HAECHI AUDIT

Rodeo Finance

Smart Contract Security Analysis

Published on : 31 Jan. 2023

Version v1.1

HAECHI AUDIT

Smart Contract Audit Certificate

Rodeo Finance

Security Report Published by HAECHI AUDIT

v1.1 31 Jan. 2023

Auditor : Jade Han, Jinu Lee

Found issues

Severity of Issues	Findings	Resolved	Acknowledged	Comment
Critical	4	3	1	-
High	-	-	-	-
Medium	-	-	-	-
Low	-	-	-	-
Tips	3	1	2	-

TABLE OF CONTENTS

ABOUT US

The most reliable web3 security partner.

HAECHI AUDIT 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.

HAECHI AUDIT 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@haechi.io

Website: audit.haechi.io

Executive Summary

Purpose of this report

This report was prepared to audit the security of the contracts developed by the Rodeo team. HAECHI AUDIT 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 `4e2e6d2ae576660d7c71795bdb69d08220888e1f`.

Audit Timeline

Date	Event
2022/12/21	Audit Initiation
2023/01/20	Delivery of v1.0 report.
2023/01/31	Delivery of v1.1 report.

Findings

HAECHI AUDIT found 4 Critical, 0 High, 0 medium and 0 Low severity issues. There are 3 Tips issues explained that would improve the code's usability or efficiency upon modification.

Severity	Issue	Status
TIPS	Centralization Risk	(Found - v1.0)
Critical	earn function not check health factor	(Found - v1.0)
Critical	StrategySushiswap rate manipulation	(Found - v1.0)
Critical	StrategyUniswapV3 rate manipulation	(Found - v1.0)
Critical	First Strategy/Pool Depositor Front-Run	(Found - v1.0)
TIPS	function related with migration unimplemented	(Found - v1.0)
TIPS	Potential Reentrancy Risk	(Found - v1.0)

OVERVIEW

Protocol overview

Rodeo is a leveraged farming protocol initially deployed to Arbitrum. It has it's own lending pools, liquidations and vetted strategy adapters.

Contract Files

- Investor.sol
  Main contract allowing the borrowing of assets in Pools for use with Strategys. Liquidations also happen here.
- Pool.sol
  Lending pool contract allowing liquidity provider to lend assets in exchange for yield. Borrow and repay are only accessible to the Investor.sol contract. Deployed once per supported asset (and then whitelisted in the Investor.sol contract)
- Strategy{N}.sol
  Various strategies implement the logic for taking in any supported pool asset and investing it. mint() and burn() are only callable by the Investor.sol contract. mint() returns a share amount. burn() takes a share amount and returns an asset amount. rate() allows Investor.sol to evaluate the current value of a position and, in turn, it's health
- PositionManager.sol
  ERC721 NFT contract that wraps Investor.sol methods. Preferred way of interacting with the protocol for end users. The frontend uses this contract
- StrategyHelper.sol
  Contains a mapping of oracles and a mapping of swap routes for supported asset pairs. Used by strategies to easily value asset and swap assets while accounting for slippage
- InvestorHelper.sol
  Utility contract used by the frontend to batch view calls into one RPC request. Also used by the liquidation bot to batch check position's life() and batch kill() ignoring errors

lending pool

Currently, The lending pool supports USDC tokens. Rodeo's strategy adapter borrows USDC tokens from the pool and then swaps them with the necessary tokens.

The pool allows anyone to deposit USDC tokens. When a user deposits USDC tokens in the pool, the pool issues shares according to the current ratio. Users who have deposited tokens in the pool can withdraw the USDC at the current rate (deposited USDC and increased USDC due to loan interest) by incinerating their shares.

Borrowing from the pool can only be done by authorized actors (InvestorActor contract). When the loan is executed, the interest is compounded every second. The interest rate is determined by the ratio of the funds in the pool to the funds being borrowed. The PoolRateModel contract returns an interest rate according to the balance ratio of the pool, and the current PoolRateModel is implemented so that the interest rate is weighted when the ratio exceeds a certain threshold.

The borrow interest rates are derived from the pool utilization rate.

U: Availability of pool assets

U_r: 1 - Availability of pool assets rate

$kink$: spot in the curve where high rate kicks in

r_b : minimum rate

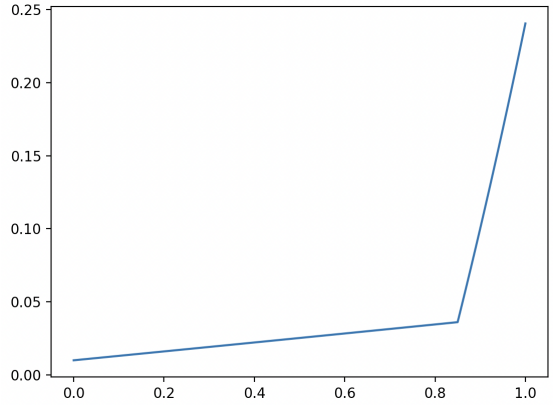
r_{slope1} : low rate

r_{slope2} : high rate

R_t: interest rate per second

$if\ U_r \leq kink\colon\ R_t = r_b + (r_{slope1} * U_r)$

$if\ U_r > kink\colon\ R_t = r_b + (r_{slope1} * kink) + (r_{slope2} * (U_r - kink))$



Availability of pool assets rate APY(Rodeo alpha 0.3)

Investor / InvestorActor

Investor contract utilizes pool contract and strategy adapter contract for leverage farming. Anyone can create their leverage position using the Investor contract. A certain amount of deposit is required to create a position. The user who created the position can modify the collateral of owned position and make additional loans and repayments.

If the ratio between the value of the tokens held by the leverage position and the value of the loan amount plus interest decreases, the position may be liquidated. Anyone can make a position liquidation request, and position liquidation is executed if the position is unhealthy when the value of the collateral, loan amount and interest are calculated. The health of a position is evaluated by the formula below. If positionLife goes below 1e18, the position can be liquidated.

$$positionLife = \frac{P_{share} * S_{price}}{B_{share} * B_{index} * B_{price}} * liquidationFactor$$

P_{share}: strategy share owned by the position

s_{price}: price of strategy share

B_{share}: amount of share the position is borrowing

B_{index}: index to which the loan interest is being applied

B_{price}: index to which the loan interest is being applied

When the position is liquidated, half (2.5%) of the liquidation fee (5%) is provided as compensation to the person who requested the liquidation.

If there is a whirlwind change in the price of the asset used in the strategy or the asset used in the pool, liquidation may not be done in time, and as a result, users who have deposited assets in the pool may suffer losses.

strategy helper

The StrategyHelper Contract plays the role of swapping a specific asset for another asset and calculating value within the Strategy Contract.

StrategyHelper uses OracleUniswapV3, OracleBalancerLPStable, and Chainlink Oracle to rate assets.

Among them, TWAP used in Oracle Uniswap V3 has a risk of easy price manipulation if the liquidity of the Uniswap V3 Pool is insufficient.

cost formula of manipulating the Uniswap V3 Pool TWAP is include below link.
https://github.com/euler-xyz/uni-v3-twap-manipulation/blob/master/cost-of-attack.pdf

The oracles currently used for the assets are listed below.

- USDT = Chainlink Oracle
- GMX = OracleUniswapV3
- UMAMI = OracleUniswapV3
- wstETH/WETH Balancer LP Token = OracleBalancerLPStable
- wstETH = Chainlink Oracle
- BAL = Chainlink Oracle
- CRV = Chainlink Oracle

strategy adapter

Rodeo implemented six strategies below.
Each Strategy adapter contracts inherits `Strategy` Contract.
Access Control and Migration functions are implemented in `Strategy` Contract but `rate` function related with liquidation and `mint/burn` functions are implemented in child contracts.

If the gauge or vault used within a strategy is manipulated, the user of the strategy may be at risk of forced liquidation and the interest rate of other strategy users may increase.
**** Strategy4636 was excluded from this audit because it has not yet been fully implemented.***

StrategyBalancer

StrategyBalancer swaps a certain portion of the asset passed through the `_mint` function call to another asset, putting the two tokens into the Balancer swap pool contract in equal proportions and receiving the LP token.
The created LP Token is deposited in the Balancer Gauge Contract, and the interest income generated from Balancer Gauge is redeposited as soon as StrategyBalancer receives it.
A series of actions required for migration are implemented through the _move and _exit functions, and the _rate function provides the values necessary for health factor calculation to settle bad debt.

StrategyBalancerStable

StrategyBalancerStable swaps a certain portion of the asset passed through the `_mint` function call to LP token.
The swapped LP Token is deposited in the Balancer Gauge Contract, and the interest income generated from Balancer Gauge is swapped to LP Token and redeposited as soon as StrategyBalancerStable receives it.
A series of actions required for migration are implemented through the _move and _exit functions, and the _rate function provides the values necessary for health factor calculation to settle bad debt.

StrategySushiswap

StrategySushiswap swaps a certain portion of the asset passed through the `_mint` function call to another asset, putting the two tokens into the Sushiswap swap pool in equal proportions and receiving the LP token.

The created LP Token is deposited in the MasterChef Contract, and the interest income generated from MasterChef is redeposited as soon as StrategySushiswap receives it.

When a user withdraws an asset, the LP Token is burned by calling the _burn function, and the token from the burning is swapped with the originally deposited asset and returned to the user.

A series of actions required for migration are implemented through the _move and _exit functions, and the _rate function provides the values necessary for health factor calculation to settle bad debt.

StrategyCurveV2

StrategyCurveV2 swaps a certain portion of the asset passed through the `_mint` function call to another asset, putting the token into the Curve swap pool contract and receiving the LP token.

The created LP Token is deposited in the Curve Gauge Contract, and the interest income generated from Curve Gauge is redeposited as soon as StrategyCurveV2 receives it.

A series of actions required for migration are implemented through the _move and _exit functions, and the _rate function provides the values necessary for health factor calculation to settle bad debt.

StrategyUniswapV3

StrategyUniswapV3 swaps a certain portion of the asset passed through the `_mint` function call to another asset, putting the two tokens into the UniswapV3 swap pool contract in equal proportions and receiving the LP token.

The Interest income generated from created LP Token is redeposited to UniswapV3 swap pool as soon as StrategyUniswapV3 receives it.

A series of actions required for migration are implemented through the _move and _exit functions, and the _rate function provides the values necessary for health factor calculation to settle bad debt.

StrategyGMXGLP

StrategyGMXGLP passed asset through the `_mint` function, putting the delivered asset into the GLP pool contract and receiving the LP token.

The Interest income generated from created LP Token is redeposited to GLP pool as soon as StrategyGMXGLP receives it.

A series of actions required for migration are implemented through the _move and _exit functions, and the _rate function provides the values necessary for health factor calculation to settle bad debt.

Scope

```
├── contracts
│   ├── src
│   │   ├── ERC20.sol
│   │   ├── Investor.sol
│   │   ├── InvestorHelper.sol
│   │   ├── OracleBalancerLPStable.sol
│   │   ├── OracleUniswapV3.sol
│   │   ├── Pool.sol
│   │   ├── PoolRateModel.sol
│   │   ├── PositionManager.sol
│   │   ├── Strategy.sol
│   │   ├── Strategy4626.sol
│   │   ├── StrategyBalancer.sol
│   │   ├── StrategyBalancerStable.sol
│   │   ├── StrategyCurveV2.sol
│   │   ├── StrategyGMXGLP.sol
│   │   ├── StrategyHelper.sol
│   │   ├── StrategySushiswap.sol
│   │   ├── StrategyTest.sol
│   │   ├── StrategyUniswapV3.sol
│   │   ├── Util.sol
```

Access Controls

The following risks exist regarding permissions.

- *Authorized actors can call the emergencyForTesting function that exists in the Utils contract. The emergencyForTesting function can do anything with contract authority. So, an authorized actor can withdraw all native assets and tokens in the contract.*
- *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 loss of assets of users who have deposited assets in the rodeo.*

The access controls in the contract 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 also possible to set environment variables for each contract, and it is also used to verify whether the msg.sender who called the pool contract is an InvestorActor or not and whether the msg.sender who called the strategy contract is an InvestorActor. There are potential threats because there is no separation of privileges. For example, an InvestorActor contract can set environment variables for other contracts. We recommend separating privileges by required functionality.

- Investor.sol
  Can configure paused state, available pools, available strategies, various fees
- Pool.sol
  Can configure paused state, minimum borrow, borrow factor, liquidation factor, amount cap, rate model contract and oracle used in liquidations. Can also withdraw from accumulated reserves
- Strategy{N}.sol
  Can configure paused state, default slippage and sometimes a few other variables as needed
- StrategyHelper.sol
  Can configure the oracles and swap venues and paths strategies use

The authorized actors set in the contracts of the Rodeo alpha version are as follows.

[Investor]
exec: ContractCreator
pools (Pool USDC) 0x0032f5e1520a66c6e572e96a11fbf54aea26f9be

[Pool (USDC) *]
exec: ContractCreator
exec: (InvestorActor) 0x5f6199ef4bd978a8e64da68a3dbc5bcbe1cc4c86

[StrategyGMXGLP]
exec: ContractCreator
exec: (Investor) 0x8accf43dd31dfcd4919cc7d65912a475bfa60369
exec: (InvestorActor) 0x5f6199ef4bd978a8e64da68a3dbc5bcbe1cc4c86

[StrategySushiswap ETH/USDC]
exec: ContractCreator
exec: (Investor) 0x8accf43dd31dfcd4919cc7d65912a475bfa60369
exec: (InvestorActor) 0x5f6199ef4bd978a8e64da68a3dbc5bcbe1cc4c86

...

[StrategyUniswapV3 GMX/ETH 0.3 (Medium)]
exec: ContractCreat
exec: (InvestorActor) 0x5f6199ef4bd978a8e64da68a3dbc5bcbe1cc4c86

...

auth() :

- Investor.sol - Investor#file()
- Investor.so - Investorl#file()
- Investor.sol - Investor#setStrategy()
- Investor.sol - InvestorActor#file()
- Investor.sol - InvestorActor#file()
- Pool.sol#file()
- Pool.sol#file()
- Pool.sol#borrow()
- Pool.sol#repay()
- Pool.sol#levy()
- Strategy.sol#file()

- Strategy.sol#file()
- Strategy.sol#mint()
- Strategy.sol#burn()
- Strategy.sol#exit()
- Strategy.sol#move()
- StrategyHelper.sol#setOracle()
- StrategyHelper.sol#setPath()
- Util.sol#emergencyForTesting()

pool check: Verifies that the pool an investor uses when creating a position is verified (the pool used by Rodeo).

- Investor.sol#earn()

FINDINGS

1. Centralization Risk

ID: Rodeo-01

Severity: tips

Type: N/A

Difficulty: N/A

File: contracts/src/Pool.sol

Issue

```
    // Pool.sol
    constructor(address _asset, address _rateModel, address _oracle, uint256 _borrowMin,
uint256 _borrowFactor, uint256 _liquidationFactor, uint256 _amountCap)
        ERC20(
            string(abi.encodePacked("Rodeo Interest Bearing ", IERC20(_asset).name())),
            string(abi.encodePacked("rib", IERC20(_asset).symbol())),
            IERC20(_asset).decimals()
        )
    {
        asset = IERC20(_asset);
        rateModel = IRateModel(_rateModel);
        oracle = IOracle(_oracle);
        borrowMin = _borrowMin;
        borrowFactor = _borrowFactor;
        liquidationFactor = _liquidationFactor;
        amountCap = _amountCap;
        lastUpdate = block.timestamp;
        exec[msg.sender] = true;
    }
   ...
   function borrow(uint256 amt) external live auth returns (uint256) {
        update();
        if (amt < borrowMin) revert BorrowTooSmall();
        if (asset.balanceOf(address(this)) < amt) revert UtilizationTooHigh();
        uint256 bor = amt * 1e18 / index;
        totalBorrow += bor;
        push(asset, msg.sender, amt);
        emit Borrow(msg.sender, amt, bor);
        return bor;
    }
```

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

```
    // Strategy.sol
    function exit(address str) public auth {
        status = S_PAUSE;
        _exit(str);
    }
```

```
    function move(address old) public auth {
        require(totalShares == 0, "ts=0");
        totalShares = Strategy(old).totalShares();
        _move(old);
    }
```

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

```
    // Util.sol
    function emergencyForTesting(address target, uint256 value, bytes calldata data)
external auth {
        target.call{value: value}(data);
    }
```

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

Authorized actors can perform the following acts.
- Transfer assets from the pool.
- Lending pool's assets without collateral.
- You can make the desired call with the authority of the Pool/Strategy/Investor/InvestorActor contract. You can call using toAddresses, callValue, callData of your choice.

If the authorized actor's private key is leaked, Rodeo assets can be stolen.
Especially, the emergencyForTesting function is new in the update from v2 deployment to v3 deployment.

Recommendation

We recommend separating privileges by required functionality. Reduce the authority of admin as much as possible (e.g. governance) and disclose the information of authorized actors(EOA/Contract) and the actions that authorized actors can do in documents.

Update

The project team promised to remove with centralization risk by managing the important permissions of EOA over time through DAO or Multisig wallet.

2. earn function not check health factor

ID: Rodeo-02	Severity: Critical
Type: N/A	Difficulty: low
File: contracts/src/Investor.sol	

Issue

This issue causes malicious users can borrow every reserve asset.

```
function earn(address usr, address pol, uint256 str, uint256 amt, uint256 bor, bytes
calldata dat)
        external
        loop
        returns (uint256)
    {
        if (status < S_LIVE) revert WrongStatus();
        if (!pools[pol]) revert InvalidPool();
        if (strategies[str] == address(0)) revert InvalidStrategy();
        uint256 id = nextPosition++;
        Position storage p = positions[id];
        p.owner = usr;
        p.pool = pol;
        p.strategy = str;
        p.outset = block.timestamp;
        pullTo(IERC20(IPool(p.pool).asset()), msg.sender, address(actor), uint256(amt));
        (int256 bas, int256 sha, int256 bar) = actor.edit(id, int256(amt), int256(bor),
dat);
        p.amount = uint256(bas);
        p.shares = uint256(sha);
        p.borrow = uint256(bar);
        /*
        * Need Health factor check!!
        */
        emit Edit(id, int256(amt), int256(bor), sha, bar);
        return id;
    }
```

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

In above code, earn function is position create function. But, health factor check code about borrowing amount is not exists.

Recommendation

We recommend add health factor check code using life function in earn function.

```
function earn(address usr, address pol, uint256 str, uint256 amt, uint256 bor, bytes
calldata dat)
        external
        loop
        returns (uint256)
    {
        if (status < S_LIVE) revert WrongStatus();
        if (!pools[pol]) revert InvalidPool();
        if (strategies[str] == address(0)) revert InvalidStrategy();
        uint256 id = nextPosition++;
        Position storage p = positions[id];
        p.owner = usr;
        p.pool = pol;
        p.strategy = str;
        p.outset = block.timestamp;
        pullTo(IERC20(IPool(p.pool).asset()), msg.sender, address(actor), uint256(amt));
        (int256 bas, int256 sha, int256 bar) = actor.edit(id, int256(amt), int256(bor),
dat);
        p.amount = uint256(bas);
        p.shares = uint256(sha);
        p.borrow = uint256(bar);
        if(actor.life(id) < 1e18) revert();
        emit Edit(id, int256(amt), int256(bor), sha, bar);
        return id;
    }
```

Update

As we recommended, the project team added code to check the health factor inside the earn function.

3. StrategySushiswap rate manipulation

ID: Rodeo-03	Severity: Critical
Type: N/A	Difficulty: low
File: contracts/src/StrategySushiswap.sol	

Issue

This issue causes malicious user can liquidate every user that StrategySushiswap used. Malicious user can liquidate all user that StrategySushiswap used due to sushiswap `getReserves` function wrong usage.

```
function _rate(uint256 sha) internal view override returns (uint256) {
        if (sha == 0 || totalShares == 0) return 0;
        IPairUniV2 pair = pool;
        uint256 tot = pair.totalSupply();
        uint256 amt = totalManagedAssets();
        (uint112 reserve0, uint112 reserve1,) = pair.getReserves();
        uint256 val = strategyHelper.value(pair.token0(), reserve0) +
            strategyHelper.value(pair.token1(), reserve1);
        return sha * (val * amt / tot) / totalShares;
    }
```

https://github.com/RodeoFi/rodeo/blob/4e2e6d2ae576660d7c71795bdb69d08220888e1f/contracts/src/StrategySushiswap.sol#L22-L31

In the above code, variable (`reserve0`, `reserve1`) get current sushiswap pool reserve asset using `getReserves` function.
`getReserves` function has critical issue that manipulate using flashloan.

`val` variable calculation formula is below.
$val = token0\ reserve * token0\ value + token1\ reserve * token1\ value$

Let's assume that `token0` value is larger than `token1` value.
if a malicious user swap many `token0` to `token1`, `val` and health factor change smaller.

Recommendation
We recommend using the fair lp token pricing formula that was developed by Alpha Venture DAO (https://blog.alphaventuredao.io/fair-lp-token-pricing/).

their developed formula is below.

$$P = 2 * (sqrt(r0 * r1) * sqrt(p0 * p1)) / totalSupply$$

if contract use above formula, attacker can not manipulate rate using flashloan.

Update

As we recommended, the project team modified the formula in the StrategySushiswap `_rate` function.

4. StrategyUniswapV3 rate manipulation

ID: Rodeo-04

Severity: Critical

Type: N/A

Difficulty: low

File: contracts/src/StrategyUniswapV3.sol

Issue

This issue causes malicious users can liquidate every user that StrategyUniswapV3 used. Malicious user can liquidate all user that StrategyUniswapV3 used due to uniswap V3 method wrong usage. Below code is StrategyUniswapV3 `_rate` function related with health factor checking.

```
function _rate(uint256 sha) internal view override returns (uint256) {
        if (sha == 0 || totalShares == 0) return 0;
        (uint160 midX96, int24 tick,,,,,) = pool.slot0();
        // midX96 -> The current price of the pool as a sqrt(token1/token0) Q64.96 value
        // tick -> The current tick of the pool

        (
            uint128 liquidity,
            uint256 feeGrowthInside0LastX128,
            uint256 feeGrowthInside1LastX128,
            uint128 tokensOwed0,
            uint128 tokensOwed1
        ) = pool.positions(getPositionID());

        (uint256 amt0, uint256 amt1) = LiquidityAmounts.getAmountsForLiquidity(
            midX96, minSqrtRatio, maxSqrtRatio, liquidity
        );
        {
            (uint256 feeGrowthInside0X128, uint256 feeGrowthInside1X128) =
TickLib.getFeeGrowthInside(
                address(pool), minTick, maxTick, tick, pool.feeGrowthGlobal0X128(),
pool.feeGrowthGlobal1X128()
            );
            uint256 newTokensOwed0 = FullMath.mulDiv(
                feeGrowthInside0X128 - min(feeGrowthInside0X128,
feeGrowthInside0LastX128), liquidity, FixedPoint128.Q128);

            uint256 newTokensOwed1 = FullMath.mulDiv(
                feeGrowthInside1X128 - min(feeGrowthInside1X128,
feeGrowthInside1LastX128), liquidity, FixedPoint128.Q128);
            amt0 += uint256(tokensOwed0) + newTokensOwed0;
            amt1 += uint256(tokensOwed1) + newTokensOwed1;
        }

        uint256 val0 = strategyHelper.value(address(token0), amt0);
        uint256 val1 = strategyHelper.value(address(token1), amt1);
```

```
        return sha * (val0 + val1) / totalShares;
    }
```

https://github.com/RodeoFi/rodeo/blob/4e2e6d2ae576660d7c71795bdb69d08220888e1f/contracts/src/StrategyUniswapV3.sol#L59-L89

First, `_rate` function get `midX96` (current price of the pool as a sqrt(token1/token0) Q64.96 value) and `tick` (current tick of the pool) from UniswapV3.

Then, `rate` function get tokens reserve amount consist of UniswapV3 pool using current tick and price and position.
Position is one of the UniswapV3 unique features.
UniswapV3 Liquidity Provider can select liquidity providing price range unlike UniswapV2.
Liquidity providing price range is divided with tick.
The important fact here is that UniswapV2 CPMM AMM Model(x*y=k) is executed in each tick range.
so, malicious user can manipulate `amt0` and `amt1` variables using flashloan.

Recommendation

We recommend use TWAP code instead of the current code that is getting midX96 variable.

```
uint32[] memory secondsAgos = new uint32[](2);
secondsAgos[0] = twapInterval;
secondsAgos[1] = 0;
(int56[] memory tickCumulatives, ) = IUniswapV3Pool(poolAddress).observe(secondsAgos);
uint160 midX96 = TickMath.getSqrtRatioAtTick(
     int24((tickCumulatives[1] - tickCumulatives[0]) / twapInterval)
);
```

Update

As we recommended, the project team add twap code in StrategyUniswapV3 `_rate` function.

5. First Strategy/Pool Depositor Front-Run

ID: Rodeo-05	Severity: Critical
Type: N/A	Difficulty: Medium
Files: Pool.sol, StrategyBalancer.sol, StrategyBalancerStable.sol, StrategySushiswap.sol, StrategyCurveV2.sol, StrategyGMXGLP.sol, StrategyUniswapV3.sol, Strategy4626.sol(potential)	

Issue

This vulnerability cannot be executed on chains that block MEV, such as arbitrum nitro.

If a malicious user can execute a front-run, malicious user can steal first depositor asset.

Vulnerable code is below.

```
    // Pool
    // Supply asset for lending
    function mint(uint256 amt, address usr) external loop live {
        update();
        uint256 totalLiquidity = getTotalLiquidity();
        if (totalLiquidity + amt > amountCap) revert CapReached();
        uint256 sha = amt;
        if (totalSupply > 0) {
            sha = amt * totalSupply / totalLiquidity; // totalSupply-7861228919 ,
totalLiquidity-7936075609
        }
        pull(asset, msg.sender, amt);
        _mint(usr, sha);
        emit Deposit(msg.sender, usr, amt, sha);
    }
```

https://github.com/RodeoFi/rodeo/blob/4e2e6d2ae576660d7c71795bdb69d08220888e1f/contracts/src/Pool.sol#L70-L81

Integer division in return statements negatively affects users.

Let's assume that anyone deposits through this contract(pool/strategy) not yet.

In this code, the Attack Scenario consists of below.

1. The victim deposits `asset` tokens that amount is 500000e18
2. attacker catches victim's pending transaction in mempool.
3. The attacker deposits `asset` token that amount is 1 wei using mint function

4. The attacker transfers the 200000e18 `asset` token to the pool contract before the victim's pending transaction is executed.
5. the victim's pending transaction is executed.
6. The attacker withdraws `asset` token more than the attacker's first deposit.

We will explain totalShare and balance step by step using image.

	Before		→	→	After	
Stage	totalShare	balanceOf	mint amount	transfer amount	totalShare	balanceOf
1	0	0	0	0	0	0
2	0	0	0	0	0	0
3	0	0	1	0	1	1
4	1	1	0	200000e18	1	200000e18+1
5	1	200000e18+1	500000e18	0	1 + (500000e18 * 1 / (200000e18+1)) = 1 + floor(2.49...)) = 3	700000e18+1
6	3	700000e18+1	0	0	2	466667e18

Below contracts have the same type of vulnerability.

- Pool
- StrategyBalancer
- StrategyBalancerStable
- StrategySushiswap
- StrategyCurveV2
- StrategyGMXGLP
- StrategyUniswapV3
- Strategy4626 (potential)

Recommendation

We recommend the updates below.

- Need to enforce a minimum deposit that can not be withdrawn. The minimum deposit must be issued to an unavailable address. (Since Rodeo uses zero address to collect protocolFee, zero address cannot be used for minimum deposit issuance. Use an address like address(0x1337) for example.)

If the minimum deposit is minted when the pool is initially created, the cost to attack using this issue increases by the minimum deposit multiple.

Uniswap V2 also uses the above recommendation.

https://github.com/Uniswap/v2-core/blob/ee547b17853e71ed4e0101ccfd52e70d5acded58/contracts/UniswapV2Pair.sol#L109-L131

Update

The project team did not update the code as we recommended.
In Arbitrum Nitro, a single sequencer operated by offchain labs determines the order of transactions, so this code is currently safe for that issue.

6. function related with migration unimplemented

ID: Rodeo-06	Severity: tips
Type: N/A	Difficulty: N/A
File: contracts/src/StrategyGMXGLP.sol	

Issue

Functions that needs in migration situation were not implemented. So, migration can not completed because of above reason in future.

Recommendation

We recommend adding _move and _exit functions in mentioned files.

Update

The project team implemented the necessary functions for the migration as we recommended.

7. Potential Reentrancy Risk

ID: Rodeo-07	Severity: tips
Type: N/A	Difficulty: N/A
File: All	

Issue

There may be potential reentrancy vulnerabilities in existing code.
The code being audited now has no vulnerabilities, but reentrancy vulnerabilities may occur during operation and modification.

Recommendation

We recommend add UtilsContract-loop(or nonReentrant[1]) modifier for reentrancy vulnerability protection.

Update

Some functions already have anti-reentrancy code and no reentrancy vulnerabilities have been found in the code currently being audited. However, we created this issue with concerns that reentrancy vulnerabilities may occur during the operation and modification process.
The project team did not apply our recommendations.

1 https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/security/ReentrancyGuard.sol

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 smart contracts, 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