yAcademy Singularity v2 review

Review Resources:
Docs

Residents:

- NibblerExpress
- engn33r

Fellows:

- ben s
- blockdev
- devtooligan
- SaharAP
- toastedsteaksandwich
- uk
- verypoor

Guest Auditor:

- cmichel

Table of Contents

Review Summary

Singularity v2

The Singularity v2 protocol is an AMM providing single-sided ERC20 token pools. These pools allow users to perform single-sided deposits, without a token pair counterpart as required in AMMs like Uniswap or Sushiswap. Singularity v2 uses a collateralization ratio with a custom curve to calculate the fees for deposit, withdraw, and swap operations. Singularity v2 is not permissionless and relies on an admin to create new pools, pause pools, and set deposit/withdraw caps. The design of Singularity v2 is similar to Uniswap in that users interact with a router and a factory creates new pools for new tokens.

The main branch of the Singularity v2 Repo was reviewed over 15 days, 2 of which were used to create an initial overview of the contract. The code review was performed between June 12 and June 27, 2022. The code was reviewed by 2 residents for a total of 35 man hours (NibblerExpress: 8 hours, engn33r: 27 hours). A number of yAcademy Fellows also reviewed the contracts and contributed over 50 man hours. The repository was under active development during the review, but the review was limited to one specific commit.

Scope

Code Repo
Commit

The commit reviewed was a3cdbc5515374c9e1792b3cb94ff1b084a9a1361. The scope of the review consisted of three contracts in the repository at this specific commit:

1. SingularityFactory.sol
2. SingularityRouter.sol
3. SingularityPool.sol

After the findings were presented to the Singularity v2 team, fixes were made and included in several PRs.

The review was a time-limited review to provide rapid feedback on potential vulnerabilities. The review was not a full audit. The review did not explore all potential attack vectors or areas of vulnerability and may not have identified all potential issues.

yAcademy and the residents make no warranties regarding the security of the code and do not warrant that the code is free from defects. yAcademy and the residents do not represent nor imply to third parties that the code has been audited nor that the code is free from defects. Singularity and third parties should use the code at their own risk.

Code Evaluation Matrix

Category	Mark	Description
Access Control	Average	The onlyAdmin modifier was applied to function in the factory contract. Access controls are applied where needed. msg.sender is properly used so that the user cannot perform actions they should not be able to.
Mathematics	Good	The FixedPointMathLib math library is used properly and is considered a relatively trusted implementation.
Complexity	Average	The Factory and Router contracts have relatively low complexity because they are mostly forked from Uniswap, but the Pool contract has substantial complexity. The complexity arises from the collateralization ratio's impact on fees and slippage and the interactions between different Singularity pools during swap operations. More extensive modeling of this complexity should be performed in order to prevent unexpected edge cases from resulting in loss of user value.
Libraries	Average	Singularity v2 relies on some solmate contracts (for ERC20, ReentrancyGuard, FixedPointMathLib) and Chainlink integration. This is an average number of dependencies compared to similar protocols. Because the solmate libraries are copied into the Singularity project manually, they should be checked for vulnerabilities or updated versions before production release.
Decentralization	Average	The protocol relies on an admin to create new pools, pause/unpause pools, and perform other actions. While this is a less decentralized design than a DEX like Uniswap, if the admin role is assigned to a trusted multisig, the centralization level would be comparable to other similar protocols. Additional checks on certain admin-controlled values would increase user trust that the admin would have more limited abilities to modify fees.
Code stability	Average	The lack of updated documentation and recent changes demonstrate the protocol is still under development. A dev branch was created while the yAcademy review was ongoing with some changes applied to this branch.
Documentation	Low	There is a significant lack of NatSpec comments around the functions in all contracts. NatSpec documentation should be added for all functions. Beyond comments in the contracts, some of the online documentation was outdated compared to the implementation in the code.
Monitoring	Good	Events are emitted consistently from all functions that control value transfers.
Testing and verification	Average	The test coverage is over 90% for the entire protocol. Despite the high amount of coverage, the findings in this report demonstrate that the tests should cover more edge cases than they currently do.

Findings Explanation

Findings are broken down into sections by their respective impact:

- Critical, High, Medium, Low impact
  - These are findings that range from attacks that may cause loss of funds, impact control/ownership of the contracts, or cause any unintended consequences/actions that are outside the scope of the requirements

- Gas savings
    - Findings that can improve the gas efficiency of the contracts
- Informational
    - Findings including recommendations and best practices

High Findings

Residents High findings

1. High - Possible locking of pool fee funds (engn33r)

The SingularityFactory.sol `collectFees()` function loops through the `allPools[]` array to call `collectFees()` for each pool. Because the function loops through the entire array every time that it is called, if too many pools are added to the Singularity protocol, this function could run out of gas. This would lock the fees in the pools without an alternative to withdraw them.

Technical Details

The SingularityPool.sol `collectFees()` function has a `onlyFactory()` modifier, so fees can only be collected by the factory. Unlike the other SingularityFactory.sol functions with the onlyAdmin modifier, there are no function arguments to the `collectFees()` function that can control which pools have fees collected from them. Because the for loop will call `collectFees()` for every single pool, if the Singularity protocol has a large number of pools in the future, the fees may be locked and impossible to withdraw.

Impact

High. Fees may be locked, which prevent the protocol admin from collecting the value that has been accruing.

Recommendation

Allow fees to be collected one pool at a time, either by adding a new function in SingularityFactory.sol or modifying the existing function, so that there is a way to collect fees if there is a very large number of pools.

Developer response

Fixed in commit 57541d07f6e43648e42de8f8ee3cd2f1be8a0d35

2. High - Pools mechanics don't incentivize end goal (engn33r)

The Singularity mechanics are designed for pools to stay near a collateralization ratio of 1. Undercollateralized pools should receive deposits or swapIns to increase the collateralization ratio. The documentation says there should be zero fees for deposits of undercollateralized pools, but the code implements a fee for this case. A deposit fee disincentivizes deposits for undercollateralized pools, making it possible that a very undercollateralized pool will not recover to a collateralization ratio of 1.

Technical Details

The `getDepositFee()` function calculates a fee for deposits when the collateralization ratio is under 1. The same happens in `getWithdrawalFee()` when the collateralization ratio is over 1. In both of these functions, the user is disincentivized from moving the collateralization ratio closer to 1.

Impact

High. The protocol mechanics should incentivize increasing the health of unhealthy pools, but adding a fee could lead to a case where a pool is stuck in an unhealthy state where neither deposits or withdrawals are worthwhile, leading to user loss of value.

Recommendation

Remove deposit fees when collateralization is under 1. Withdrawal fees could be removed when collateralization is over 1, though a collateralization over 1 is less hazardous for the protocol than a collateralization under 1. While commits with these changes to deposit and withdraw were made in the dev branch while the review was in progress, the issue still exists after these changes.

The key issue is that the fee structure should incentivize actions that move the collateralization ratio closer to 1, but the current design does not always do this. A visual view of the collateralization ratio impact from each pool function is below (only the non-linear g function values are considered, so collateralization ratio values over 0.3).

`deposit()`: --> 1 <-- (collateralization ratio moves towards 1)
`withdraw()`: <-- 1 --> (collateralization ratio moves away from 1)
`swapIn()`: --> 1 --> (collateralization ratio increases)
`swapOut()`: <-- 1 <-- (collateralization ratio decreases)

Because `deposit()` causes the collateralization ratio to always move towards 1 (which is good for pool stability) and `withdraw()` causes the collateralization ratio to move away from 1 (which is bad for pool stability), in theory deposits should never have a fee (to incentivize pool stability) while withdrawals should always have a fee (to disincentivize pool instability). A similar approach can be used for `swapIn()` and `swapOut()`, where `swapIn()` has a zero fee when the collateralization ratio is less than 1 and `swapOut()` has a zero fee when the collateralization ratio is greater than 1. The full impact of these suggested changes were not examined in detail, but setting the proper incentives for to penalize moving the collateralization ratio away from zero would go a long way to preventing a bank run scenario.

Developer response

Fixed in commit 04b4c710b3d9c3354f63c27f53cb16fe14741591

3. High - Flashloan/Whale slippage manipulation (engn33r, NibblerExpress)

The amount of positive slippage for `swapIn()` and negative slippage for `swapOut()` can be changed by depositing or withdrawing tokens before or after the swap. In the worst case, a user can make trades that result in more positive slippage than negative slippage and fees combined. In other cases, the user can reduce slippage paid to the protocol through manipulation.

Category	`SwapIn()`	`SwapOut()`
Collateralization Ratio < 1	Reducing liquidity increases slippage	Reducing liquidity increases slippage
Collateralization Ratio = 1	Increasing liquidity increases slippage	Reducing liquidity increases slippage
Collateralization Ratio > 1	Increaseing liquidity increases slippage	Increasing liquidity increases slippage

A user looking to benefit their own trades will want to increase liquidity for the `swapIn()` pool (e.g., by depositing pretransaction and withdrawing posttransaction) when the collateraization ratio is at least one. The user will want to similarly increase liquidity for the `swapOut()` pool when the collateralization ratio is no more than one. For an attacker sandwiching another user's loan, the attacker will want to increase liquidity for the `swapIn()` pool when the collateralization ratio is below one and increase liquidity for the `swapOut()` pool when the collateralization ratio is more than one. (Note that for the changes make the fee for `deposit()` zero and make the fee for `withdraw()` zero, the attacker can deposit to and withdraw from the `swapIn()` pool without fees when the collateralization ratio starts out below one and ends at or above one due to the swap.)

Using this pool behavior, a specific series of pool actions under a specific set of pool conditions can be profitable for a majority stakeholder of a pool, such as a whale. The example below examines a scenario with two stablecoin pools, but in theory the same approach can be used between any two pools if sufficient profit exists. Although the `swapIn()` operation provides the profit, the value is extracted from the pool used in the `swapOut()` operation when the `safeTransfer()` call transfers tokens out of the protocol. The positive slippage for `swapIn()` and negative slippage for `swapOut()` increases as the collateralization ratio of the pool declines well below 1. This is because `gDiff` is the difference in g values, which increases as the slope of g increases nearer to zero and becomes a significant factor at around 0.4.

Technical Details

For a swap of size `x`, you can compute the limit of the `gPrime` equation as assets and liabilities go to infinity and assets divided by liabilities goes to one (i.e., someone makes an infinitely large deposit). This limit asymptotically approaches `.0003*8*x`. When the collateralization ratio equals one, users and attackers get a small benefit from manipulating the pool unless the transaction is a substanial percentage of the size of the pool. A user or attacker gets more benefit when the collateralization ratio is not one, and the user or attacker can either push the collateralization ratio much closer to one or much further from one (whichever is more profitable) where the slippage can be much greater than or much less than `.0003*8*x`.

One example of a profitable arbitrage can take place if:

1. A single user is the majority pool depositor and can thereby control the collateralization ratio
2. The collateralization ratio of the pool drops below 1
3. The real world follows the assumption below (for simplicity) that the oracle price of the two stablecoins involved are identical
4. `tradingFee` is low enough for this strategy to profit

Consider the following scenario for a stablecoin pool:

USDC Pool status before transaction 1

Whale has deposited 23,400,000 into the Singularity USDC pool

- assets = 24,000,000
- liabilities = 25,000,000
- collateralization ratio = 0.96
- getG = 0.0000415864

txn 1: Whale withdraws 23,400,000 in value. The withdrawal fee is 122,403.75, resulting in a temporary loss for the whale. The collateralization ratio drops significantly.

USDC Pool status before transaction 2

- assets = 600,000
- liabilities = 1,600,000
- collateralization ratio = 0.375
- getG = 0.0767134
- tradingFee = 0.00001 (0.001%)

DAI Pool status before transaction 2

- assets = 15,000,000
- liabilities = 10,000,000
- collateralization ratio = 1.5
- getG = 0.0767134

- tradingFee = 0.00001 (0.01%)

txn 2: Whale swaps in 999,999 USDC to the USDC pool and swapsOut of the DAI pool. Why 999,999 and not 1,000,000? So that the collateralization ratio remains less than 1, allowing Whale to deposit (the collateralization ratio will remain under 1 after the deposit) and cycle through the process again if a suitable pool exists to profit after the `swapOut()` fees and slippage are considered. Whale receives 1,122,558.81 in DAI, resulting in a 122,559.81 profit from the positive slippage in the USDC pool. After subtracting the first transaction's withdrawal fee cost, there is a $156.06 profit. If Whale needed to increase the DAI pool collateralization ratio to reduce the `swapOut()` slippage, Whale can withdraw from the DAI pool immediately after withdrawing from the USDC pool (consider this transaction 1.5) in order to increase the DAI pool collateralization ratio and decrease the slippage costs.

USDC Pool status before transaction 3

- assets = 1,599,999
- liabilities = 1,600,000
- collateralization ratio = 0.999999375
- getG = 0.00003

txn 3: Whale deposits 23,400,001 into the USDC pool and the USDC pool returns to where it started, except Whale profited and can rinse and repeat this process if another pool with low enough `swapOut()` fees exists. The DAI pool collateralization ratio is closer to 1, so it may not be profitable to `swapOut()` from the DAI pool again.

USDC Pool status after transaction 3

- assets = 24,000,000
- liabilities = 25,000,001
- collateralization ratio = 0.999999375
- getG = 0.00003

Impact

High. Flashloan or whale can manipulate the pools to generate extra profit from the differential between `slippageIn()` and `slippageOut()` in pools with different collateralization ratios. The profit is removed from the `assets` value, and a low `assets` value can lead to a bank run in an undercollateralized pool where depositors cannot withdraw their liquidity.

Recommendation

Modify the protocol so the assumptions used above are not valid. For example, set a minimum `tradingFee` value on stablecoin pools such that it disincentivizes this arbitrage approach. Consider adding a deposit or withdrawal cooldown period like Bancor does.

Developer response

Acknowledged, will modify trading fees on a per-asset basis

4. High - Single depositor in pool can steal funds from other pools (NibblerExpress)

When there are assets in a pool but no liabilities, the collateralization ratio is always one. As a result, there is no slippage out on a pool with no liabilities. An attacker can immediately remove all assets from the pool by performing a swap without paying any slippage. Swaps can be used to overcollateralize the pool with one depositor and undercollateralize all other pools in the tranche. The attacker can remove the overcollateralization.

Technical Details

The `_calcCollatalizationRatio` returns one when there are no liabilities, so `gPrime` will be zero. There is zero slippage on a pool with no liabilities.

An attack could then occur when a new pool is added to a tranche with no depositors or when the number of depositors drops to the point of having a single depositor. The attack steps are:

0. Deposit (if not a despoitor already) in the vulnerable pool to become the only depositor.
1. Swap into the vulnerable pool and out of another in the tranche to remove tokens from the other pool.
2. Withdraw all LP tokens from the vulnerable pool so that the liabilities are zero.
3. Swap into the other pool and out of the vulnerable pool to remove all or almost all tokens from the vulnerable pool.
4. Repeat while profitable

Although the negative slippage in step 1 of the attack is greater than the positive slippage, the positive slippage in step 3 makes up for much of the negative slippage in step 1 with no negative slippage during step 3. The profit will be slightly smaller than the positive slippage experienced in step 1.

Impact

High. All assets can be removed from a pool with no liabilities without experiencing any slippage. A single depositer can use swaps to lower the collateralization ratio in pools other than the vulnerable pool and then remove all assets from that pool without experiencing any slippage.

Recommendation

Increase the fee/penalty for withdrawing a pool down to very low or zero liquidity when that produces a very high or infinite collateralization ratio.

Developer response

Acknowledged

Fellows High findings

1. High - allPrices gas grieving causes complete loss of on-chain oracle (devtooligan)

The use of an unbound array to store price data for the on-chain oracle in SingularityOracle.sol for the `allPrices` mapping state variable is problematic as it increases the gas cost of calling `getLatestRound()` every time a new price is added. This problem stems from the fact that the array of `PriceData` structs from `allPrices` is copied to memory in `getLatestRound()`

Proof of concept

Tests were setup that pushed 1, 100, and 1000 prices prior to calling `getLatestRound()`. Foundry reported gas usage as follows:

```
testGetLatestRoundWith1Prices()    (gas:    30_914)
testGetLatestRoundWith100Prices()  (gas:   684_475)
testGetLatestRoundWith1000Prices() (gas: 6_594_464)
```

Impact

At some point, the `getLatestRound()` function will no longer be callable as the gas cost will exceed the gas limit, but admin will likely stop using it before then because of the cost. This would force the oracle to set the `onlyUseChainlink` flag to true and prohibit the use of non-Chainlink price feeds forever. This would render the on-chain oracle system useless, raise the risk attributable to Chainlink sandwich and other attacks, and also significantly increase the protocol's dependence on Chainlink as the sole price provider.

Recommendation

An immediate fix would be to change line 40 in SingularityOracle.sol to use *storage* instead of *memory*. This would then use a pointer to the storage location instead and eliminate the need to copy it to memory.

```
    PriceData[] storage prices = allPrices[token]; // proposed change
```

However, we also noted that in the two places in the code where an `allPrices` array is accessed, only the last element is used. As such, we propose changing the `allPrices` mapping value to be a single `PriceData` instead of an array `PriceData[]` as it is currently declared on line 24.

```
    mapping(address => PriceData) public allPrices; // proposed change
```

If the mapping gets updated, the getLatestRound() and pushPrice() logic would have to be updated to reference a single `PriceData` and not the last element of a `PriceData[]` array.

Developer response

Fixed in commit 808dc21cc701def2d307892e78b0a11720281cef

2. High - Pool drain via Oracle Update Sandwich Attack (Benjamin Samuels)

The token-denominated value of each pool may be drained by an attacker by swapping against a pool immediately before an oracle price update, then immediately after the oracle price update.

This attack was discovered previously and there are some mitigations in the code under audit, however those mitigations were found to be inadequate.

```
function getTradingFeeRate() public view override returns (uint256 tradingFeeRate) {
    if (isStablecoin) {
        return baseFee;
    } else {
        (, uint256 updatedAt) = getOracleData();
        uint256 oracleSens = ISingularityFactory(factory).oracleSens();
        uint256 timeSinceUpdate = block.timestamp - updatedAt;
        if (timeSinceUpdate * 10 > oracleSens * 11) {
            tradingFeeRate = type(uint256).max; // Revert later to allow viewability
        } else if (timeSinceUpdate >= oracleSens) {
            tradingFeeRate = baseFee * 2;
        } else {
            tradingFeeRate = baseFee + (baseFee * timeSinceUpdate) / oracleSens;
        }
    }
}
```

The code above intended to add a volatility fee premium based on the amount of time that has elapsed since the last oracle update. The volatility fee premium scales from `baseFee` up to `2 * baseFee` linear scaling. There are two flaws in this mitigation approach.

1. No volatility fee premium is added for stablecoin pools, leaving said pools exposed to the attack.
2. If the change in oracle price is larger than `2 * baseFee`, then the sandwich attack is still possible.

Proof of concept

Assume there are two singularity pools deployed, one for USDC and one for DAI. Each pool has 10 USDC and 10 DAI in them, and assets are currently equal to liabilities. The Chainlink oracle for each pool is reporting that 1 DAI = $1 and 1 USDC = $1.

The attacker has 1 DAI they will be utilizing in the attack.

The attack is as follows:

1. Chainlink recognizes that the price of DAI has dropped by 1% and queues up a transaction to update the price oracle so 1 DAI = $0.99.
2. Attacker detects the transaction in the mempool, and attempts to sandwich the oracle update transaction;
3. Attacker sends a transaction for `swapExactTokensForTokens()` that will swap 1 DAI for 0.999363 USDC. This transaction must be executed before the chainlink oracle price update transaction.
4. Attacker sends a transaction for `swapExactTokensForTokens()` that will swap 0.999363 USDC for 1.009291253 DAI. This transaction must be executed after the chainlink oracle price update transaction.

When the block is mined, the DAI pool's assets will be 9.990487685, reflecting a drain equal to ~1% of the attacker's starting capital.

The attacker will have 1.009291253 DAI. In real terms, according to the oracle, 1.009 DAI at this point is equal to $1. However, since DAI has a target price of $1, it is very likely the oracle will evaluate 1.009 DAI as == $1.009 in the future, enabling the attacker to repeat this attack whenever there is volatility in the DAI price.

A coded PoC can be found here: https://gist.github.com/bsamuels453/b0fc4715fc384d94eb3e8ef22841f82a

Impact

The amount drained by the attack is dependent on:

1. The attacker's starting capital
2. The difference between the oracle's previous price update and upcoming price update.

Larger oracle price swings (volatility) allow larger amounts to be drained by the sandwich.

The impact of this attack is limited by:

1. The amount of capital required to execute the attack relative to the pool's size.
2. Large slippage fees that occur as the pool becomes undercollateralized.

With regards to the second point, it should be noted that large enough swings in oracle price will prevent slippage protections from stopping this attack. This is discussed further in the recommendations below.

Recommendation

Fundamentally this issue is caused by a mispricing of trade fees. The attack will remain viable as long as there is any possibility that `% change in oracle price > % trade fees + % slippage`. The following recommendations are designed to mitigate that risk as much as possible, although this auditor is not certain whether they are adequate.

Calculate trading fees based off an implied volatility oracle

The safest point at which to conduct trades on Singularity is immediately after an oracle price update. This is because the price the oracle is reporting(aka the price the trade is executed at) is likely to be very close to the true price of the asset. As time passes, there will be some amount of volatility in the asset's true price that will not be reflected in the oracle until the next time the oracle will be updated.

Singularity needs a way to estimate how volatile an asset's price will be over time, and to calculate trading fees based off that implied volatility. One example might be DAI having an implied volatility of 0.1% per day. If the Chainlink oracle for DAI only updates once every 24 hours, then 23.99 hours after the previous price update, the base trading fees for the DAI pool must be *at least* 0.1%.

Much care needs to be taken in the design of such an oracle, and on its own, it will not be an adequate mitigation. Black Swan events are always possible, and such an event could easily cause more volatility than the most sophisticated oracle is expecting.

Obtain price update guarantees from Chainlink

Chainlink price oracle updates tend to be scheduled on regular intervals. If Chainlink's price oracle was to update when a certain amount of volatility is realized, then Singularity could set its base fee to a multiple of that volatility threshold.

However, this mitigation is not perfect either. If Chainlink guarantees price oracle updates at every 0.1% of price volatility, and 0.5% of price volatility is realized in a single block, an attacker could still execute this attack when Chainlink updates their price oracle.

Emergency pause mechanism

An emergency pause mechanism could be used to prevent trading through Singularity when there are on-chain conditions indicating that the next Chainlink update will be too volatile for the system's trading fees.

Developer response

Acknowledged, will fix via off-chain oracle

3. High - Incorrect tolerance for price reported by Chainlink Oracle (blockdev)

When the latest price is fetched from Chainlink Oracle, it is verified that it does not deviate beyond a certain tolerance threshold from the price reported before. SingularityOracle.sol#L21 defines this tolerance as 1.5% —

```
uint256 public maxPriceTolerance = 0.015 ether; // 1.5%
```

However, the way it's applied to calculate the deviation at SingularityOracle.sol#L43 is incorrect —

```
uint256 percentDiff = (priceDiff * 1 ether) / (price * 100);
```

This sets the tolerance threshold to 150% instead of 1.5%.

Proof of concept

The following code shows that a price difference more than 1.5% is accepted as a valid price when it should not be —

```
uint256 price = 1e8;
uint256 chainlinkPrice = 1.1e8;
uint256 maxPriceTolerance = 0.015 ether;

uint256 priceDiff = price > chainlinkPrice ? price - chainlinkPrice : chainlinkPrice - price;
uint256 percentDiff = (priceDiff * 1 ether) / (price * 100);
```

Here `chainlinkPrice` deviates by 10% from `price`. Following the way SingularityOracle.sol calculates the deviation, `percentDiff` turns out be less than `maxPriceToTolerance` which is not the intended result.

A full PoC can be found here: https://github.com/0xbok/singularity-v2/blob/21793993f634a385fb3ef18198a2761d732d91b5/foundry-test/Contract.t.sol#L305-L317

Impact

High. In situations where an asset price is too volatile, or chainlink oracle misbehaves, an incorrect price can be accepted leading to a loss of funds from pools.

Recommendation

Change the way `priceDiff` is calculated at SingularityOracle.sol#L43 to

```
uint256 percentDiff = (priceDiff * 1 ether) / price;
```

Developer response

Fixed in commit c26c70975cd12c51476de14999cec08f18320f3c

4. High - Frozen protocol fee when too many pools are deployed (blockdev)

A transaction containing a potentially unbounded loop can cause it to run out of gas. If that loop is reading and writing on storage, every iteration becomes expensive. Each block has a target gas limit, if a transaction is close to that limit or exceeds it, it becomes impossible to execute it. `SingularityFactory.sol`'s `collectFees()` function has this risk.

Proof of concept

SingularityFactory.sol#L111-L119 has a `collectFees()` function which collects the protocol fee from all the pools in a single transaction.

```
function collectFees() external override onlyAdmin {
    uint256 length = allPools.length;
    for (uint256 i; i < length; ) {
        ISingularityPool(allPools[i]).collectFees(feeTo);
        unchecked {
            ++i;
        }
    }
}
```

If too many pools are deployed, it will be impossible to collect fee due because the transaction will cost more than the block gas limit.

Impact

High. Protocol fee will be permanently frozen.

Recommendation

Rename `collectFees()` to `collectFeesForAll()`, and add a new function which takes in a list of pools to collect fee from.

```
function collectFees(address[] calldata tokens) external override onlyAdmin {
    for (uint256 i; i < tokens.length; ) {
        ISingularityPool(allPools[i]).collectFees(feeTo);
        unchecked {
            ++i;
        }
    }
}
```

Developer response

Fixed in commit 57541d07f6e43648e42de8f8ee3cd2f1be8a0d35

5. High - Planned functionality incentivizes low pool liquidity (uk)

The Singularity documentation states that when the `cRatio <= 1` that deposits are free. Similarly withdrawals incur no fee when `cRatio >= 1`. These changes have adverse effects that can lead to unplanned loss of fees and very low absolute liquidity. At certain values of `cRatio` large depositors are incentivized to take advantage of positive slippage and subsequently withdraw all of their funds from a pool. This change is currently only in the documentation and planned in the dev branch

Proof of concept

Consider a system with:

```
USDC Pool (assets: 950, liabilities: 1000)
USDT Pool (assets: 1000, liabilities: 1000)
```

A small swap from USDT to USDC (less than `51 USDC` worth after fees, due positive slippage) will cause the USDC pool to become overcollateralized. All depositors, whale or not, are incentivized to withdraw their assets from the USDC pool now as each withdrawal will be free. A single large withdrawal could cause the pool to have a very high `cRatio` and very low absolute liquidity.

Impact

- Loss of fees for protocol and LPs.
- Low liquidity across the protocol.

Withdrawals that should be charged large fees can be made free by doing swaps with much smaller fees. These withdrawals will cause pools to have low liquidity and high `cRatio`.

Recommendation

Add a sliding fee instead of setting it to 0 outright. Alternatively, continue to charge a fee for all withdrawals regardless of `cRatio`.

Developer response

Acknowledged

Guest auditors High findings

1. High - Deposit fees even below 100% CR & withdrawal fees above 100%

https://github.com/revenant-finance/singularity-v2/blob/a3cdbc5515374c9e1792b3cb94ff1b084a9a1361/contracts/SingularityPool.sol#L279

Impact

One pays deposit fees even if the CR is below 100% which does not match the documentation.
One also pays withdrawal fees even if the CR is above 100%.

> "When collateralization ratio before deposit <= 1: `depositFee = 0`" https://docs.revenantlabs.io/singularity/details/providing-liquidity/depositing
> "When collateralization ratio before withdrawal >= 1: withdrawalFee = 0`" https://docs.revenantlabs.io/singularity/details/providing-liquidity/withdrawing

Recommended Mitigation Steps

Don't charge fees on the excess amount that pushes the CR over/under the desired threshold.

> Note that the recent changes do not fix this issue correctly.

The deposited/withdrawn amount needs to be split up into:

- an `amountToPayFees` which is the amount where 100% CR is hit after depositing / withdrawing. Fees should be charged on this amount
- the remaining `amount - amountToPayFees` which one does not have to pay fees on anymore.

Otherwise, one can **no fees** by strategically depositing / withdrawing in parts. **Example:**

- Imagine current CR is 90%. User wants to withdraw 50% of the assets which would charge fees on their entire withdrawal amount.
- Instead, they first deposit an amount that brings the CR above 100% (no fees). Then they withdraw their original amount plus the newly deposited amount and skip the withdrawal fee.

Developer response

Fixed in commit 7042e4dc411e2d68a4a43bf2e56d2037274a075c

2. High - `_getG` is not continuous

https://github.com/revenant-finance/singularity-v2/blob/a3cdbc5515374c9e1792b3cb94ff1b084a9a1361/contracts/SingularityPool.sol#L442

Impact

The `_getG` function is not continuous and jumps at a CR of 30% from `g(0.3) = 0.00003 / (0.3^8) = 0.45724737` to `g(0.2999...) = 0.43 - 0.3 = 0.13`.
Note that `g(0.3) > g(0.2999)` but a correct `g` function is always decreasing.

Users that trade across this threshold of 30% will trigger a revert in this function as `g(0.2999) - g(0.3) < 0`.

Recommended Mitigation Steps

The function should be continuous everywhere and be a decreasing function.

Developer response

Fixed in commit ff71a174560ff12077b1ad48bfd964a97b9d097c

3. High - `maxPriceTolerance` does not work correctly

https://github.com/revenant-finance/singularity-v2/blob/a3cdbc5515374c9e1792b3cb94ff1b084a9a1361/contracts/SingularityOracle.sol#L43

Impact

The `percentDiff` is compared to `maxPriceTolerance` but it is additionally divided by `100` which makes the check pass even if the price difference is larger than `maxPriceTolerance`.

Recommended Mitigation Steps

Fix it and add a test for it.

```
+ uint256 percentDiff = (priceDiff * 1 ether) / (price * 100);
- uint256 percentDiff = (priceDiff * 1 ether) / price;
require(percentDiff <= maxPriceTolerance, "SingularityOracle: PRICE_DIFF_EXCEEDS_TOLERANCE");
```

Developer response

Fixed in commit c26c70975cd12c51476de14999cec08f18320f3c

4. High - Price updates can be sandwiched

Impact

It's possible to profit from oracle price updates by sandwiching them. The attacker's profit is the LPs' loss.

Proof of concept

See `bfs#7030`'s POC and writeup.

Recommended Mitigation Steps

Consider interpolating the current price to the new price over a short time frame to smooth out the price and make atomic sandwich attacks not profitable.

Developer response

Acknowledged, will fix via off-chain oracle

Medium Findings

Residents Medium Findings

1. Medium - Chainlink oracle may return stale data (engn33r)

In SingularityOracle.sol, `latestRoundData()` is used but there is no check to validate if the return value contains fresh data. Proper checks should be applied to prevent the use of stale price data.

Technical Details

Chainlink data from `latestRoundData()` is used in this line of code. The only check performed on the return data is a check if the price is zero.

```
(, int256 answer, , uint256 updatedAt, ) = IChainlinkFeed(chainlinkFeeds[token]).latestRoundData();
```

`latestRoundData()` can return stale price data. The contract using this data should properly check the price data is fresh. If it is the responsibility of other contracts integrating with SingularityOracle.sol to add these checks, this must be very clearly documented because forgetting these checks is a common mistake.

Impact

Medium. Incorrect pricing data could lead to loss of value depending on how this function is used.

Recommendation

Add more checks to prevent stale pricing data from being used.

```
(uint80 roundID, int256 answer, , uint256 updatedAt, uint80 answeredInRound) =
IChainlinkFeed(chainlinkFeeds[token]).latestRoundData();
require(answeredInRound >= roundID, "Stale price");
require(updatedAt != 0,"Round not complete");
require(answer > 0,"Chainlink answer reporting 0"); // this last check already exists in the SingularityOracle.sol
contract's getLatestRound() function
```

2. Medium - Fee-free `deposit()` and `withdraw()` can be gamed (engn33r)

The dev branch of the repository contains a change to make the fee for `deposit()` zero when the collateralization ratio of the pool is less than 1 and make the fee for `withdraw()` zero when the collateralization ratio is greater than 1. Removing the fees from these scenarios can allow a user to perform a roundabout series of actions that may reduce the fee they pay compared to a more direct action.

Technical Details

Consider the following scenario.

Bob wants to withdraw 500 from a stablecoin Singularity pool. The initial pool status is as follows

- assets = 950
- liabilities = 1000
- collateralization ratio = 0.95

Because the collateralization ratio is less than 1, Bob has to pay a fee to withdraw. There would be no withdrawal fee is the collateralization ratio is above 1. Bob may find a scenario where the fee to move the collateralization ratio above 1 is cheaper than paying the withdrawal fee. Bob may instead:

Step 1.

- Bob deposits 10,000 into the pool, which changes the pool status to:
- assets = 10,950
- liabilities = 11,000
- collateralization ratio = 0.995454545

Step 2.
Bob uses `swapIn()` to change 100 stablecoins, which changes the pool status to:

- assets = 11,050
- liabilities = 11,000
- collateralization ratio = 1.004545455

Step 3.
Bob withdraws 10,500 without any withdrawal fee because the collateralization ratio is above 1 now.

- assets = 550

- liabilities = 500
- collateralization ratio = 1.1

Impact

Medium. The mechanics changes from this commit may not provide the correct user incentives as intended.

Recommendation

Reconsider the mechanics and incentives of the system. I have no direct silver bullet for this, but considering the incentive design that other single-sided AMMs have chosen is one starting point.

3. Medium - Mechanics increase bank run risk (engn33r)

Multiple scenarios can arise where depositors in a pool cannot withdraw the value they originally deposited. This can lead to a bank run scenario which may cause loss of user value. The system may not be able to safely wind down.

An undercollateralized pool will not allow users to withdraw their assets. A `swapOut()` from an undercollateralized pool that brings the collateralization ratio to under 0.35 will already result in slippage of around 15-20%. Because `withdraw()` and `swapOut()` make a pool "unhealthier" as described in the protocol documentation, a bank run scenario can lead to user loss of value. This scenario could be caused by risks with the pool's token, external to any code in the Singularity protocol.

Technical Details

In a traditional LP like Uniswap, the LP token holder is at risk of impermanent loss. The user could lose value when the balance of the two tokens in the pool is different from when they deposited. However, this risk does not prevent a user from receiving some amount of value from their LP tokens. In Singularity v2, there are situations where the LP tokens cannot be redeemed, or at least not fully.

One scenario where a `withdraw()` action would lead to a transaction revert is when `amount > assets`. Because the `liabilities` variable tracks the amount of value that should be returned to users (including lpfees) but `assets` tracks the value that the pool currently holds, a scenario where a user wants to retrieve more than `assets` will result in a revert during the `safeTransfer` call. This can happen in an undercollateralized pool such as the following scenario.

Pool status before transaction 1

- assets = 10,000
- liabilities = 15,000
- collateralization ratio = 0.6666
- getG = 0.00076887

txn 1: User Alice withdraws 10,000 in value

Pool status before transaction 2

- assets = 0
- liabilities = 5,000
- collateralization ratio = 0
- getG = 0.43

txn 2: User Bob tries to withdraw 3,000 in value, but the transaction reverts when the token transfer finds there are insufficient tokens held by the pool.

Several issues with the protocol mechanics of the `deposit()` and `withdraw()` operations exist. These may worsen the case of an undercollateralized pool scenario:

1. The `withdraw()` operation moves the collateralization ratio away from 1. If the collateralization ratio is greater than 1, a `withdraw()` will cause the collateralization ratio to increase more. If the collateralization ratio is less than 1, a `withdraw()` will cause the collateralization ratio to decrease more.
2. When the collateralization ratio is below 0.3 and the linear portion of the piecewise defined `getG()` function is entered, the fee calculation in `getWithdrawalFee()` can result in `feeB > feeA` which permits a free withdrawal. For example, when the collateralization ratio is 0.1, removing all remaining assets will result in a zero withdrawal fee. This incentives the removal of the remaining assets value in a pool leading to a bank run scenario that prevents additional withdrawals.
3. The `deposit()` function applies a fee to deposits regardless of the collateralization ratio. `deposit()` moves the collateralization ratio value closer to 1, making the pool healthier, so a deposit into an undercollateralized pool should be incentivized with zero fee (or even negative fees).

Because an undercollateralized pool can remain undercollateralized for extended periods of time, the pools in Singularity may all trend towards an undercollateralized state. This wil leave the protocol with bad debt, and with no solution to resolve this debt, the entire protocol may converge towards a bank run scenario.

Impact

Medium. Protocol mechanics can lead to a bank run in unhealthy pools where users lose value, similar to Luna, though the exact circumstances that would happen to cause this were not pinned down precisely.

Recommendation

Because the pool mechanics permit pool undercollateralization for extended periods of time, there may be no way for users to retrieve pool value with the existing protocol mechanics. Several changes could help this, but do not fully solve the problem:

1. If the `assets` value is considered to accurately reflect the value held by the pool, remove the logic from this line so that `_assets - amount` reverts earlier if `amount > _assets` to save the user some gas.
2. Remove fees in `deposit()` when the collateralization ratio of the pool is under 1.
3. Consider changing the feeA and feeB calculation in `getWithdrawalFee()` to prevent fee-free withdrawals in undercollateralized pools.
4. Set up monitoring to track pool health on public deployments and create plans to improve pool health if the collateralization ratio of certain pools does not return towards 1 within a certain timeframe.

Developer response

Acknowledged, will fix off-chain

4. Medium - Oracle price lag enables 1% arbitrage opportunities (engn33r)

The Chainlink oracle deviation threshold value for several major USD cryptocurrencies pairs is 0.5%. In highly volatile market conditions, this can enable a 1% arbitrage profit opportunity when one token is near the bottom of its deviation threshold and another is near the top of its deviation threshold. The result is that the Singularity pools will allow a swap of these tokens with an outdated price that favors the user at the cost of the LP holders, potentially resulting in a slow loss of value over time.

Technical Details

A Singularity swap operation trusts the Chainlink oracle price to swap between two Singularity pools using `getAmountToUSD()` and `getUSDToAmount()`. Take a scenario where Chainlink says the ETH price is $2000 and the BTC price is $30000. The exchange rate between ETH and BTC should be around 15 ETH to 1 BTC. Because the deviation threshold value is 0.5%, it is possible for the ETH price to drop to $1990 and the BTC price to rise to $30150. This price change moves the exchange rate to about 15.15 ETH to 1 BTC, or about a 1% change. This means an arbitrageur could `swapIn()` 15 ETH to receive 1 BTC in Singularity v2 and receive a 1% profit by trading the 1 BTC for 15.15 ETH using the more updated market price. The exact toll this takes on a pool in the long term, when consider lpfees accruing to LP token holders, is not clear without more detailed modeling of market conditions. However there can be cases where the pool loses value by trusting the Chainlink oracle prices as "correct" if:

1. Market conditions are highly volatile with assets in Singularity pools moving in opposite directions
2. Swap fees are low enough
3. Slippage favors the user (see the separate arbitrage issue about this situation)

Impact

Medium. Arbitrageurs can wait for specific market conditions to extract value from pools, although the market conditions that make this profitable may be rare.

Recommendation

Protocol fees should be high enough to disincentivize arbitrage of this kind or compensate LP holders sufficiently to protect against loss of value. Additional pricing sources can provide more accurate prices for high volatility assets.

Developer response

Acknowledged, will fix via off-chain oracle

Fellows Medium findings

1. Medium - Slippage calculations are not path independent for large trades (blockdev, Benjamin Samuels)

The slippage calculation curve used in `getSlippageOut()` and `getSlippageIn()` is not path independent for excessively large slippage values; this means that under certain collateralization conditions, users can optimize their slippage fees by splitting their trade into multiple smaller swaps.

This issue was only found to be exploitable under the following conditions:

1. The SwapIn pool is overcollateralized.
2. The SwapOut pool is undercollateralized.
3. The amount of tokens swapped in/out are a large fraction of each pool's assets.

Depending on the collateralization conditions of each pool, users may be incentivized to split their trade into many smaller trades, or to group their trades with others' to obtain favorable slippage conditions.

Proof of concept

Assume there are two singularity pools deployed, one for USDC and one for USDT. Each pool has the following properties:

- USDC Pool Liabilities: 1 USDC
- USDC Pool Assets: 1.51058 USDC
- USDT Pool Liabilities: 1 USDT

- USDT Pool Assets: 0.48942 USDT

The trader wants to transfer 0.378 USDC to USDT. The output of a single monolithic trade vs. a batch of two trades is compared as follows:

Monolithic trade

```
Amount swapped in:   0.37800 USDC
Amount received out: 0.07642 USDT
Slippage in:  0.00000 USDC
Slippage out: 0.30243 USDT
```

Split trade

```
Amount swapped in during first swap:  0.36678 USDC
Amount swapped in during second swap: 0.01216 USDC
Total amount received out:            0.08075 USDT
Total Slippage in:  0.00000 USDC
Total Slippage out: 0.29810 USDT
```

Splitting the trade into two reduce the output slippage fees by 0.00432 USDT, improving the execution price by 5.5% compared to the single swap.

It's important to note that in this proof of concept, the vast majority of the trade is eaten away by slippage.

A hardhat script for this PoC is available here.

Impact

During fuzzing, it was found that trade sizes had to be single to double digit percentages of the pool's total liquidity before there were meaningful benefits to splitting trades. Trades of this size lose much of their value to slippage, so it is likely that any attempt to exploit this vulnerability would only occur during the hack of another protocol, or some other set of liquidity-tight on-chain conditions.

Unfortunately in the interest of time, the boundary conditions of this vulnerability were not able to be tested. It may be possible to exploit this vulnerability using smaller trades or to create much larger slippage discounts by breaking up swaps in different ways.

At time of writing, it is believed that this vulnerability does not put the protocol at risk of loss of funds. However, if an attacker is able to find a set of inputs for which slippage fees can be drastically reduced, many of the cryptoeconomic assumptions the protocol makes about slippage may fall apart; allowing funds to be drained by swapping back and forth between two pools using ideal swap parameters.

Recommendation

This vulnerability appears to be related to fixed point aliasing caused by the `getG()` function. The `getG()` function involves raising an input number to the 8th power which suggests a large dynamic range for possible inputs and outputs. Fixed point numbers are notorious for being unable to accommodate accurate representation of large ranges of values; it is likely that `getG()` is losing a relatively large amount of accuracy during the rpow operation.

The following recommendations should increase the accuracy of `getG()`:

- Replace `rpow` with a different function in `getG()`; there may be other functions with similar shape that perform with better accuracy properties in a fixed point system.
- Use different parameters for `rpow`. There may be higher or lower power values that provide improved accuracy properties in a fixed point system.
- Use higher precision fixed point numbers. Some Solidity libraries such as DecimalMath offer 27 decimals for fixed point numbers.

2. Medium - Reentrancy in `SingularityPool.collectFees()` (Benjamin Samuels)

The `SingularityPool.collectFees()` function is vulnerable to a reentrancy attack via `safeTransfer()`. The `collectFees()` implementation is as follows:

```
function collectFees(address feeTo) external override onlyFactory {
    if (protocolFees == 0) return;

    IERC20(token).safeTransfer(feeTo, protocolFees);
    protocolFees = 0;

    emit CollectFees(protocolFees);
}
```

This function does not comply with the checks-effects-interactions design pattern.

Proof of concept

1. A Singularity pool is added for an ERC-777 compliant token.
2. Some amount of trading fees is accrued by the pool.
3. Admin calls `collectFees()` using a `feeTo` address that has implemented a `tokensReceived` hook using ERC-1820.
4. The `feeTo` contract calls `collectFees()` again, effectively doubling the amount of fees collected on behalf of feeTo.

5. The reentrancy attack is repeated until the pool is entirely drained of the ERC-777 compliant token.

Impact

This vulnerability allows the protocol admin to drain a pool of all funds if the pool's underlying token can trigger a re-entrancy using `safeTransfer()`. This is possible with ERC-777 compliant tokens, and with some ERC-20 tokens. The CREAM hack in particular involved an ERC-20 token that offered a re-entrancy vector through `transfer`.

Recommendation

Always use the checks-effects-interactions pattern. In this specific instance, this vulnerabilitiy is remediated by storing the fees in a temp variable and zeroing out protocolFees before transferring:

```
function collectFees(address feeTo) external override onlyFactory {
    // checks
    if (protocolFees == 0) return;

    // effects
    uint256 protocolFeesToTransfer = protocolFees;
    protocolFees = 0;

    // interactions
    IERC20(token).safeTransfer(feeTo, protocolFeesToTransfer);

    emit CollectFees(protocolFeesToTransfer);
}
```

In addition, extreme caution should be taken by the protocol admin when allowlisting tokens and creating new pools.

Developer response

Fixed in commit 5dea908b44af0f02b20a49e5545a4b8e02fe1eeb

Guest auditors Medium findings

1. Medium - It can be profitable to split up orders

Impact

It can be profitable to split up orders into smaller ones. Users can avoid paying high slippage fees this way, essentially circumventing the protocol's real fee structure and leading to undesired behavior of having to do multiple swaps.

Proof of concept

two pools with assets=liabilities = 1000.0 USDC, 1000.0 DAI (`baseFee = 0.0004`)
single swap: swap(650.0 DAI) = 517.0 USDC
two swaps: swap(604.0 DAI) + swap(46.0 DAI) = 573.0 USDC

That's a `10.83%` profit by splitting it up into two trades.
The percentage gain from splitting up swaps is largest in high-slippage (low CR) areas.

See the POC here.

Recommended Mitigation Steps

This issue is because of the dynamic fee design and it's hard to fully mitigate unless a complete overhaul of this mechanism is done.

2. Medium - `_getG` exponentiation can overflow for large CR values

https://github.com/revenant-finance/singularity-v2/blob/a3cdbc5515374c9e1792b3cb94ff1b084a9a1361/contracts/SingularityPool.sol#L446

Impact

For a CR of 14,000,000% (`140_000e18`) the `collateralizationRatio.rpow(8, 1 ether);` function overflows and reverts.

Proof of concept

An attacker can break pools if the pool currently has low liquidity. Imagine they are the first to deposit into the `DAI` pool.
They `deposit(1e6)` ("10^-12 DAI").
Then they swap a bit less than `140_000 * 1e6 = 14e10` DAI from DAI to some other pool.
The pool's CR is close to the overflow value `140_000e18` after the swap.
Any other users trying to swap into the pool will have their transactions reverted.

Recommended Mitigation Steps

At such high CR ratios, the `g(CR)` function is close to zero. Don't do this computation if it would overflow and directly return 0 instead.

Low Findings

Residents Low Findings

1. Low - Todo comment indicates necessary change (engn33r)

In SingularityOracle.sol, a comment exists stating `change to _updatedAt in prod`. This change should be made prior to production release, but the change was not made prior to publishing the existing code to Fantom.

Technical Details

There is a line of code that includes a comment that `block.timestamp` should be changed to `_updatedAt` for production. This change was not made to the contract deployment on Fantom.

```
return (chainlinkPrice, block.timestamp); // change to _updatedAt in prod
```

Impact

Low. Change should happen before prod release.

Recommendation

Change `block.timestamp` to `_updatedAt` for production.

2. Low - Denial of service edge case if `onlyUseChainlink` is false (engn33r)

The pricing solution that relies on the `allPrices` mapping in SingularityOracle.sol is not fully implemented, but if a condition arises where the pricing data delta between Chainlink and this custom solution is over 1.5%, the protocol will be unusable. Users will not be able to `swapIn()` or `swapOut()` from a pool in such conditions.

Technical Details

This pricing tolerance check compares the custom pricing value to the Chainlink price. If the prices vary by a value greater than the tolerance, users will not be able to `swapIn()` or `swapOut()` from the protocol because those operations call `getSlippageIn()/getSlippageOut()` and `getTradingFees()` which call `getLatestRound()` in SingularityOracle.sol. The difference from a paused pool is that the admin controls whether a pool is paused while the `maxPriceTolerance` check is not directly controlled by an admin and may happen without warning, depending on the custom oracle solution.

Impact

Low. The result is that the `swapIn()` and `swapOut()` pool actions are effectively paused until the pricing data passes the `maxPriceTolerance` check.

Recommendation

The pricing solution that relies on the `allPrices` mapping should be implemented in a way that it can be trusted on its own. Ideally the price data should not be stored on-chain because this will require substantial cost to store this data. Instead, an off-chain alternative to Chainlink data should perform the 1.5% delta logic check offline and handle the case in such a way that it does not create a complete denial of service condition immediately. The off-chain logic check could result in automatic pausing of the pool if the pricing delta persists or grows, but the current design effectively pauses the pool immediately when this condition arises.

3. Low - No pool existence check (engn33r)

When SingularityRouter.sol performs a lookup with `poolFor()` to calculate the address where the pool is located, there is no check for whether the pool exists before interacting with this address. It is unlikely that another contract with the same function arguments would exist at this address, but if a malicious contract was located at this address, users could lose value.

Technical Details

Uniswap's `_addLiquidity()` function checks whether a pair exists. There is no similar check in SingularityRouter.sol. This is one example of interacting with the address provided by `poolFor()` without verifying the pool exists, but there are many other instances of this happening in the contracts. A check for whether a pool exists is only done once in SingularityFactory.sol.

Impact

Low. There is a low likelihood of calls to the calculated pool address completing without a revert if the Singularity Factory did not create the pool, but the cases where this happens would lead to loss of user value.

Recommendation

Follow the same approach as Uniswap and check whether a pool exists before interacting with the calculated pool address. While it might be acceptable for the frontend web app to perform these checks, adding a check in the smart contract to validate the pool exists before interacting with it would improve protocol security.

Developer response

Acknowledged

4. Low - Missing zero address check (engn33r)

The `_swap()` function in SingularityRouter.sol is forked from the Uniswap's Router implementation. The logic in `_swap()` attempts to duplicate the logic from Uniswap's implementation, but a zero address check in Uniswap's implementation is missing in the SingularityRouter.sol implementation.

Technical Details

The `_swap()` function checks for the edge case of `tokenIn == tokenOut`, which is borrowed from the check that Uniswap performs in `sortTokens()`. But the other check in Uniswap's `sortTokens()`, the zero address check, is not implemented in SingularityRouter.sol.

Impact

Low. The logic from the reference implementation has not been fully borrowed, which may lead to unexpected edge cases.

Recommendation

Add a zero address check near the start of the `_swap()` function in SingularityRouter.sol.

Developer response

Fixed in commit 808dc21cc701def2d307892e78b0a11720281cef

5. Low - g function discontinuity at 0.3 (engn33r)

The g function is calculated in `_getG()`. The function is piecewise-defined. There is a discontinuity at the point 0.3, where there is a switch from `g = 0.43 - x` to `g = 0.00003 / (x^8)`. In the first function, `g(0.3) = 0.13` while in the second function `g(0.3) = 0.45725`. In the second function, `g(0.316228) = 0.3`. This discontinuity means that users are more incentivized to move the collateralization ratio below 0.3 than to keep the collateralization ratio between 0.3 and 0.316228.

Technical Details

There is a discontinuity in the piecewise-defined function in `_getG()`.

Impact

Low. The incentives are designed to keep the collateralization ratio away from this range, but entering this range may cause unexpected pricing edge cases.

Recommendation

Modify the g function to avoid discontinuities.

Developer response

Fixed in commit ff71a174560ff12077b1ad48bfd964a97b9d097c

Fellows Low findings

1. Low - Missing two-step transfer ownership pattern (blockdev, SaharAP)

SingularityFactory.sol and SingularityOracle.sol has several functions which can only be called by its `admin`. The `admin` is first set via its contructor. The current admin can set another address as `admin` by calling `setAdmin()` function. If an inaccessible address is passed to this function, all the privileged functionality will be permanently lost.

Proof of concept

SingularityFactory.sol#L81 and SingularityOracle.sol use `setAdmin()` to set `admin` to a new address —

```
function setAdmin(address _admin) external override onlyAdmin {
    require(_admin != address(0), "SingularityFactory: ZERO_ADDRESS");
    admin = _admin;
}
```

If `setAdmin(0x0000000000000000000000000000000000000001)` is called by the current admin, all the functions guarded by `onlyAdmin` modifier are permanently locked.

Impact

Low. In case, the caller is not careful, an incorrect argument can be passed leading to a permanent loss of admin control on `SingularityFactory.sol`. Privileged functionalities like creating new pools or collecting fee will be lost.

Recommendation

Follow two-step transfer ownership pattern. The current admin `a` first proposes a new address `b` for `admin`. Now, `b` accepts this proposal which then sets `admin` as `a`. Until then, `a` continues to be `admin`. You can use BoringOwnable.sol as a reference.

2. Low - Division before multiplication (SaharAP)

Performing multiplication before division is generally better to avoid loss of precision because Solidity integer division might truncate.

Proof of concept

In the following code in `getSlippageIn` function there is a division before multiplication for computing `slippageIn`

```
// Calculate G'
uint256 gDiff = _getG(currentCollateralizationRatio) - _getG(afterCollateralizationRatio);
uint256 gPrime = gDiff.divWadDown(afterCollateralizationRatio - currentCollateralizationRatio);
// Calculate slippage
slippageIn = amount.mulWadDown(gPrime);
```

First, `amount` should be multiplied by gDiff and then divided by `(afterCollateralizationRatio - currentCollateralizationRatio)`. `getSlippageOut` function also has this problem for computing `slippageOut`.

Impact

Low. In some cases division before multiplication can lead to a loss of precision of the final result.

Recommendation

Always use multiplication before division.

Developer response

Acknowledged

Guest auditors Low Findings

1. Low - Deposit cap is inaccurate

Impact

The `deposit` function checks if the new liabilities would exceed the `depositCap`.
However, it uses the pre-fee `amount`, whereas the liabilities only increase by the `amountPostfee`.

Recommended Mitigation Steps

Fix the deposit check.

Developer response

Fixed in commit 7042e4dc411e2d68a4a43bf2e56d2037274a075c

Gas Savings Findings

Residents Gas Savings Findings

1. Gas - Use unchecked in SingularityPool.sol (engn33r)

In SingularityPool.sol, an unchecked clause can be added because there is no risk of overflow or underflow.

Technical Details

No subtraction underflow is possible in this line and the similar `deposit()` function. The revised code would be:

```
unchecked { fee = feeA - feeB; }
```

Impact

Gas savings

Recommendation

Use unchecked when no risk of overflow or underflow exists.

Developer response

Fixed in commit 808dc21cc701def2d307892e78b0a11720281cef

2. Gas - Replace modifiers with internal functions (engn33r)

Internal functions use less gas than modifiers because the code is not inlined.

Technical Details

There are three modifiers in SingularityPool.sol and one each in SingularityFactory.sol and SingularityRouter.sol.

Impact

Gas savings

Recommendation

Replace modifiers with internal functions.

Developer response

Fixed in commit fd145537aaa6702c8d0f77e0b6a0b17695091a2c

3. Gas - Payable functions can save gas (engn33r)

If there is no risk of a function accidentally receiving ether, such as a function with the `onlyAdmin`, `onlyFactory`, or `onlyRouter` modifiers, this function can be payable to save gas.

Technical Details

The following functions have access control modifiers and can be marked as payable

- First instance
- Second instance
- Third instance

Impact

Gas savings

Recommendation

Mark functions that have access control modifiers like onlyAdmin as payable for gas savings. This might not be aesthetically pleasing, but it works.

Developer response

Acknowledged

4. Gas - Use short require strings (engn33r)

Reason strings for a require check takes at least 32 bytes. Using a reason strings over 32 bytes (characters) will increase gas consumption.

Technical Details

While there are many instances of this, here and here are two representative examples of the issue.

Impact

Gas savings

Recommendation

Replace modifiers with internal functions.

Developer response

Acknowledged

5. Gas - Use Solidity errors in 0.8.4+ (engn33r)

Using solidity errors is a new and more gas efficient way to revert on failure states

- https://blog.soliditylang.org/2021/04/21/custom-errors/
- https://twitter.com/PatrickAlphaC/status/1505197417884528640

Technical Details

Require statements are used throughout the contracts and error messages are not used anywhere. Using this new solidity feature can provide gas savings on revert conditions.

Impact

Gas savings

Recommendation

Add errors to replace each `require()` with `revert errorName()` for greater gas efficiency.

Developer response

Acknowledged

6. Gas - Using simple comparison (engn33r)

Using a compound comparison such as ≥ or ≤ uses more gas than a simple comparison check like >, <, or ==. Compound comparison operators can be replaced with simple ones for gas savings.

Technical Details

The `getAmountToUSD()` function in SingularityPool.sol contains:

```
        if (decimals <= 18) {
            value *= 10**(18 - decimals);
        } else {
            value /= 10**(decimals - 18);
        }
```

By switching around the if/else clauses, we can replace the compound operator with a simple one

```
        if (decimals > 18) {
            value /= 10**(decimals - 18);
        } else {
            value *= 10**(18 - decimals);
        }
```

Another case of this is in `getTradingFeeRate()`, where the most common case is currently the last else statement, but can be moved up the chain of logic by replacing a compound operator with a simple operator. The modified code would be:

```
if (timeSinceUpdate * 10 > oracleSens * 11) {
    tradingFeeRate = type(uint256).max; // Revert later to allow viewability
} else if (timeSinceUpdate < oracleSens) {
    tradingFeeRate = baseFee + (baseFee * timeSinceUpdate) / oracleSens;
} else {
    tradingFeeRate = baseFee * 2;
}
```

There are likely other instances of this gas savings in the code (see here and here), but only one example was highlighted for illustrative purposes.

Impact

Gas savings

Recommendation

Replace compound comparison operators with simple ones for gas savings.

Developer response

Fixed in commit c26c70975cd12c51476de14999cec08f18320f3c

7. Gas - Using Yul `iszero()` (engn33r)

Yul has an `iszero()` operation which can be used when comparing a value to zero to save gas.

Technical Details

There are many comparisons to zero in the code, including here and here.

Impact

Gas savings

Recommendation

Use Yul `iszero()` for gas savings.

Developer response

Acknowledged

8. Gas - Replace bool with uint256 (engn33r)

A bool type uses more gas than toggling a uint256 between 0 and 1.

Technical Details

There are two bools (1, 2) in SingularityFactory.sol, three bools (1, 2, 3) in SingularityPool.sol, and three bools (1, 2, 3) in SingularityRouter.sol.

Impact

Gas savings

Recommendation

Replace all bool variables with uint256 variables.

Developer response

Acknowledged

9. Gas - Declare immutable variables internal when possible (engn33r)

Declaring immutable variables with internal visibility is cheaper than public immutables. Some of these immutable variables may need to be public, but others may not.

Technical Details

SingularityPool.sol and SingularityRouter.sol have public immutable variables.

Impact

Gas savings

Recommendation

Make immutable variables internal for gas savings.

Developer response

Fixed in commit 581cde77e655e66f9cb2f4c20c453cd5b91e4145

10. Gas - Redundant function (engn33r)

The `getAssetsAndLiabilities()` function returns the value of two state variables. These state variables are public and therefore already have built-in getter functions, making this function unnecessary.

Technical Details

The redundant function can be replaced by the getter functions for external calls and by using the state variables directly for internal calls.

Impact

Gas savings

Recommendation

Remove `getAssetsAndLiabilities()` and use the built-in getter functions or the state variables directly for `assets` and `liabilities`.

Developer response

Fixed in commit 808dc21cc701def2d307892e78b0a11720281cef

11. Gas - Remove nonReentrant modifiers (engn33r)

The `withdraw()` and `swapOut()` functions can remove the nonReentrant modifier if the `token.safeTransfer()` call is the last line of the function, immediately before the emit call. This change would allow the functions to follow the checks-effects-interactions pattern.

Technical Details

The `withdraw()` and `swapOut()` functions do not need the nonReentrant modifier if they follow the checks-effects-interactions pattern.

The `withdraw()` function in SingularityPool.sol has the following code:

```
        IERC20(token).safeTransfer(to, withdrawalAmount);

        // Update assets and liabilities
        assets -= amount;
        liabilities -= amount;
```

The code can be modified to modify the state variables before performing the token transfer to follow the checks-effects-interactions best practices pattern.

```
        // Update assets and liabilities
        assets -= amount;
        liabilities -= amount;

        IERC20(token).safeTransfer(to, withdrawalAmount);
```

Impact

Gas savings

Recommendation

Remove the nonReentrant modifier for gas savings if functions follow the checks-effects-interactions pattern.

Developer response

Fixed in commit 808dc21cc701def2d307892e78b0a11720281cef

12. Gas - Remove `feeA > feeB` test in `getDepositFee()` (engn33r)

In `getDepositFee()`, the case `feeA < feeB` will never occur. This allows the if/else statement comparing `feeA` and `feeB` can be removed.

Technical Details

`getDepositFee()` in SingularityPool.sol has the following code:

```
if (feeA > feeB) {
    fee = feeA - feeB;
    require(fee < amount, "SingularityPool: FEE_EXCEEDS_AMOUNT");
} else {
    fee = 0;
}
```

The case `feeA < feeB` cannot occur. This was confirmed in four basic cases:

1. Large collateralization ratio and very small deposit amount
2. Large collateralization ratio and very large deposit amount
3. Small collateralization ratio and very small deposit amount
4. Small collateralization ratio and very large deposit amount

The logic can be simplified to:

```
fee = feeA - feeB;
require(fee < amount, "SingularityPool: FEE_EXCEEDS_AMOUNT");
```

Impact

Gas savings

Recommendation

Use the suggested code change to increase gas savings.

Developer response

Fixed in commit fd145537aaa6702c8d0f77e0b6a0b17695091a2c

Fellows Gas Savings Findings

1. Gas - Collateral ratio calculations have redundant SLOADs (Benjamin Samuels)

In order to calculate current and future collateralization ratios, the `getSlippageIn()` and `getSlippageOut()` functions perform three redundant SLOADs, one on `SingularityPool.assets`, and two on `SingularityPool.liabilities`. These SLOADs occur in the `getSlippage()` functions and in the `getCollateralizationRatio()` function.

This pattern extends into the `getWithdrawalFee()` and `getDepositFee()` functions as well.

Proof of concept

n/a?

Impact

- 1360 gas savings on average in `SingularityRouter.swapExactETHForTokens()`
- 1360 gas savings on average in `SingularityRouter.swapExactTokensForETH()`
- 1360 gas savings on average in `SingularityRouter.swapExactTokensForTokens()`
- 337 gas savings on average in `SingularityPool.withdraw()`
- 43 gas savings on average in `SingularityPool.addLiquidity()`
- 202 gas savings on average in `SingularityPool.removeLiquidity()`

Recommendation

Only access `SingularityPool.assets` and `SingularityPool.liabilities` once during each swap. Keep them in memory, and if their values are needed later, use the values from memory instead of calling SLOAD again.

These changes are implemented in the following patch:

```
diff --git a/contracts/SingularityPool.sol b/contracts/SingularityPool.sol
index 1de798d..47ebcfe 100644
--- a/contracts/SingularityPool.sol
+++ b/contracts/SingularityPool.sol
@@ -223,10 +223,14 @@ contract SingularityPool is ISingularityPool, SingularityPoolToken, ReentrancyGu
     /// @dev Collateralization ratio is 1 if pool not seeded
     /// @return collateralizationRatio The collateralization ratio of the pool
     function getCollateralizationRatio() public view override returns (uint256 collateralizationRatio) {
-        if (liabilities == 0) {
+        (uint256 _assets, uint256 _liabilities) = getAssetsAndLiabilities();
+        return getCollateralizationRatio(_assets, _liabilities);
+    }
+
+    function getCollateralizationRatio(uint256 _assets, uint256 _liabilities) internal pure returns (uint256
collateralizationRatio) {
+        if (_liabilities == 0) {
             collateralizationRatio = 1 ether;
         } else {
-            (uint256 _assets, uint256 _liabilities) = getAssetsAndLiabilities();
             collateralizationRatio = _assets.divWadDown(_liabilities);
         }
     }
@@ -278,9 +282,9 @@ contract SingularityPool is ISingularityPool, SingularityPoolToken, ReentrancyGu
     function getDepositFee(uint256 amount) public view override returns (uint256 fee) {
         if (amount == 0 || liabilities == 0) return 0;

-        uint256 currentCollateralizationRatio = getCollateralizationRatio();
-        uint256 gCurrent = _getG(currentCollateralizationRatio);
         (uint256 _assets, uint256 _liabilities) = getAssetsAndLiabilities();
+        uint256 currentCollateralizationRatio = getCollateralizationRatio(_assets, _liabilities);
+        uint256 gCurrent = _getG(currentCollateralizationRatio);
         uint256 afterCollateralizationRatio = _calcCollatalizationRatio(_assets + amount, _liabilities + amount);
         uint256 gAfter = _getG(afterCollateralizationRatio);

@@ -311,7 +315,7 @@ contract SingularityPool is ISingularityPool, SingularityPoolToken, ReentrancyGu

         (uint256 _assets, uint256 _liabilities) = getAssetsAndLiabilities();
```

```
-        uint256 currentCollateralizationRatio = getCollateralizationRatio();
+        uint256 currentCollateralizationRatio = getCollateralizationRatio(_assets, _liabilities);
         uint256 gCurrent = _getG(currentCollateralizationRatio);
         uint256 afterCollateralizationRatio = _calcCollatalizationRatio(
             _assets > amount ? _assets - amount : 0,
@@ -344,8 +348,8 @@ contract SingularityPool is ISingularityPool, SingularityPoolToken, ReentrancyGu
     function getSlippageIn(uint256 amount) public view override returns (uint256 slippageIn) {
         if (amount == 0) return 0;

-        uint256 currentCollateralizationRatio = getCollateralizationRatio();
         (uint256 _assets, uint256 _liabilities) = getAssetsAndLiabilities();
+        uint256 currentCollateralizationRatio = getCollateralizationRatio(_assets, _liabilities);
         uint256 afterCollateralizationRatio = _calcCollatalizationRatio(_assets + amount, _liabilities);
         if (currentCollateralizationRatio == afterCollateralizationRatio) {
             return 0;
@@ -366,8 +370,8 @@ contract SingularityPool is ISingularityPool, SingularityPoolToken, ReentrancyGu
         if (amount == 0) return 0;
         require(amount < assets, "SingularityPool: AMOUNT_EXCEEDS_ASSETS");

-        uint256 currentCollateralizationRatio = getCollateralizationRatio();
         (uint256 _assets, uint256 _liabilities) = getAssetsAndLiabilities();
+        uint256 currentCollateralizationRatio = getCollateralizationRatio(_assets, _liabilities);
         uint256 afterCollateralizationRatio = _calcCollatalizationRatio(_assets - amount, _liabilities);
         if (currentCollateralizationRatio == afterCollateralizationRatio) {
             return 0;
```

Developer response

Acknowledged

2. Gas - Replace ProtocolFees state var with Slippage (devtooligan)

Currently protocol fees are explicitly tracked via the `protocolFees` state variable. This var is updated during `deposit()`, `withdraw()`, `swapIn()` and `swapOut()` at an average cost of around ~5,000 gas for SSTORE when changing a non-zero value to another non-zero value.

Slippage is not explicitly stored but can be calculated as: `token.balanceOf(pool) - protocolFees - assets`. Slippage only occurs in `swapIn()` and `swapOut()`.

If it is switched so that `slippage` is explicitly stored as a state var, and protocolFees are calculated, then that would eliminate the need to update `protocolFees` in four functions, and only incur the cost of updating `slippage` in two functions.

Impact

Save ~5,000gas for `deposit()` and `withdraw()`

Recommendation

Remove the `protocolFees` var and track `slippage` instead. This will not only save gas, but has the added benefit of being able to retrieve tokens that were inadvertently transferred to the pool. They would be considered protocol fees and could be retrieved along with the protocol fees with `collectFees()`. If additional tracking of protocol fees is needed, recommend using events to broadcast fee increases and storing this offchain.

3. Gas - Tight variable packing (devtooligan)

The storage variables in SingularityPool.sol are all 256 bits but they don't need to be. Storing `assets` and `liabilities` as a uint128 would allow for retrieving both of them from the same slot thereby saving a cold SLOAD (2100gas) for all transactional core functions. This would involve adding some logic to cast/recast, but it would also enforce caching the vars in memory which would result in additional savings.

Beyond assets and liabilities, other state vars could be recast to fit together in a single slot as well. For example, 18 digits of precision is probably not needed for `depositCap`, 0 digits are probably fine. This would allow reducing it to a uint32() for a ~4.3b max. Other state vars could also be downcast including `basefee`, `protocolFees` and even `isPaused`. All 4 of these could comfortably fit into a single storage slot which would save up to 3 cold SLOAD's (~6,300 gas) in some functions.

Proof of concept

Here is the top half of `deposit()` refactored for use with uint128 assets / uint128 liabilities:

```
   function deposit(uint256 amount, address to) external override notPaused nonReentrant returns (uint256
mintAmount) {
       // Because assets/liabs are changed to u128 and packed into 1 slot, this is 1 SLOAD.
       // Assigned to uint256 memory vars (cast up to u256 to allow for easy interaction with other uint256
amounts)
       (uint256 assets_, uint256 liabilities_) = (assets, liabilities); //                      LINE ADDED
```

```
        require(amount != 0, "SingularityPool: AMOUNT_IS_0");
        require(amount + liabilities_ <= depositCap, "SingularityPool: DEPOSIT_EXCEEDS_CAP"); // LINE UPDATED

        // Transfer token from sender
        IERC20(token).safeTransferFrom(msg.sender, address(this), amount);

        if (liabilities_ == 0) {
            mintAmount = amount;

            // Mint LP tokens to `to`
            _mint(to, mintAmount);

            // Update assets and liabilities
            assets = (assets_ + amount).u128();  // safeCast to u128                             LINE UPDATED
            liabilities = (liabilities_ + amount).u128(); // safeCast to u128                    LINE UPDATED
        } else {
```

Impact

~2100+ gas for `deposit()`, `withdraw()`, `swapIn`, `swapOut`, `getCollateralizationRatio()`, `getDepositFee()`, `getWithdrawalFee()`, `getSlippageIn()`, `getSlippageOut`

Recommendation

At a minimum, pack assets and liabilities into 1 slot by casting them as uint128. Recommend using a SafeCast library like Solmate SafeCastLib.sol. For additional savings, consider the storage layout of other slots and other contracts.

Developer response

Acknowledged

4. Gas - Use `unchecked` when there is no risk of overflow or underflow (blockdev)

Since Solidity v0.8, arithmetic operations revert on underflow and overflow. If there is no such possibility, these operatiosn can be wrapped in an `unchecked` clause to save gas.

Proof of concept

Instances where `unchecked` can be used —

- SingularityPool.sol#L185

  ```
  uint256 amountPostSlippage = amount - slippage;
  ```

- SingularyPool.sol#L252-L256

  ```
  if (decimals <= 18) {
      value *= 10**(18 - decimals);
  } else {
      value /= 10**(decimals - 18);
  }
  ```

- SingularityPool.sol#L300 and SingularityPool.sol#L335

  ```
  fee = feeA - feeB;
  ```

Impact

Gas savings

Recommendation

Use `unchecked` when when no risk of overflow or underflow exists.

Developer response

Acknowledged

5. Gas - Do not load calldata length in memory for loops (blockdev)

Reading calldata is cheaper than reading from memory. So, to iterate on a calldata array, we can just use its length already stored in calldata.

Proof of concept

- SingularityFactory.sol#L123-L124, SingularityFactory.sol#L136-L137, SingularityFactory.sol#L150-L151

  ```
  uint256 length = tokens.length;
  for (uint256 i; i < length; ) {
  ```

Impact

Gas savings

Recommendation

Use `tokens.length` directly for iteration instead of using `length`.

Developer response

Fixed in commit 5d91208a816f8229b4389b4d48048465816f2356

6. Gas - Unnecessary named return variable assignment (blockdev)

`uint256` variables without explicit assignment are initialized to 0. So, named return variables don't need to be explicitly assigned to 0.

Proof of concept

SingularityPool.sol has `getDepositFee()` and `getWithdrawalFee()` have named return variable `fee` which is assigned as follows:

```
if (feeA > feeB) {
    fee = feeA - feeB;
    require(fee < amount, "SingularityPool: FEE_EXCEEDS_AMOUNT");
} else {
    fee = 0;
}
```

The `else` clause is not required as `fee` is already 0.

Impact

Gas savings

Recommendation

Remove the `else` clause in the code piece shown above for `getDepositFee()` and `getWithdrawalFee()`.

Developer response

Fixed in commit c253d1d7b66d5cb9b82816fb1d4295e078f42701

7. Gas - Combine accounting updates in `swapIn` and `swapOut` (uk)

In `swapIn` and `swapOut` the `assets` storage slot is unnecessarily updated multiple times.

Proof of concept

The following lines in each function can be combined

```
assets -= protocolFee;
assets -= amountOut;
```

Example taken from `swapOut()`, there is a corresponding instance in `swapIn()`.

Impact

Gas Savings

Recommendation

Combine the two statements into one update.

Developer response

Fixed in commit 2f1864d28e9280c24045052e174a49495c8b97e0

8. Gas - Routers can give infinite token approval to pools (blockdev)

To save gas, it's better to give infinite token approval to a contract when compared to approving it only for the required amount. This, however, carries a security risk as the contract can misuse the infinite approval to drain all the tokens.

`SingularityRouter.sol` always transfers the received tokens to a pool so it never holds any funds. Hence, it can give infinite token approval to pools to save gas on approval on each swap or deposit.

Proof of concept

For every swap and deposit, `SingularityRouter.sol` increases its token allowance to the related pools at SingularityRouter.sol#L101 and SingularityRouter.sol#L134.

This costs extra gas when compared to the case where the router gives infinite approval only once.

Impact

Gas Savings

Recommendation

Remove lines SingularityRouter.sol#L101 and SingularityRouter.sol#L134.

Add this function to `SingularityRouter.sol` —

```
function approvePool(address token) external {
    require(msg.sender == factory, "SingularityRouter: NOT_FACTORY");
    address pool = poolFor(token);

    IERC20(token).approve(pool, type(uint256).max);
}
```

and this function to `SingularityFactory.sol` —

```
function approvePool(address token) external onlyAdmin {
    ISingularityRouter(router).approvePool(token);
}
```

Now, whenever a new pool for a token `t` is added, `admin` can call `SingularityFactory.approvePool(t)`.

Developer response

Acknowledged

9. Gas - Immutable state variable which is defined as a public variable (saharAP)

Public state variable `tranche` has been assigned just once in the constructor of SingularityFactory contract Therefore, it is better to define it as `immutable` to save gas.

Proof of concept

`tranche` has been assigned just once in the constructor and it is defined as public state variable.

Impact

Gas Savings

Recommendation

Define `tranche` public variable as `immutable`.

Informational Findings

Residents Informational Findings

1. Informational - Difference from Uniswap's `safeTransferFrom()` logic (engn33r)

The SingularityRouter.sol contract is forked from Uniswap's V2 router. There is a difference in the `safeTransferFrom()` logic. The Uniswap V2 router uses its own `safeTransferFrom()` implementation while SingularityRouter.sol uses the OZ `safeTransferFrom()` implementation.

Technical Details

While it is unlikely that the differences between the Uniswap `safeTransferFrom()` logic and the OZ `safeTransferFrom()` logic leads to any security problems, the Uniswap implementation has been thoroughly tested in the Uniswap V2 router contract and may be a better fit for where SingularityRouter.sol uses `safeTransferFrom()`.

Impact

Informational

Recommendation

Consider using the exact Uniswap `safeTransferFrom()` logic to better mimic Uniswap's AMM logic.

Developer response

Acknowledged

2. Informational - Centralization risk with admin role (engn33r)

The `admin` role in PoolFactory.sol has access to all factory contract administrative functions. The deployer of the factory contract in the existing Fantom deployment may not be a multisig address, which is a centralization risk.

Technical Details

The `admin` role can perform actions including setting the admin address, setting the oracle address, the basefee, and more. If the admin is an EOA, this is a large centralization risk, and even if a multisig is used some additional protection measures could be added (timelock, 2-step admin transfer process, etc.) to increase the protocol's security.

Impact

Informational

Recommendation

Use a multisig for the admin role. Considering adding a timelock or 2-step admin transfer process for certain administrative functions such as `setAdmin()`.

Developer response

Acknowledged

3. Informational - `baseFee` value not limited (engn33r)

`setBaseFee()` does not limit the value that baseFee can hold. Without checks in place, the `baseFee` value could be set to such a high value that it makes the protocol unusable.

Technical Details

`setBaseFee()` has no checks on the function argument value. A `baseFee` value of 5 ether would result in fees 10x the value being traded, making the protocol unusable. Limits should be hardcoded in the contract to increase user trust that the admin address will not rug the users.

Impact

Informational

Recommendation

Add checks to limit the basefee to a reasonable range, such as 0 to 0.5 (50%) or similar. Other methods of increasing user trust may achieve the same end goal.

Developer response

Acknowledged

4. Informational - No way to remove pools (engn33r)

Uniswap is designed to be a DEX, where the D stands for decentralized. The Singularity v2 solution requires an admin to add pools with `createPool()`, so this process is more controlled. It can be useful to have a complementary `removePool()` function to allow for pools to be removed.

Technical Details

Unexpected circumstances related to an ERC20 token that has a Singularity v2 pool may require the pool to be modified or removed by the admin. This is not currently possible with the existing code.

Impact

Informational

Recommendation

Add a `removePool()` function in SingularityFactory.sol.

Developer response

Acknowledged

5. Informational - Fee-on-transfer tokens not supported (engn33r)

The Singularity v2 solution requires an admin to add pools with `createPool()`. The Uniswap router functions supporting fee on transfer tokens have been removed from SingularityRouter.sol, and the `deposit()` and `withdraw()` functions of SingularityPool.sol will not track balances properly with weird ERC20 tokens. The admin must keep this in mind in the future because adding a fee-on-transfer token may result in loss of value.

Technical Details

The `deposit()` function does not check whether the increase in balance is equal to the amount transferred, and therefore this function is not compatible with fee-on-transfer tokens. The same happens in the `withdraw()` function.

Impact

Informational

Recommendation

Clearly document that fee-on-transfer tokens are not compatible with this protocol. Make sure the admin is always aware of this limitation to prevent the addition of such an ERC20 token in the future.

Developer response

Acknowledged

6. Informational - Documentation inconsistencies (engn33r)

The Singularity documentation did not match the contract code in some places. The dev team confirmed in Discord that the code contains more updated information. The docs should be updated.

Technical Details

The official Singularity docs were not consistent with the code. For example, the docs described a g function with the collateralization ratio raised to the 7th power, but the code raised this value to the 8th power.

Impact

Informational

Recommendation

Update the developer documentation to match the implementation.

Developer response

Acknowledged

7. Informational - Decimals value borrowed from underlying ERC20 (engn33r)

The decimals of the Singularity LP token will use the same decimals value of the underlying token used by the pool. This is different than the approach used by Uniswap and many other LPs, where the decimals value is consistently 18. This inconsistency may cause problems with future integrations.

Technical Details

Uniswap uses 18 decimals for all LP tokens. But Singularity uses the decimals value from the underlying ERC20 token.

Impact

Informational

Recommendation

Use a consistent decimals value of 18 for all Singularity LP tokens. Following Uniswap's approach even save some gas by storing the value in a constant.

Developer response

Acknowledged

8. Informational - Typo (engn33r)

There is a typo in a function name.

Technical Details

`_calcCollatalizationRatio()` might be better spelled `_calcCollateralizationRatio()`.

Impact

Informational

Recommendation

Fix typos.

Developer response

Fixed in commit 5d5a38cf7bffd5510c3aa0bfe983bf9bd5626fa2

9. Informational - nonReentrant modifier is specific to each pool (engn33r)

There is a `nonReentrant` modifier on several functions in SingularityPool.sol. These modifiers only prevent reentrancy in other functions in the same contract. These modifiers do not prevent reentrancy risk between different pools. The lack of a global lock is one key detail that led to the CREAM exploit. Although no exploit path along these lines was found during this code review, such protections may be worth considering if ERC20 tokens with callbacks are added to the protocol in the future.

Technical Details

The nonReentrant modifier only protects against reentrancy within the contract. The CREAM exploit leveraged the lack of global reentrancy protection.

Impact

Informational

Recommendation

Consider adding global reentrancy protection if certain weird ERC20 tokens are added to the protocol in the future.

Developer response

Acknowledged

10. Informational - Pool may not be able to reach `depositCap` (engn33r)

The `depositCap` state variable sets an upper bound on the value that a pool can hold. The pool may revert before it can reach this upper bound because `rpow()` may cause the revert for large collateralization ratio values.

Technical Details

If the collateralization ratio is very large (greater than, say, 10^10), the `rpow()` call in `_getG()` may revert because of how `rpow()` is implemented.

Impact

Informational

Recommendation

When setting `depositCap`, consider the limits at which `rpow()` will revert. A change was made in the dev branch while the review was ongoing for a related issue, but it is unclear whether the chosen value of 1.5 ether is the exact tipping point that results in a revert for this specific case.

Developer response

Acknowledged

Fellows Informational Findings

1. Informational - Incorrect balance accounting for fee-on-transfer and rebasing tokens (blockdev)

SingularityPool.sol does not support creating pools for ERC20 tokens which take fee on transfer. To transfer tokens, the pool uses `safeTransfer()` function which internally calls ERC20's `transfer()` function. If the token takes a fee on transfer, this call still succeeds but the pool might receive fewer tokens than intended.

There is a safeguard since the protocol team decides which token will have a pool, but a fee can be turned on after a pool is created. This is also a risk with proxy and rebasing tokens.

Proof of concept

USDT has a zero fee on transfer but it can be turned in future, resulting in lower funds in pool. Consider its `transfer()` function —

```
function transfer(address _to, uint _value) public onlyPayloadSize(2 * 32) {
    uint fee = (_value.mul(basisPointsRate)).div(10000);
    if (fee > maximumFee) {
        fee = maximumFee;
    }
    uint sendAmount = _value.sub(fee);
    balances[msg.sender] = balances[msg.sender].sub(_value);
    balances[_to] = balances[_to].add(sendAmount);
    if (fee > 0) {
        balances[owner] = balances[owner].add(fee);
        Transfer(msg.sender, owner, fee);
    }
    Transfer(msg.sender, _to, sendAmount);
}
```

2. Informational - Reduce footgun risk of deposit() and withdraw() (devtooligan)

The dev team has confirmed that the `deposit()` and `withdraw()` functions are intentionally "low-level functions" that could result in loss if the caller is not careful. While true, we still recommend mitigating the risk of misuse as a kind of public good. The lowest cost way of reducing this risk is through natspec on the functions with a clear warning to callers of the potential pitfalls.

Another potential solution to consider that would greatly eliminate this risk would be by adding the `minIn` / `minOut` params to the internal `swapIn()` `swapOut()` fns, thereby forcing the caller to intentionally pass 0 if they really did not want slippage protection. This would slightly add to the overall runtime gas profile but the difference would be negligible. It costs much less gas to do this than to, for example, call `routerOnly` which incurs a cold SLOAD.

Guest auditors Informational Findings

1. Informational - Code does not match docs

Impact

The code does not match the math mentioned in the docs. Some discrepancies:

1. gCurve: docs use `0.00002/r^7`, code `0.00003/r^8`. https://github.com/revenant-finance/singularity-v2/blob/a3cdbc5515374c9e1792b3cb94ff1b084a9a1361/contracts/SingularityPool.sol#L445
2. `_getG` is not continuous, there's a strange bump at 30% CR: https://github.com/revenant-finance/singularity-v2/blob/a3cdbc5515374c9e1792b3cb94ff1b084a9a1361/contracts/SingularityPool.sol#L441 `g(0.3) = 0.00003 / (0.3^8) = 0.45724737`, `g(0.2999...) = 0.43 - 0.3 = 0.13`.
3. Trading fee docs uses hardcoded `oracleSens = 27` https://docs.revenantlabs.io/singularity/details/swapping/trading-fee code seems to use a base `oracleSens` value of 60 and works with 10% buffer before considering it stale
4. "45% is locked as protocol owned liquidity" https://docs.revenantlabs.io/singularity/details/swapping/trading-fee The code only uses `protocolFee` and `lpFee`.
5. `getDepositFee` and `getWithdrawalFee` is different from the docs. https://github.com/revenant-finance/singularity-v2/blob/a3cdbc5515374c9e1792b3cb94ff1b084a9a1361/contracts/SingularityPool.sol#L290
6. Consider using `e18` scientific notation instead of the `ether` keyword

Recommended Mitigation Steps

Update the docs.
Update: the docs have been updated in the meantime.

2. Informational - Just-in-time LP provision can be profitable

Impact

Even though there can be bad deposit and withdrawal fees to providing liquidity to pools, it can be profitable to provide just-in-time liquidity for trades to capture the part of trading fees that go to LPs.
Sandwich a trade by depositing and withdrawing liquidity.

two pools with assets=liabilities = 1000.0 USDC, 1000.0 DAI (`baseFee = 0.0015`)
A trader wants to swap 650.0 DAI to USDC.
The sandwich is created as follows:

1. deposit(2,100.0 DAI)
2. swap(650.0 DAI) is performed
3. withdraw liquidity from step 1
4. make a profit of 0.214 DAI (0.032923% of swap size)

The higher the trading fees (`baseFee` or "stale price modifier") the more profitable the attack becomes.

See the POC here.

Recommended Mitigation Steps

It's hard to fix in the current AMM design. One can play around with the `baseFees` and `withdrawal`/`deposit` fees to reduce the profitability of JIT LPing.

Developer response

Acknowledged

3. Informational - Potentially misleading description of "no impermanent loss"

Impact

The docs say:

> "In contrast, Singularity implements single-sided deposits. This effectively eliminates impermanent loss."

If we take a broad definition of impermanent loss meaning that it can be worse to provide liquidity compared to just holding onto the tokens, then this claim is false.
There are potentially large withdrawal fees for under-collateralized pools and one can still realize a loss compared to a hold-strategy.

A different section of the docs later clarifies this behavior.

> "While LPs are not exposed to impermanent loss, they are exposed to pool imbalance risk when withdrawing their liquidity. If a pool is heavily undercollateralized (assets < liabilities), then the withdrawal fee incurred may be significant."

Recommended Mitigation Steps

Consider stating the exact risks instead of saying there is no impermanent loss.

Developer response

Acknowledged

4. Informational - Miscellaneous

Impact

- SingularityOracle.getLatestRound: Unresolved ToDo
- Gas: `SingularityOracle.getLatestRound`: This reads all elements from storage and stores it in memory which is very inefficient. Directly access only the last element instead: `price = allPrices[token][allPrices.length - 1].price`
- SingularityOracle.getLatestRound: This will revert if no price has been pushed yet.

Final remarks

NibblerExpress

As noted by devtooligan, the pools behave best when the collateralization ratio is close to one. There is an incentive for arbitrageurs to bring the collateralization ratio back towards one, so the system should perform well in many instances. The arbitrage back to one should be encouraged while being careful about fee manipulation. The pools exist in a metastable state when undercollateralized. There may end up being a bank run draining all assets in the pool if LP holders do not believe the collateralization ratio will return to one. The protocol charges low fees to the first users to exit the pool and high fees to later users to exit. While users will feel like they still have access to their money when the collateralization ratio is less than one, it does create the potential for a race to the exit. The developers should be more careful about discontinuities in functions and first derivatives of functions. Although I haven't identified an attack related to the discontinuity in g at 0.3, I am wary that one may exist. The protocol should punish users and attackers more harshly for manipulations of the collateralization ratio by depositing or withdrawing funds.

engn33r

I like the AMM design using the Uniswap v2 router and factory. The issues that concern me are in the SingularityPool.sol contract. The incentives and mechanics of the design concern me and may lead to gamification by arbitrageurs and other profit-seeking actors. This is especially true because flashloans can be used (there is no limit on multiple pool interactions per block) and slippage can lead to pool loss of asset value. Undercollateralization of multiple pools could cause health problems for the overall protocol and a bank run would result in loss of user value. Beyond changes to the protocol mechanics design, the NatSpec documentation in the contracts was almost non-existent and should be improved before launch.

Benjamin Samuels

I focused most of my time into building Echidna invariants & attacking the various swap interfaces. My largest concerns stem from the `_getG()` function and how slippage+trading fees are clueless as to how volatile their pool's assets are.

With respect to the `_getG()` function, while this function currently fits the requirements, it feels like the function lacks formal rigor; why use the 8th power?; why use an inverse?; why use 0.00003 for the numerator? The requirements might be met by the current function, but it's very likely that there's something better out there that can provide better prices for users. I recommend taking a look at Curve's whitepaper and potentially borrowing some of their work, especially with respect to adjustable curve steepness.

High volatility is by far the largest threat to Singularity LPs, and the trading fees they earn should correlate with the asset's volatility. I recommend re-examining how trading fees & slippage are calculated; this is important not only to adequately compensate LPs, but to protect the protocol from being drained if the trusted oracle publishes a large price change.

Given how unique the protocol's construction is, I recommend being very conservative with deposit caps until the formal properties of the protocol are better understood.

blockdev

The code was modular and easy to understand. There are certain trust assumptions present in the code which if violated can lead to a loss of funds across all the pools.

First, trust on Chainlink to report correct prices. This assumption is stronger for stablecoin pools. There is no safeguard against Chainlink oracle misbehavior.

Second, trust on admin to align with protocol's long term benefit. Since the admin sets the fee and collects it, they have an incentive to act for their gains against protocol health. The admin also allowlists a token for a pool. It's critical that every new token is thoroughly reviewed so that it's pool doesn't lead to a protocol exploit.

Finally, a pool can interact with any other pool through swaps, protocol security is critical since a vulnerability can lead to a loss of funds from all pools. Hence, I recommend the devs to maintain high standards for code security, documentation and admin access.

devtooligan

The code was pretty clean and easy to grok. There was clearly some thought given to gas optimizations.

When collateralization ratios are closer to 1 the protocol behaves pretty well. But when the collateralization ratio drops enough things get a little weird when the g curve starts steepening. I think there is still more work that can be done in considering the shape of the curve and rigorously modelling some of the things that might happen when collateralization ratios are further away from 1.

The oracles should also be rigorously tested. The "on-chain oracle" is not in a final polished state at this point. It appears ithasn't been fully specced out yet and that the intention was to build something generic that could handle whatever "on-chain oracle" gets decided upon later. Once the oracle system and the SingularityOracle.sol contract are finalized, recommend conducting a thorough security review of all of that.

uk

I really liked the design of the system, and the code was well written and easy to read. My main concerns are around the economics and system design. Singularity is designed around a very particular curve and the choices that led to this curve being used are not clear. The parameters seem to be changing from time to time and it is unclear if one set is superior to another. There has been care put into protecting LPs, with specific attention paid to the collateralization ratio. However, it is not obvious to me that optimizing purely on the cRatio is the best move. It seems like this can be done to users' detriment (incentivizes low liquidity, high cRatio pools). I would also encourage the authors to comment the code in depth, and keep the code and documentation in sync.

About yAcademy

yAcademy is an ecosystem initiative started by Yearn Finance and its ecosystem partners to bootstrap sustainable and collaborative blockchain security reviews and to nurture aspiring security talent. yAcademy includes a fellowship program and a residents program. In the fellowship program, fellows perform a series of periodic security reviews and presentations during the program. Residents are past fellows who continue to gain experience by performing security reviews of contracts submitted to yAcademy for review.