Pre-Audit, Prepared For Singularity-v2

Goober, 0xBebis, 0xNazgul

for commit: 1c0f89db7335a709b7a38c9c6c0b66691e79096c

1) Introduction

The purpose of this pre-audit is to de-risk the beta launch of the singularity protocol. The approach taken was to fuzz immediately used functions like depositing and withdrawing from singularity pools, and to create a broad, holistic overview of security concerns and attack vectors.

2) Conversations

2.1) Oracles

After discussions with both the Revenant Labs team and internally with the masons, the largest point of failure was decidedly the reliance on the oracle, and the lack of fall-backs for if it fails to produce good data. A Zero-Check was already implemented which takes care of the case in which the oracle utterly fails to produce proper data, which, falling well outside the discussed 1%,[1] rails of acceptable value would pause trading with that pool. That leaves the issue mainly with the oracle producing data that may be different from on chain conditions about the asset. As discussed, Chainlink should report new round data with a .2% price movement, but the actual on chain conditions where the liquidity may be lower than the Chainlink sources could reflect a much sharper shift in price. Therefore, as discussed, an on chain TWAP or Spot Read, should thoroughly derisk operation during volatility. Should the price on chain differ from the oracle price by more than 1%, operation of that pool should immediately cease until an owner/guardian can evaluate that the pool can safely restart operation.

2.2) Collateral Damage

As discussed, creating a clear internal framework for evaluating the worthiness of collateral, and which tranche they should belong to is imperative to the safety of the system. I personally recommend starting with only top-tier collateral at the genesis of the system, such as USDC, USDT[2], and DAI. Eventually the FRAX and UST pools must come to stay competitive, but not without proper incentive to do so.

Continued next page ...

1 | At the time of writing this is currently set to 1.5%
2 | Debateable if it is top-tier, but top-tier nonetheless

3) Nitpicks and Gas Optimizations

3.1) Unlocked Pragma Version

Severity: Informational
Context: All Contracts
Description: Contracts should be deployed using the same compiler version/flags with which they have been tested. Locking the pragma (for e.g. by not using ^in pragma solidity 0.5.10) ensures that contracts do not accidentally get deployed using an older compiler version with unfixed bugs.

Recommendation: Lock the pragma version to only one.

3.2) Too recent of a Pragma

Severity: Informational
Context: All Contracts
Description: Using too recent of a pragma is risky since they are not battle tested. A rise of a bug that wasn't known on release would cause either a hack or a need to secure funds and redeploy

Recommendation: Use a Pragma version that has been used for sometime. I would suggest '0.8.4' for the decrease of risk and still has the gas optimizations implemented.

3.3) Caching The Array Length Prior To Loop

Severity: Gas Optimization
Context: ['SingularityFactory.sol#L93-L96'], ['SingularityFactory.sol#L99-L105'], ['SingularityFactory.sol#L108-L115'],['SingularityFactory.sol#L118-L124'], ['SingularityFactory.sol#L127-L131'], ['SingularityOracle.sol#L48-L56'], ['SingularityOracle.sol#L65-L70'], ['SingularityOracle.sol#L89-L94'], ['SingularityOracle.sol#L48-L56']
Description: One can save gas by caching the array length (in stack) and using that set variable in the loop. Replace state variable reads and writes within loops with local variable reads and writes. This is done by assigning state variable values to new local variables, reading and/or writing the local variables in a loop, then after the loop assigning any changed local variables to their equivalent state variables.

Recommendation: Simply do something like so before the for loop: "`uint length = variable.length`". Then add "`length`" in place of "` variable.length`" in the for loop.

Continued next page ...

3.4) Setting The Constructor To Payable

Severity: Gas Optimization
Context: ['SingularityFactory.sol#L29-L38'], ['SingularityGovToken.sol#L16-L18'] ['SingularityPool.sol#L50-L57'], ['SingularityRouter.sol#L28-L32']
Description: You can cut out 10 opcodes in the creation-time EVM bytecode if you declare a constructor payable. Making the constructor payable eliminates the need for an initial check of 'msg.value == 0' and saves 21 gas on deployment with no security risks.

Recommendation: Set the constructor to payable.

3.5) Function Ordering via Method ID

Severity: Gas Optimization
Context: All Contracts
Description: Contracts most called functions could simply save gas by function ordering via Method ID. Calling a function at runtime will be cheaper if the function is positioned earlier in the order (has a relatively lower Method ID) because 22 gas are added to the cost of a function for every position that came before it. The caller can save on gas if you prioritize most called functions. One could use ['This tool'](https://emn178.github.io/solidity-optimize-name/) to help find alternative function names with lower Method IDs while keeping the original name intact.

Recommendation: Find a lower method ID name for the most called functions for example "'mostCalled()" vs. "'mostCalled 41q()" is cheaper by 44 gas.

3.6) Improper State Upgrade Flow

Severity: Minor
Context: ['SingularityPool.sol L336']
Description: setDepositCap input should be sanitized to make sure it can only be increased or at the very least should never be lowered to an amount that is less than the existing amount deposited.

Recommendation: require(newDepositCap > depositCap, "New Deposit Cap Must Be Greater Than Existing Cap");