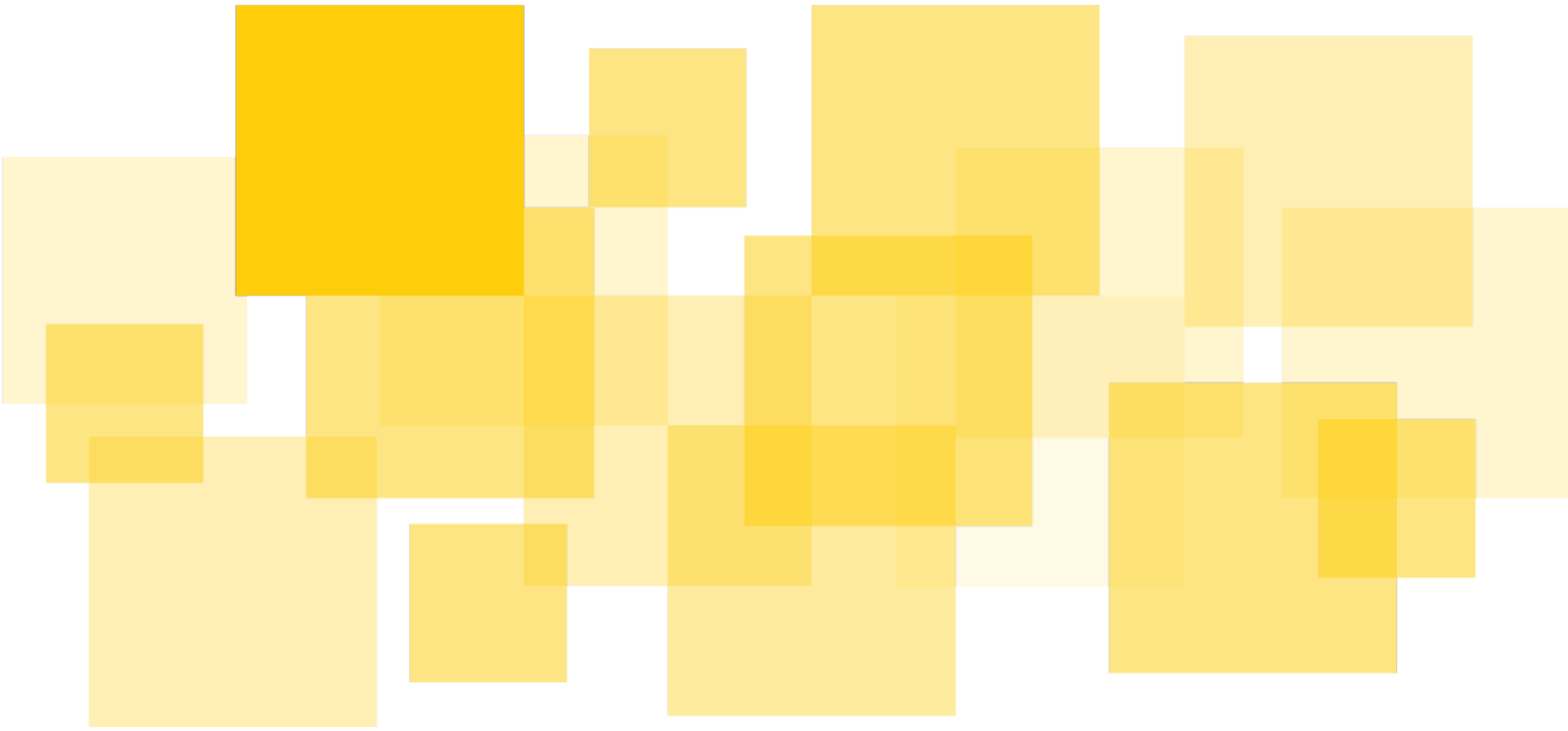
Security Audit Report

Swaap Labs Pools

Delivered: May 27, 2022



Prepared for Swaap Labs by

runtime verification

Disclaimer

This report does not constitute legal or investment advice. You understand and agree that this report relates to new and emerging technologies and that there are significant risks inherent in using such technologies that cannot be completely protected against. While this report has been prepared based on data and information that has been provided by you or is otherwise publicly available, there are likely additional unknown risks which otherwise exist. This report is also not comprehensive in scope, excluding a number of components critical to the correct operation of this system. This report is for informational purposes only and is provided on an "as-is" basis and you acknowledge and agree that you are making use of this report and the information contained herein at your own risk. The preparers of this report make no representations or warranties of any kind, either express or implied, regarding the information in or the use of this report and shall not be liable to you or any third parties for any acts or omissions undertaken by you or any third parties based on the information contained herein.

Summary

Runtime Verification, Inc. conducted a security audit on the Swaap Pool contracts. The audit was conducted from March 14th, 2022, to May 27th, 2022.

Several issues have been identified as follows:

- Implementation flaws: A03
- Potential security vulnerabilities: A01, A02, A04, A05, B01, B07

A number of additional suggestions have also been made, including:

- Input validations: B10, B11
- Gas optimization: B08, B09
- Code readability: B02, B03, B03, B04, B05, B06
- Arithmetic Precision: B12, B13, B14, B15
- Best Practice: B16, B17

All the critical security issues have been addressed by the client. Details can be found in the Findings section as well as the Informative Findings section.

Scope

The targets of the audit are the smart contract source files at git-commit-id `f91cf548fc229ac21badeca4adff67a5dfc06b6d`

The audit focused on the following core contracts and interfaces:

- `ChainlinkUtils.sol`
- `Const.sol`
- `Factory.sol`
- `GeometricBrownianMotionOracle.sol`
- `Math.sol`
- `Num.sol` (only `abs()` and `max()` functions)
- `Pool.sol`
- `PoolToken.sol`
- `interfaces/IAggregatorV3.sol`
- `interfaces/IPausedFactory.sol`
- `structs/Struct.sol`

The audit is limited in scope within the boundary of the Solidity contract only. Off-chain and client-side portions of the codebase, as well as deployment and upgrade scripts are *not* in the scope of this engagement but are assumed to be correct (see below).

Assumptions

The audit is based on the following assumptions and trust model.

- External token contracts conform to the ERC20 standard, especially that they *must* revert in failures. They do *not* allow any callbacks (e.g., ERC721 or ERC777) or any external contract calls. They do *not* implicitly update the balance of accounts (e.g., rebasing tokens or fees on transfers) other than explicit events of transfers, mints, or burns.
- For the authorizable contracts, their owners and authorized users are trusted to behave correctly and honestly, as described in the Security Model section.
- The contracts will be deployed and integrated correctly.

Methodology

Although the manual code review cannot guarantee to find all possible security vulnerabilities as mentioned in Disclaimer, we have followed the following approaches to make our audit as thorough as possible. First, we rigorously reasoned about the business logic of the contract, validating security-critical properties to ensure the absence of loopholes in the business logic and/or inconsistency between the logic and the implementation. To this end, we developed a high-level model of the central logical parts. Then, we carefully checked if the code is vulnerable to known security issues and attack vectors. Finally, we symbolically executed part of the liquidity provision formulas to systematically search for unexpected, possibly exploitable, behaviors and enhance arithmetic precision.

Security Model

The contracts are operated by multiple parties, and the secure operation of the contracts requires all the participating agents to behave correctly, honestly, and diligently. This section contains an (incomplete) list of the privileges and responsibilities of the participants.

Swaap Labs factory owner

Privileges

- Forward collected exit fees
- Transfer swaap labs privileges to new address
- For a limited time period after deployment: pause/unpause some capabilities of the protocol

Responsibilities

- Pause the protocol in case of emergency
- Unpause the protocol in case the emergency has been resolved

Pool Controller

Privileges

- Configure pool parameters, including swap fee, dynamic coverage fees z, dynamic coverage fee horizon, price statistics loopback in rounds, price statistics loopback in seconds
- Bind new tokens to non-finalized pools. This comprises setting the token address, the price feed address, the initial supply and weight.
- Rebind token in non-finalized pools. This comprises updating the price feed address, the initial supply and weight of a previously bound token.
- Unbind tokens from the pool.
- Transfer controller privileges to new address
- Allow/Disallow public swapping on the pool (trading is now permission-less)
- Finalize the pool (liquidity provision is no permission-less)

Responsibilities

- Before transferring controller privileges withdraw/unbind all tokens unless you want to donate the tokens to the new controller

- Ensure that a token points to the true source address when binding
- Ensure that the price feed points to the correct chainlink oracle when (re)binding
- Ensure that all price feeds use the same currency peg when (re)binding
- Ensure that no token is bound more than once
- Only bind stock-like assets that cannot have negative prices

Liquidity Provider

Responsibilities

- Validate pool is finalized
- Validate the pool parameters and be aware the controller is able to change the parameters for non-finalized pools.
- Validate pool tokens point to the true source address of a token
- Validate oracle addresses point to the correct price feed
- Validate that all price feeds use the same currency peg
- Validate that no token is bound more than once by different addresses, see A05
- Validate that no bound token is deflationary, see A01
- Observe the price feeds for unexpected negative prices, see A02
- Always specify your trading limits when joining/exiting a pool
- Observe paused-state and re-evaluate risk

Trader

Responsibilities

- Validate the pool parameters and be aware that the controller is able to change the parameters for non-finalized pools.
- Validate pool tokens point to the true source address of a token
- Validate oracle addresses point to the correct price feed
- Validate that all price feeds use the same currency peg
- Validate that no token is bound more than once by different addresses, see A05
- Observe the price feeds for unexpected negative prices, see A02
- Always specify your trading limits. This becomes even more important when you are trading a non-finalized public pool, see A06.
- Observe paused-state and re-evaluate risk

Findings

A01: Pools are vulnerable to deflationary tokens

[Severity: High | Difficulty: Mid | Category: Security]

Description

Some tokens charge fees on transfers or can possibly do that in the future (e.g. USDT), this means that these tokens don't transfer the entire amount of tokens specified by the caller. When not considering this kind of behaviour the internal accounting may become corrupted. In the pool contract this can happen when transferring tokens to the pool. Since the spot price calculation is based on the internal balances and not on the real balances, this may lead to unexpected prices. Moreover, an attacker can exploit this scenario to drain pools.

Notice that the difficulty classification is hardly applicable here. If there is a deep pool with a deflationary token, the attack is easy to execute. If there is no such pool, the attacker needs to create such a pool and lure his victims into using it, e.g. by social engineering - which we consider difficult. Another difficult possibility for the attacker would be to bribe the development team of an upgradeable token to deflate it.

Notice, that a similar issue has been exploited on Balancer pools before: https://blog.1inch.io/balancer-hack-2020-a8f7131c980e

Scenario

- The pool controller binds 2 tokens to the pool with the initial transfer of 100 tokens of each token.
- One of the tokens, let's say Token2 charge 10% as a fee for transfer while Token1 doesn't charge any fee.
- This means that the real balance of Token1 is 100 but the real balance of Token2 is 90. However, the _records[Token2].balance will store 100
- Let's suppose that now some trader wants to swap 10 tokens of Token1 by Token2. Simplifying the other parameters of the functions, according to the balances stored in the pool contract the trader will receive 10 tokens of Token2 because the spot price calculated is 1. However, if we considered the real balances of the tokens, the trader would only receive 9 tokens.

This happens because the contract doesn't consider the possible fee charge on transfers when updating the balances of the tokens:

Pool.sol#L862-L864:

```
_records[token].balance = balance;
if (balance > oldBalance) {
    _pullUnderlying(token, msg.sender, balance - oldBalance);
```

Notice that the fee charged in this example is high in order to demonstrate the problem easily, but even with smaller fees the difference between the real balance and the balance recorded in the contract may become significant because that difference will grow with the number of the transfers. Notice that this behaviour also happens in other functions:

- `joinPool`
- `joinswapExternAmountInMMM`
- `joinswapPoolAmountOutMMM`
- `_rebindMMM`
- `_swapExactAmountInMMMWithTimestamp`
- `_swapExactAmountOutMMMWithTimestamp`

Recommendation

Update the token balance internally by calling the balanceOf function after every transfer.

Status

Swaap Labs addressed the issue by checking if the real balance is as expected after an asset is pulled in commit id: `d52cc29`

A02: Undefined behavior in case of negative oracle prices

[Severity: Mid | Difficulty: n/a | Category: Security]

Description

The protocol has been designed with stock-like assets in mind that always have a positive price. In practice, however, there are valid use cases for assets that can take negative prices. Oracles like Chainlink, therefore, return signed integers to account for these use cases as well. Now, if an oracle returns a negative price to a Swaap pool it is unclear how to proceed. The implementation treats these prices as zero.

```
function _getTokenCurrentPrice(IAggregatorV3 priceFeed) internal view
returns (uint256) {
     (, int256 price, , ,) = priceFeed.latestRoundData();
     return Num.abs(price);  // we consider the token price to be > 0
}
```

Recommendation

Since the intended behavior is not precisely specified we recommend against treating negative prices as zero, and instead revert the transaction. Moreover, an observed negative price may indicate a misconfigured pool, and participants are encouraged to monitor for such reversals and re-evaluate the pool's configuration.

Status

Swaap Labs addressed the issue by reverting when a price oracle reports a negative price in commit id: `6376d1a`

A03: Incorrect variance calculation in GBM

[Severity: Mid | Difficulty: n/a | Category: Implementation flaw]

Description

The protocol dynamically charges spreads on trades when the price of an asset is volatile. Asset prices are modeled using a geometric brownion motion, and the spread factor is determined by the variance and mean parameters of said process. According to the specification, the variance parameter should be estimated by the following formula:

$$\sigma^2 = \frac{1}{n}[-\frac{1}{T}\log(\frac{S_n}{S_0})^2 + \sum\frac{(x_i)^2}{\Delta_i}]$$

where

n is the number of price samples

T is the sample period

S_i is the i-th price sample

$x_i = S_i - S_{i-1}$ for $i \geq 1$

Δ_i is the time difference between samples i and $i-1$ for $i \geq 1$

However, the implementation doesn't follow that formular and instead calculates the following incorrect value:

$$\sigma^2 = \frac{1}{n}[-(\frac{1}{n\,T}\log(\frac{S_n}{S_0})^2) + \sum\frac{(x_i)^2}{\Delta_i}]$$

Notice the additional factor `1/n` in the first summand.

As a consequence, the variance of the price history is systematically under-estimated, in particular volatile assets might mistakenly be classified as stable and the adaptive fees applied to traders will be too low. That is liquidity providers will receive less fees.

Recommendation

Change the implementation to follow the formula of the whitepaper.

Status

Swaap Labs addressed the issue in commit-id: `d122e34`

A04: Pool controller can withdraw deposits from previous pool operators

[Severity: High | Difficulty: High | Category: Security]

Description

Pool controllers are enforced to deposit an initial supply when binding an asset. Because the binding happens before the pool is finalized the controller will not get pool share tokens minted immediately to represent his ownership. If the controller transfers his privileges to another account, the new controller will implicitly become the new owner of the initial pool supplies. He can then, for example, withdraw the initial supplies by unbinding an asset.

If a malicious user manages to convince a pool controller to grant him the controlling privileges, i.e. by social engineering, he can exploit the scenario and steal assets from his victim.

Scenario

1. Alice creates a new pool
2. Alice binds wETH with an initial supply of 10 wETH to the pool.
3. Alice transfers the operator privileges to Bob.
4. Bob unbinds wETH and thereby withdraws Alice's 10 wETH.

Recommendation

Add a sanity check to the setOperator method and revert if the pool is not finalized and has tokens bound to it.

Status

Swaap Labs addressed the issue in commit id: `9cf16f8`

A05: Tokens with multiple addresses can be used to create corrupted pools

[Severity: High | Difficulty: High | Category: Security]

Description

Some proxied tokens have multiple addresses, and state-changes made to one address are immediately replicated on the other addresses. That makes it possible for a controller to bind the same token more than once to a pool. In that case the internal accounting of the Pool becomes corrupted. An attacker can exploit such pools.

Scenario - Draining a pool

- Assume we had a 3-token Pool: TokenA/TokenA/TokenB
  - Notice that the pool binds TokenA twice
  - Assume the internal balances are: 50/50/100
  - The real balances are: 100 TokenA and 100 TokenB
- Alice joins the pool with 50/50/100
  - Alice mints 50% of pool share tokens
  - The internal balances are updated to: 100/100/200
  - The real balances are: 200 TokenA and 200 TokenB
- Alice calls the gulp function
  - The internal balances are updated to 200/200/200
  - The real balances don't change
- Alice exits the pool with 50% pool share tokens.
  - Alice receives 100/100/100 = 200 TokenA and 100 TokenB
  - The internal balance is updated to: 100/100/100
  - The real balance is: 0 TokenA and 100 TokenB

Recommendation

Document the behavior prominently and transparently. Liquidity providers and traders should avoid pools with tokens that are bound more than once.

Status

Swaap Labs addressed the issue by publicly documenting the behavior at https://docs.swaap.finance/docs/security/recommendations

A06: Malicious pool controllers can manipulate swap fees on public pools

[Severity: High | Difficulty: Mid| Category: Security]

Description

A malicious pool controller can change the swap fees arbitrarily on non-finalized pools. In particular, he can adjust the fees on *public* pools. Operators can exploit this situation and arbitrage slippage.

Scenario

1. Assume Alice creates an A/B pool with a swap fee of 1%.
2. Bob wants to buy 100 A tokens
    a. The current spot price for 1A/B.
    b. Bob specifies that he wants to pay at most 101B.
    c. The transaction is queued in the mempool.
3. The price of A drops to 1.1A/B.
    a. Bob would make a profit due to Slippage
    b. He only needs to pay ~= 90 B + 1B fees
4. Alice frontruns Bob transaction
    a. Alice increases swap fee to 10%
5. Bob's transaction comes in
    a. Bob is quoted a price of 91B + 10B fees.

Notice, that this is not the only scenario where liquidity providers can arbitrage slippage. Essentially, liquidity providers can frontun any trade and move the price to the limit that the trader is still willing to pay. The above scenario is highlighted, because: 1) The pool operator is the only LP and collects the slippage all for himself 2) There is no financial risk for the pool controller and it's much cheaper to collect the slippage via changing the swap-fees than for other liquidity providers.

Recommendation

A. Add another parameter to the trading functions that can be used to specify a limit on the trading fees.
B. Forbid changing the swap fee after a pool is made public for the first time.

Discussion

Swaap Labs acknowledges the issue and asses that such scenarios are unlikely to occur. Notably, such pools would not be competitive to perform additional trades while frontrunning the trader. Also, traders are still protected by the slippage tolerance they implement.

Status

Swaap Labs acknowledges this issue. No on-chain solutions is applied.

Informative Findings

B01: Unsafe token transfer in Factory.collect

[Severity: Informative | Difficulty: Low | Category: Security]

Description

The Factory.collect() function is responsible for transferring collected exit fees to the swaap labs account. Exit fees can be either pool share tokens (when the pool is finalized) or underlying asset tokens (before the pool is finalized). Therefore, the collect method should integrate with arbitrary ERC20 tokens. However, a call to collect will revert if the underlying asset token does not return a boolean value on transfer-calls. This is for example the case for USDT. In this case, the accumulated exit fees are permanently locked into the factory contract.

```
function collect(Pool pool)
    external
{
    require(msg.sender == _swaaplabs, "34");
    uint256 collected = IERC20(pool).balanceOf(address(this));
    bool xfer = pool.transfer(_swaaplabs, collected);
    require(xfer, "19");
}
```

Notice, that the exit fee is set to 0 globally via a constant. That means two things:

1. The factory cannot accumulate exit fees
2. It's not possible to change the exit fee on-chain

Consequently, there is no immediate malfunction arising from the code. However, if the exit fee is changed in a future version of the contract the defect can cause loss of accumulated fees.

Recommendation

Long term: Consider removing the exit fee feature all together.

Short term: Use a library, e.g. OpenZeppelin's SafeERC20, and wrap the transfer-call into a safeTransfer-call to integrate properly with tokens that don't return boolean values on transfer calls.

Status

Swaap Labs addressed the issue by incorporating the short term solution in commit id: `c000465`

B02: Confusing variable name and type in Factory.collect

[Severity: Informative | Difficulty: n/a| Category: Code Readability]

Description

The parameter name and type of the `Factory.collect` function is confusing. The purpose of the method is to transfer exit fees to the `_swaapLabs` address. The exit fee is either paid in pool share tokens (when the pool is finalized) or in underlying asset tokens (otherwise). The parameter name and type `Pool pool` suggest that this function can only be used to withdraw pool share tokens. The type is, however, not enforced at runtime, and the method can in fact be used to withdraw exit fees paid in underlying asset tokens as well.

```
function collect(Pool pool)
    external
{
    require(msg.sender == _swaaplabs, "34");
    uint256 collected = IERC20(pool).balanceOf(address(this));
    bool xfer = pool.transfer(_swaaplabs, collected);
    require(xfer, "19");
}
```

Recommendation

Rename the `pool` parameter and give it the more general type `IERC20`.

Status

Swaap Labs addressed the issue in commit id: `c000465`

B03: Inconsistent NatSpec comments

[Severity: Informative | Difficulty: n/a| Category: Code Readability]

Description

The NatSpec comment of Chainlink.getTokenRelativePrice is self-contradictory. The general description says “Computes the price of token 1 in terms of token 2” while the return-value description says “The price of token 2 in terms of token 1”. The return-value comment seems to be the correct one.

Similarly, the NatSpec comment of Chainlink.getPreviousPrice confuses some parameter names.

Recommendation

Fix the NatSpec comments.

Status

Swaap Labs addressed the issue in commit-id: `9fa88d2`

B04: Factory lacks IPausableFactory-interface inheritance

[Severity: Informative | Difficulty: n/a | Category: Code Readability]

Description

The Factory-contract implements the IPausableFactory-interface, but the contract declaration lacks the appropriate inheritance specifier. In general, this is not a problem due to how down-casting and the ABI encoding works in Solidity. However, it might confuse programmers coming from traditional programming languages. Moreover, it renders the code less future-proof because the contract is not actually checked against the interface declaration. For example, if the interface adds a method and the programmer forgets to implement the method in the contract he will not get a compiler warning.

Recommendation

Add the IPausableFactory to the inheritance specifier list of the Factory contract.

Status

Swaap Labs addressed the issue in commit-id: `92cba60`

B05 Discrepancy between code comment and implementation in Math.calcSingleInGivenPoolOut

[Severity: Informative | Difficulty: n/a| Category: Code Readablity]

Description

The code-comment above the Math.calcSinlgeInGivenPoolOut-function (Math.sol#194-L203) states that it implements the following formula:

```
/**********************************************************************************************
// calcSingleInGivenPoolOut                                                                  //
// tAi = tokenAmountIn              //(pS + pAo)\     /    1    \\                           //
// pS = poolSupply                 || ---------  | ^ | --------- || * bI - bI                //
// pAo = poolAmountOut              \\    pS    /     \(wI / tW)//                           //
// bI = balanceIn          tAi =  -----------------------------------------                  //
// wI = weightIn                            /      wI  \                                     //
// tW = totalWeight                        |  1 - ----  |  * sF                             //
// sF = swapFee                             \      tW  /                                     //
**********************************************************************************************/
```

However, the implementation is actually implementing the following formula:

```
// Using the same equation as calcPoolOutGivenSingleIn but with tAi as the unknown          //
// calcSingleInGivenPoolOut                                                                  //
// tAi = tokenAmountIn              //(pS + pAo)\     /    1    \\                           //
// pS = poolSupply                 || ---------  | ^ | --------- || * bI - bI                //
// pAo = poolAmountOut              \\    pS    /     \(wI / tW)//                           //
// bI = balanceIn          tAi =  -----------------------------------------                  //
// wI = weightIn                           /     /      wI  \       \                       //
// tW = totalWeight                       | 1 - |  1 - ----  |  * sF |                      //
// sF = swapFee                            \     \      tW  /       /                       //
**********************************************************************************************/
```

Recommendation

Evaluate which of the formulas is the correct one and adjust the code-comment or the implementation respectively.

Status

Swaap Labs addressed the issue by changing the code-comment to reflect the formula of the implementation in commit id: `9fa88d2`

B06: Discrepancy between code comment and implementation in Math.calcSpotPrice

[Severity: Informative | Difficulty: n/a| Category: Code Readability]

Description

There is discrepancy between a code comment above the `Math.calcSpotPriceMMM` function (Math.sol#L422) and its implementation.

The comment states the the spot price is calculated according to the formula:

```
price = (balance_in * weight_out) / (balance_in * weight_out)
```

But it should be the following formula:

```
price = (balanceIn * weightOut) / (balanceOut * weightIn)
```

Recommendation

Change the code comment to reflect the accurate spot price formula.

Status

Swaap Labs addressed the issue in commit id: `9fa88d2`

B07: Protect privileged accounts against invalid addresses

[Severity: Low | Difficulty: High | Category: Security]

Description

There are two account types with elevated privileges in the protocol. The _swaapLabs-account of the Factory-contract and the controller-account of a pool. See Security Model for a summary of the privileges and responsibilities that come with these accounts. Either account type is allowed to transfer the privileges to another account via the `setSwaapLab`-s and `setController`-methods respectively. Both transfers lack sanity checks that could prevent the irreversible loss of privileges. First, the setSwaapLabs-account lacks a check against the zero-address. Second, both transfers do not check if the target account is existent and willing to accept the privileges.

Scenario A

Imagine that the _swaapLabs account wants to transfer its privileges to another account via the `setSwaapLabs`-method. The account holder uses an off-chain tool for that purpose. However, the tool fails to set the parameters correctly, and calls setSwaapLabs with the zero address. The privileges of the _swaapLabs account are now permanently lost.

Scenario B

Imagine that the _swaapLabs account holder wants to transfer its privileges to another account, but he made a copy-and-paste error and accidentally inserted the wrong address. Again, the privileges are lost permanently. The same scenario also applies to the controller account of the Pool-contract.

Recommendation

1. To protect against scenario A: Add a sanity check against the zero-address to the setSwaapLabs-method and revert in case of failure.
2. To protect against scenario B: Split the transfer of privileges for all account types into two phases: In the first phase the current holder of the privileges offers to transfer them to the new account. Second, the potential receiver of the privileges needs to accept the offer. As long as the receiver account hasn't accepted the offer, the original account should be able to withdraw the offer or make a new offer to a different account.

Status

Swaap Labs addressed the issue by following the second recommendation in commit-id: `29acee2`

B08: Safe gas by removing superfluous require clauses

[Severity: Informative | Difficulty: n/a| Category: Gas Optimization]

Description

The following require-clauses can never be false because unsigned integers cannot be negative. There is no deeper problem with that, other than a small amount of gas spent on the operations.

Pool.sol#L234

```
require(swapFee >= 0, "16");
```

Pool.sol#674

```
require(_dynamicCoverageFeesZ >= 0, "20");
```

Recommendation

Remove the superfluous checks. In the mid term consider using a statical analysis tool, like Slither, to detect these tautologies automatically.

Status

Swaap Labs addressed the issue in commit id: `d83ef4d`

B09: Safe gas by removing superfluous re-entrancy locks

[Severity: Informative | Difficulty: n/a | Category: Gas Optimization]

Description

Some functions are decorated with the `_lock_` modifier even though they cannot trigger external calls, e.g. `Pool.setSwapFee` or `Pool.setController` as well as other setters. The modifiers are not harmful, but they also don't have a utility on functions that don't allow external calls. Therefore, removing those modifiers can save some gas costs. Notice that the modifier contains expensive storage writing and reading operations.

On the other hand, these locks can be seen as a defensive programming technique and protect from a scenario where a developer adds an external call to one of the functions in the future and forgets to add the lock manually.

Recommendation

Consider the trade-off between gas-savings and defensive programming. If you decide to remove the re-entrancy locks we recommend using a static analysis tool, like Slither, to detect external calls and possible re-entrancy attacks.

Discussion

Swaap Labs is going to deploy to Polygon where gas-costs are currently lower than on Ethereum mainnet.

Status

Swaap Labs decided to prioritize defensive programming over the gas costs reduction.

B10: Add input validation to Pool.joinPool and Pool.exitPool

[Severity: Informative | Difficulty: n/a | Category: Input Validation]

Description

The `joinPool` and `exitPool` functions take parameters to limit the maximum (resp. minimum) amount of tokens that the user is willing to deposit (resp. withdraw). The user has to provide one limit per bound token. If he provides fewer limits both functions will revert (due to invalid array access in the loop's body). However, if he provides more limits, the functions will run to completion. The problem here is that a mismatch of the number of limits and tokens indicates a mistake made by the user, and the functions should revert.

Recommendation

Add additional input validations on the joinPool and exitPool functions and revert if the number of limits supplied by the user does not match the number of tokens bound to the pool.

Status

Swaap Labs addressed the issue in commit id: `3852e53`

B11: Add sanity checks to Pool.finalize

[Severity: Informative | Difficulty: n/a | Category: Input Validation]

Description

Pools can have at least 2 tokens and at most 8 tokens bound to it. The former side-condition is checked during Pool.finalize while the second one is checked during Pool.bindMMM.

Recommendation

We recommend adding a sanity check to `Pool.finalize` for the maximum number of bound tokens as well. This is a defensive programming technique and should protect from the case that a future revision of the contract adds a new method to bind tokens and forgets to check the side-condition. Moreover, it protects against a scenario where a clever user finds a way to bind more than 8 tokens that we couldn't see. The deeper problem with pools of more than 8 tokens is that they can become inoperable due to the block-gas limit.

Related

See A05 for a related issue on how a controller can bypass the lower bound of token.

Discussion

Swaap Labs assess that such scenarios are unlikely to occur.

Status

Swaap Labs acknowledges the issue. No on-chain solution is applied.

B12: Potential precision optimization in Math.calcSpotPrice

[Severity: Low | Difficulty: n/a | Category: Arithmetic Precision]

Description

The spot price calculation can be modified to yield a higher precision in the average case:

$$sP = \frac{bI/wI}{bO/wO} * \frac{1}{(1 - sF)}$$

With: $wI \neq 0$, $bO \neq 0$, $wO \neq 0$ and $sF \neq 1$

$$= \frac{bI * wO}{wI * bO * (1 - sF)}$$

With: $wI \neq 0$, $bO \neq 0$, $sF \neq 1$. Notice that wO can be 0 now

The first equation shows the formula that is currently implemented, the simplified equation can be more precise in the average case since it uses one less rounding operation. Also, notice that the simplified formula is closer to the definition from the whitepaper. In particular it is defined for the case that `w0 = 0`, which is not the case for the first formula.

Status

Swaap Labs addressed the issue in commit id: `178c34c`

B13: Potential precision optimization in Math.calcPoolOutGivenSingleIn

[Severity: Low | Difficulty: n/a | Category: Arithmetic Precision]

Description

The liquidity provision function Malc.calcPoolOutGivenSingleIn can be modified to yield a higher precision in the average case:

$$sP = \left(\left(\frac{tAi * \left(1 - \left(\left(1 - \frac{wI}{tW}\right) * sF\right)\right) + tBi}{tBi}\right)^{\frac{wI}{tW}}\right) * pS - pS$$

$$= \left(\left(\frac{tAi * \left(1 - \left(\frac{(tW - wI) * sF}{tW}\right)\right) + tBi}{tBi}\right)^{\frac{wI}{tW}}\right) * pS - pS$$

$$= \left(\left(\frac{tAi * \left(\frac{tW - (tW - wI) * sF}{tW}\right) + tBi}{tBi}\right)^{\frac{wI}{tW}}\right) * pS - pS$$

$$= \left(\left(\frac{\frac{tAi * (tW - (tW - wI) * sF)}{tW}}{tBi} + 1\right)^{\frac{wI}{tW}}\right) * pS - pS$$

$$= \left(\left(\frac{tAi * (tW - (tW - wI) * sF)}{tBi * tW} + 1\right)^{\frac{wI}{tW}}\right) * pS - pS$$

The first equation shows the formula that is currently implemented, the final equation can be more precise in the average case since it uses one less rounding operation.

Status

Swaap Labs addressed the issue in commit id: `b449bec`

B14: Potential precision optimization in Math.calcSinlgeInGivenPoolOut

[Severity: Low | Difficulty: n/a | Category: Arithmetic Precision]

Description

The liquidity provision function Malc.calcSinlgeInGivenPoolOut can be modified to yield a higher precision in the average case:

$$tAi = \frac{\left(\left(\frac{ps + pAo}{pS}\right)^{\frac{1}{wI/tW}}\right) * bI - bI}{1 - \left(\left(1 - \frac{wI}{tW}\right) * sF\right)}$$

$$= \frac{\left(\left(\frac{ps + pAo}{pS}\right)^{\frac{tW}{wI}}\right) * bI - bI}{1 - \left(\frac{(tW - wI) * sF}{tW}\right)}$$

$$= \frac{\left(\left(\frac{ps + pAo}{pS}\right)^{\frac{tW}{wI}}\right) * bI - bI}{\frac{tW - (tW - wI) * sF}{tW}}$$

$$= \frac{\left(\left(\left(\frac{ps + pAo}{pS}\right)^{\frac{tW}{wI}}\right) * bI - bI\right) * tW}{tW - (tW - wI) * sF}$$

$$= \frac{\left(\left(\frac{ps + pAo}{pS}\right)^{\frac{tW}{wI}} - 1\right) * bI * tW}{tW - (tW - wI) * sF}$$

The first equation shows the formula that is currently implemented, the final equation can be more precise in the average case since it uses one less rounding operation.

Status

Swaap labs removed that function from the code base.

B15: Potential precision optimization in Pool.calcSingleOutGivenPoolIn

[Severity: Low | Difficulty: n/a| Category: Arithmetic Precision]

Description

The liquidity provision function Malc.calcSingleOutGivenPoolIn can be modified to yield a higher precision in the average case:

$$tAo = \left(bO - \left(\left(\frac{pS - (pAi * (1 - ef))}{pS} \right)^{\frac{1}{wI/tW}} * bO \right) \right) * \left(1 - \left(1 - \frac{wO}{tW} \right) * sF \right)$$

$$= \left(bO - \left(\left(\frac{pS - (pAi * (1 - ef))}{pS} \right)^{\frac{tW}{wI}} * bO \right) \right) * \left(1 - \left(1 - \frac{wO}{tW} \right) * sF \right)$$

The first equation shows the formula that is currently implemented, the final equation can be more precise in the average case since it uses one less rounding operation.

Status

Swaap Labs addressed the issue in commit id: `2eacdcc`

B16: Duplicated code in GeometricBrownianMotionOracle

[Severity: Informative | Difficulty: n/a | Category: Best Practice]

Description

The function `getRoundData` is defined twice, once in `GeometricBrowniadMotionOracle` and once in `ChainlinkUtils`. This makes it harder to maintain the codebase in the case that the function needs to be updated in the future.

Recommendation

Remove the function from GeometricBrownianMotionOracle and import it from ChainlinkUtils.

Status

Swaap Labs removed the duplicate code from GeometricBrownianMotionOracle in commit-id: `0d6e385`

B17: Sanitize precision of prices obtained from oracles

[Severity: Low | Difficulty: n/a | Category: Best Practice]

Description

At the moment getRoundData returns the price with the respective precision of the price feed (most commonly 8 or 18 decimals). That leads to some mixed-precision arithmetic calculations on the calling side of the function. As a consequence, the calling side must ensure that the mixed precision is executed correctly. This approach is tedious and increases the chance of introducing defects.

Recommendation

Instead of handling the mixed precision at the calling side, we recommend converting the price to 18-decimal precision in the body of the `getRoundData` function. This is a defensive programming technique to reduce the risk that the mixed case is not treated appropriately on the calling side.

Status

Swaap Labs acknowledges the recommendations but decided to leave the implementation as it is since it has been tested.

Appendix A - Issue Classification

[version: 0.1.0]

Issue Severity/Difficulty Classification

Runtime Verification's issue ranking system is based on two axes, severity and difficulty. Severity covers "how bad would it be if someone exploited this", and is ranked Informative, Low, Medium, or High. Difficulty is "how hard is it for someone to exploit this", and is ranked Low, Medium, or High.

This document is a snapshot of a constantly changing guidance for security rating. The lead auditor reserves the right to change severity or difficulty ratings as needed for each situation.

Severity Ranking

Severity refers to how bad it is if this issue is exploited. This means that the effects of the exploit affect the severity, but who can do the exploit does not.

If a given attack seems to fit multiple criteria here, we use the most severe classification.

High Severity

- Permanent deadlock of some or all protocol operations.
- Loss of any non-trivial amount of user or protocol funds.
- Core protocol properties do not hold.
- Arbitrary minting of tokens by untrusted users.
- DOS attacks making the system (or any vital part of the system) unusable.

Medium Severity

- Sensible or desirable properties over the protocol do not hold, but no known attack vectors due to this ("looks risky" feeling).
- Non-responsive or non-functional system is possible, but recovery of user funds can still be guaranteed.
- Temporary loss of user funds, guaranteed to be recoverable via an external algorithmic mechanism like a treasury.
- Loss of small amounts of user funds (eg. bits of gas fees) that serve no protocol purpose.
- Griefing attacks which make the system less pleasant to interact with, potentially used to promote a competitor.

- System security relies on assumptions about externalities like "valid user input" or "working monitoring server".
- Deployments are not verifiable, so that phishing attacks may be possible.

Low Severity

- Slow processing of user transactions can lead to changed parameters at transaction execution time.
- Function reverts on some inputs that it could safely handle.
- Users receive less funds than expected in pure mathematical model, but bounds on this error is very small.
- Users are not protected from obviously bad choices (eg. trading into an asset with zero value).
- System accumulates dust (eg. due to rounding errors) that is unrecoverable.

Informative Severity

- Not following best coding practices. Examples include:
- Missing input validation or state sanity checks,
- Code duplication,
- Bad code architecture,
- Unmatched interfaces or bad use of external interfaces,
- Use of outdated or known problematic toolchains (eg. bad compiler version),
- Domain specific code smells (eg. not recycling storage slots on EVM).
- Gas optimizations.
- Non-intuitive or overly complicated behaviors (which may lead to users and/or auditors mis-understanding the code).
- Lack of documentation, or incorrect/inconsistent documentation.
- Known undesired behaviors when the security model or assumptions do not hold.

Difficulty Ranking

Difficulty refers to how hard it is to actually accomplish the exploit. The things that increase difficulty are how expensive the attack is, who can perform the attack, and how much control you need to accomplish the attack. Note that when analyzing the expense difficulty of an attack, flash loans are taken into account.

If an attack fits multiple categories here, because of factors X which makes it difficulty D1 and Y which makes difficulty D2, then difficulty is assigned as follows:

Are both X and Y necessary to make the attack happen, we assign higher difficulty.

If only one of X and Y is necessary, then use the lower difficulty.

High Difficulty

- Only trusted authorized users can perform the attack (eg. core devs).
- Performing the attack costs significantly more than how much you benefit (eg. it costs 10x to do the attack vs what is actually won).
- Performing the attack requires coordinating multiple transactions across different blocks, and can be stopped if detected early enough.
- Performing the attack requires control of the network, to delay or censor given messages.
- Performing the attack requires convincing users to participate (eg. by bribing the users).

Medium Difficulty

- Semi-authorized (or whitelisted) users can perform the attack (eg. "special" nodes, like validators, or staking operators).
- Performing the attack costs close to how much you benefit (eg. 0.5x - 2x).
- Performing the attack requires coordinating multiple transactions across different blocks, but cannot be stopped if detected early enough.

Low Difficulty

- Anyone who can create an account on the network can perform the attack.
- Performing the attack costs much less than how much you benefit (eg. < 0.5x).
- Performing the attack can happen within a single block or transaction (or transaction group).
- Performing the attack only requires access to a modest amount of capital and a flash-loan system.