uint256[] memory) {
    uint256 num = bins.length;
    uint256[] memory amounts = new uint256[](num);
    uint256 amtX = 0;
    uint256 amtY = 0;
    for (uint256 i = 0; i < num; i++) {
        uint256 id = bins[i];
        uint256 amt = pair.balanceOf(address(this), id);
        amounts[i] = amt;
        (uint128 resX, uint128 resY) = pair.getBin(uint24(id));
        uint256 supply = pair.totalSupply(id);
        amtX += mulDiv(amt, uint256(resX), supply);
        amtY += mulDiv(amt, uint256(resY), supply);
    }
    return (amtX, amtY, amounts);
}
...
function _rate(uint256 sha) internal view override returns (uint256) {
    (uint256 amtX, uint256 amtY,) = _amounts();
    uint256 balX = tokenX.balanceOf(address(this));
    uint256 balY = tokenY.balanceOf(address(this));
    uint256 valX = strategyHelper.value(address(tokenX), amtX + balX);
    uint256 valY = strategyHelper.value(address(tokenY), amtY + balY);
    return sha * (valX + valY) / totalShares;
}
```

https://github.com/rodeofi/rodeo/blob/525cc3ef7a9fe709b25ae549544142332c4eb343/contracts/src/strategies/StrategyJoe.sol

During the security audit, a potential issue was found in the smart contract's _rate function, which calculates a rate based on the values obtained from the _amounts function.

By manipulating the values of amtX and amtY returned by the _amounts function, it is possible to affect the result of the _rate function.

Although the manipulation is limited to a maximum of 1-2% of the calculated rate, it can lower the liquidation threshold for users under specific circumstances.

Recommendation

As the first derivative for TraderJoe v2.1 has been created, no established Best Practices exist. We recommend suspending the operation of StrategyJoe until a clear Best Practice emerges.

Fix Comment

The Rodeo Team said they would suspend the operation of StrategyJoe until a clear solution is found.

DISCLAIMER

This report does not guarantee investment advice, the suitability of the business models, and codes that are secure without bugs. This report shall only be used to discuss known technical issues. Other than the issues described in this report, undiscovered issues may exist such as defects on the main network. In order to write secure codes, correction of discovered problems and sufficient testing thereof are required.

Appendix. A

Severity Level

CRITICAL	Must be addressed as a vulnerability that has the potential to seize or freeze substantial sums of money.
HIGH	Has to be fixed since it has the potential to deny users compensation or momentarily freeze assets.
MEDIUM	Vulnerabilities that could halt services, such as DoS and Out-of-Gas, need to be addressed.
LOW	Issues that do not comply with standards or return incorrect values
TIPS	Tips that makes the code more usable or efficient when modified

Difficulty Level

	Low	**Medium**	**High**
Privilege	anyone	Miner/Block Proposer	Admin/Owner
Capital needed	Small or none	Gas fee or volatile as price change	More than exploited amount
Probability	100%	Depend on environment	Hard as mining difficulty

Vulnerability Category

Arithmetic	▪ Integer under/overflow vulnerability ▪ floating point and rounding accuracy
Access & Privilege Control	▪ Manager functions for emergency handle ▪ Crucial function and data access ▪ Count of calling important task, contract state change, intentional task delay
Denial of Service	▪ Unexpected revert handling ▪ Gas limit excess due to unpredictable implementation
Miner Manipulation	▪ Dependency on the block number or timestamp. ▪ Frontrunning
Reentrancy	▪Proper use of Check-Effect-Interact pattern. ▪Prevention of state change after external call ▪ Error handling and logging.
Low-level Call	▪ Code injection using delegatecall ▪ Inappropriate use of assembly code
Off-standard	▪ Deviate from standards that can be an obstacle of interoperability.
Input Validation	▪ Lack of validation on inputs.
Logic Error/Bug	▪ Unintended execution leads to error.
Documentation	▪Coherency between the documented spec and implementation
Visibility	▪ Variable and function visibility setting
Incorrect Interface	▪ Contract interface is properly implemented on code.

End of Document