PUBLIC

Code Assessment

of the Core Engine Smart Contracts

Oct 11, 2021

Produced for

by

Contents

1 Executive Summary

Dear Sir or Madam,

First and foremost we would like to thank Primitive Finance for giving us the opportunity to assess the current state of their Core Engine system. This document outlines the findings, limitations, and methodology of our assessment.

Primitive Finance's team was very responsive and professional. We found multiple issues which were addressed. The remaining issues are listed in this report.

We hope that this assessment provides more insight into the current implementation and provides valuable findings. We are happy to receive questions and feedback to improve our service and are highly committed to further support your project.

Sincerely yours,

ChainSecurity

1.1 Overview of the Findings

Below we provide a brief numerical overview of the findings and how they have been addressed.

Critical-Severity Findings	0
High-Severity Findings	7
• Code Corrected	3
• Specification Changed	4
Medium-Severity Findings	7
• Code Corrected	4
• Specification Changed	3
Low-Severity Findings	14
• Code Corrected	8
• Specification Changed	1
• Code Partially Corrected	1
• Acknowledged	4

2 Assessment Overview

In this section we briefly describe the overall structure and scope of the engagement including the code commit which is referenced throughout this report.

2.1 Scope

The assessment was performed on the source code files inside the Core Engine repository based on the documentation files. The table below indicates the code versions relevant to this report and when they were received.

V	Date	Commit Hash	Note
1	13 Aug 2021	9c2111fd31d4faaabb5b1606ff5668ad11af7261	Initial Version
2	28 Aug 2021	5b418337e8bd00cd59766d1251931b9094cf0353	Version 2
3	12 Sep 2021	82d01cd0030fab67780fbb5bef40b6748ff24644	Version 3
4	26 Sep 2021	c44f8ca1ee43772ec4ba56e031bdb6fb8aac0ca3	Version 4

For the solidity smart contracts, the compiler version `0.8.6` was chosen.

2.1.1 Excluded from scope

Explicitly excluded is the ABDK math library `ABDKMath64x64`, the underlying theoretical framework that includes the economics and math (convergence, correctness of the functions and solutions, etc.) behind the project.

2.2 System Overview

This system overview describes the initially received version (Version 1) of the contracts as defined in the Assessment Overview. Changes are tagged with the corresponding version tag. Furthermore, in the findings section we have added a version icon to each of the findings to increase the readability of the report.

Liquidity providers on current exchanges like Uniswap or Curve have a pay-off profile that depends on the used trading function (curve). Instead of a trading function dependent pay-off function, Primitive finance offers liquidity providers a specific pay-off function and adjusts the trading function to mimic to a desired pay-off function. The desired pay-off function is a covered call for liquidity providers. Version 1 / Version 2 Additionally, users can borrow the liquidity tokens from liquidity providers by paying the price for the stable part of the liquidity token in the underlying (risky token). The borrower can trade buy back the risky share of the liquidity token by providing stable tokens (like paying the price for the underlying in a long call).

The price of the options is determined by the liquidity pool's token composition. The trading function offers profitable trades for arbitrageurs so that they rebalance the pool, resulting in the target pay-off functions.

The target pay-off functions are (1) a covered call for a normal liquidity provider and Version 1 / Version 2 (2) a long call for a liquidity token borrower. The corresponding trading function (invariant) is

$$\psi V(R_1, R_2) = R_2 - K\Phi(\Phi^{-1}(1-R_1) - \sigma\sqrt{\tau})$$

for ≤0 and negative infinity otherwise. R is the reserve value of the respective coins 1 (risky) and 2 (stable).

This invariant is used to calculate the swap amounts and consecutively the swap price. The trade updates the reserve amounts and, hence, the liquidity provider should always get the pay-off when withdrawing their liquidity tokens as if they would have a covered call position. Version 1 / Version 2 The same should hold for a borrower of a liquidity token who sells of the stable part for risky tokens. Their pay-off function should be a long call. Details are described in the following papers (1) https://stanford.edu/~guillean/papers/rmms.pdf describing that the trading function can mimic a pay-off function for a covered call and (2) https://stanford.edu/~guillean/papers/cfmm-lending.pdf describing that borrowing the liquidity position should result in a long call pay-off (Version 2) or long put.

The system has two main contracts. The `PrimitiveFactory.sol` which can deploy one `PrimitiveEngine.sol` per token pair. The `PrimitiveEngine.sol` can open multiple pools/option markets per token pair. E.g. different strike prices and maturities. Both contracts are permissionless in the sense that there is no permissioned role with power. Still, there is one factory owner with no special power set at the factory's deployment.

The `PrimitiveFactory.sol` has the callable function `deploy` to deploy a new Engine for a token pair. The `PrimitiveEngine.sol` offers users the following functionality:

1. `create` : Creates a new option contract / trading function with provided parameters. Simultaneously, the new pool needs to be funded and the initial invariant is calculated.
2. `deposit` and `withdraw` : Add funds to a margin account that can be used to save gas later.
3. `allocate` and `remove` : Allocates funds from a margin account or a transfer to a specified pool / option or removes the funds. By calling `allocate`, the caller becomes a liquidity provider opening a covered call position. When calling `remove` the caller closes their position and receives the current value of the covered call option. The value is payed in the share of the current pool composition (that should be balanced to the correct option value by arbitrageurs who constantly balance the pool.
4. `supply` and `claim` : A liquidity provider can put up his position to be borrowed by another user by calling `supply`. `claim` undoes this under certain restrictions.
5. `borrow` and `repay` : Version 1 When calling `borrow` a user pays buys the stable part of a liquidity token that has been supplied to be borrowed by a liquidity provider in risky token. By calling `repay`, the user can buy risky token for stable tokens. This mimics the pay-off scenario of buying a long call. Version 2 When calling `borrow` the user can borrow liquidity tokens and sell the stable share of it for risky or vice versa. Depending on which share of the liquidity position was payed, the user can call `repay` to buy the collateral for the other token. This now mimics either a call or a put pay-off.
6. `swap` : Crucial for the system to work is, that the pool has the correct composition of risky and stable tokens to mimic the pay-off for the two different investors. The trading function changes the price for a pool asset if time progresses or an asset price changes. This opens arbitrage opportunities which should balance the pool to the correct composition at any time (given some frictions occurring through fees, transaction costs etc.). The swap function simply converts a provided amount of stable tokens to risky and vice versa (incl. a small fee). The price for an asset is determined by the solving the invariant accordingly.
7. The remaining functions are view functions.

Version 1 A `0.0015` percent fee is charged at each swap. The fee is taken from the funds transferred in. The code simply assumes in the calculation it received `0.0015` less tokens. The reserve still accounts for the full 100 percent. Hence, when the reserve pays out funds for position tokens, the fees should also payout out proportionally.

Version 2 The fee structure was changed. Liquidity provide that lend their positions to borrower bear a risk and costs that they should be compensated for (which they were not in the old version).

The system is intended to be used by smart contracts and, therefore, has optimistic transfers with callback hooks and pre and post condition checks. The functions `repay` (Version 1 / Version 2), `deposit`, `withdraw` and `allocate` can be used with a specified receiver.

In versions above version two the supply/claim and borrow/repay functionality was removed completely.

3 Limitations and use of report

Security assessments cannot uncover all existing vulnerabilities; even an assessment in which no vulnerabilities are found is not a guarantee of a secure system. However, code assessments enable discovery of vulnerabilities that were overlooked during development and areas where additional security measures are necessary. In most cases, applications are either fully protected against a certain type of attack, or they are completely unprotected against it. Some of the issues may affect the entire application, while some lack protection only in certain areas. This is why we carry out a source code assessment aimed at determining all locations that need to be fixed. Within the customer-determined time frame, ChainSecurity has performed an assessment in order to discover as many vulnerabilities as possible.

The focus of our assessment was limited to the code parts associated with the items defined in the engagement letter on whether it is used in accordance with its specifications by the user meeting the criteria predefined in the business specification. We draw attention to the fact that due to inherent limitations in any software development process and software product an inherent risk exists that even major failures or malfunctions can remain undetected. Further uncertainties exist in any software product or application used during the development, which itself cannot be free from any error or failures. These preconditions can have an impact on the system's code and/or functions and/or operation. We did not assess the underlying third party infrastructure which adds further inherent risks as we rely on the correct execution of the included third party technology stack itself. Report readers should also take into account the facts that over the life cycle of any software product changes to the product itself or to its environment, in which it is operated, can have an impact leading to operational behaviours other than initially determined in the business specification.

4 Terminology

For the purpose of this assessment, we adopt the following terminology. To classify the severity of our findings, we determine the likelihood and impact (according to the CVSS risk rating methodology).

- *Likelihood* represents the likelihood of a finding to be triggered or exploited in practice
- *Impact* specifies the technical and business-related consequences of a finding
- *Severity* is derived based on the likelihood and the impact

We categorize the findings into four distinct categories, depending on their severities. These severities are derived from the likelihood and the impact using the following table, following a standard risk assessment procedure.

Likelihood	Impact		
	High	Medium	Low
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

As seen in the table above, findings that have both a high likelihood and a high impact are classified as critical. Intuitively, such findings are likely to be triggered and cause significant disruption. Overall, the severity correlates with the associated risk. However, every finding's risk should always be closely checked, regardless of severity.

5 Findings

In this section, we describe any open findings. Findings that have been resolved, have been moved to the Resolved Findings section. All of the findings are split into these different categories:

- Security: Related to vulnerabilities that could be exploited by malicious actors
- Design: Architectural shortcomings and design inefficiencies
- Correctness: Mismatches between specification and implementation

Below we provide a numerical overview of the identified findings, split up by their severity.

Critical-Severity Findings	0
High-Severity Findings	0
Medium-Severity Findings	0
Low-Severity Findings	5

5.1 Code Duplication Balance Getters

Design Low Version 1 Acknowledged

The engine contract implements two functions: `balanceRisky()` and `balanceStable()` which return the balance of the engine for the respective token. These two functions implement the same functionality and have the same logic, therefore can be merged into a single function that takes the token address (for stable or risky) as an input parameter.

Acknowledged

The client prefers two separate functions.

5.2 Event Optimization

Design Low Version 1 Acknowledged

The `swap()` function emits two events: `UpdatedTimestamp()` and `Swap()` which can be integrated into one event to reduce the gas consumption.

Acknowledged

This behavior is desired by Primitive.

5.3 Sanity Checks

Design Low Version 1 Code Partially Corrected

When depositing, withdrawing, allocating, and swapping (VERSION4) the receiving account can be chosen. No basic sanity check if it is accidentally address zero is performed. Additionally, the strike price could be validated if it is not zero in `create`.

Code partially corrected

A sanity check for the strike price in `create` is implemented but no checks for address zero are added.

5.4 Unused Function `getRiskyGivenStable`

Design Low Version 1 Acknowledged

The function `getRiskyGivenStable` is declared `internal` but not used in the code.

Acknowledged

Primitive Finance acknowledged the issue but communicated that the function is kept as it is for now.

5.5 Unused Storage Fields

Design Low Version 1 Acknowledged

The `PrimitiveEngine` contract is deployed by the `PrimitiveFactory` contract. The engine stores the factory address as state variable `address public immutable override factory;` and an `owner` but these variables are not used.

Acknowledged

Primtive acknowledged the behavior and informed us that this is intended.

6 Resolved Findings

Here, we list findings that have been resolved during the course of the engagement. Their categories are explained in the Findings section.

Below we provide a numerical overview of the identified findings, split up by their severity.

Critical-Severity Findings	0
High-Severity Findings	7

- Low Decimal Token Issues Code Corrected
- Anyone Can Call the Repay Function After Pool's Maturity Specification Changed
- Borrower Locks Liquidity in the Pool Specification Changed
- Disable Unnecessary Functionalities After Expiry Code Corrected
- Flawed Fee and Premium Structure Specification Changed
- No Slippage Protection Code Corrected
- Violation of Maximum Ratio of Float Liquidity Specification Changed

Medium-Severity Findings	7

- Token Decimal Validation Code Corrected
- Explicitly Handling Positive Invariant Restriction Specification Changed
- Incorrect Tracking of Cumulative Values for Pool Reserves Code Corrected
- Liquidity Providers Get Rewards Without Supplying Float Liquidity Specification Changed
- Possible Overflows Code Corrected
- Possible to Frontrun on Claim Request Specification Changed
- Redundant and Improper Revert Condition Code Corrected

Low-Severity Findings	9

- Inconsistency of Input Parameters Code Corrected
- Unused Error Definition Code Corrected
- Redundant Storage Read Specification Changed
- Code Duplication Code Corrected
- Duplicated Calculation of Invariant Code Corrected
- Implementation of getStableGivenRisky Function Code Corrected
- Possible Gas Optimization in the Deposit Function Code Corrected
- Return Value in safeTransfer Code Corrected
- MANTISSA_INT Constant Code Corrected

6.1 Low Decimal Token Issues

Design High Version 3 Code Corrected

The lower the decimals of a token are and the higher the value, the more severe rounding issues will become. Simultaneously, the liquidity position accounting with 18 decimals will cause issues.

Examples issues are:

Burning one unit of a low decimal but high value token (in `create`) might be a huge loss for the user.

There is a dependency between the delta when creating a pool and the decimals of a token. `delta` cannot be chosen freely because of this dependency `(1e18 - delta) - 1e(18-decimals)` needs to be bigger than `1`, else the create will revert. This basically eliminates the support of zero decimal tokens (as the only viable option is `delta = 0`). Low decimals limit the range of `delta` and the step accuracy with which the risky token amount is calculated. E.g. the maximum value of `delta` can be `9e17` (should be `1e18`) for `1` decimals, `99e16` for `2` decimals and so on. The step size should be accordingly high to increase the resulting `delRisky` one unit.

With decreasing decimals this calculation will lose precision if the token decimals are not dividable by the fraction `delLiquidity / PRECISION` with modulo zero.

```
delRisky = (delRisky * delLiquidity) / PRECISION; // liquidity has 1e18 precision, delRisky has native precision
delStable = (delStable * delLiquidity) / PRECISION;
```

The function `getAmounts` has a similar problem and will losing precision. This can be exploited in `allocate` to receive more liquidity tokens than the user would be entitled to as the rounding in `allocate` is in the user's favor.

Code corrected

The issues above have been tackled by only accepting token decimals between six and 18. Six was chosen to support famous stable coins and did not lead to issues in tests. However, tests and fuzzing does not cover all possible states and due to the complex math, there might be a state that still results in issues regarding to the decimals.

6.2 Anyone Can Call the Repay Function After Pool's Maturity

Design | High | Version 1 | Specification Changed

After the pool's maturity has passed, i.e., the pool has expired, anyone can call the `function repay()` for any borrower and receive their premiums in case the borrower's possition yields profit. The first three lines of the `function repay()` allow any `msg.sender` to receive the premiums for any borrower:

Since the borrower is incentivized to call the `function repay()` only when there is profit, the function allows any attacker to collect the unclaimed profit of any borrower after the pool's maturity. Furthermore, if the legitimate borrwers calls the `function repay()` at the maturity of the pool, the attacker has still a possibility to frontrun the legitimate transaction.

Version 2 **Specification changed**

This version of the code introduces a grace period, around 24h, to permit only borrowers calling the function `repay()` after pool's maturity. In case a borrower does not call the function during this period, anyone can call the function `repay()` and exit the borrowers' positions, therefore releasing the locked liquidity.

Version 3 **Specification changed**

The respective code has been removed according to the new specifications of Version 3.

6.3 Borrower Locks Liquidity in the Pool

Design High Version 1 Specification Changed

Calling the `function borrow()` with a given `delLiquidity` triggers the call `reserve.borrowFloat()` which decreases the amount of available float in the pool and increases the debt of the pool reserve accordingly. This way, the borrower locks `delLiquidity` from the available float in the pool reserve. Below is the `borrowFloat` function.

```
function borrowFloat(Data storage reserve, uint256 delLiquidity) internal {
        reserve.float -= delLiquidity.toUint128();
        reserve.debt += delLiquidity.toUint128();
}
```

A liquidity provider that has supplied its liquidity as float needs to first call the `function claim()` which converts the float into liquidity before removing the liquidity from the pool. However, the only way for all liquidity providers to claim all their float liquidities is if all borrowers call the `function repay()` which triggers a call to `reserve.repayFloat()`:

```
    function repayFloat(Data storage reserve, uint256 delLiquidity) internal {
    reserve.float += delLiquidity.toUint128();
    reserve.debt -= delLiquidity.toUint128();
}
```

But, if the price of the risky token is below the strike price, the borrower has no incentive to call the `function repay()`, therefore potentially keeping locked the float liquidity. Moreover, the function `repay()` does not impose any time restriction to borrowers when they can exercise their option, i.e., the borrower can potentially call the `repay()` function at an arbitrary time after the maturity. This puts pressure on the liquidity providers to call it themselves which is possible because in the current version of the `PrimitiveEngine` contract, anyone can call the `function repay()` after the maturity of the pool. If liquidity provider have the burden to call the functions, they also bear the costs.

Version 2 **Specification changed**

This version of the code introduces a grace period, around 24h, to permit only borrowers calling the function `repay()` after pool's maturity. In case a borrower does not call the function during this period, anyone can call the function `repay()` and exit the borrowers' positions, therefore releasing the locked liquidity. Still, the additional costs need to be payed by the caller / LP if they call it to relase their funds.

Version 3 **Specification changed**

The respective code has been removed according to the new specifications of Version 3.

6.4 Disable Unnecessary Functionalities After Expiry

Design High Version 1 Code Corrected

In the current version of the PrimitiveEngine contract all functions, except `function swap()`, can be called after the pool has expired. For example, a liquidity provider could close the opened positions of borrowers, claim its share of the float liquidity, and then call `function borrow()` for the remaining amount of float.

Code corrected

The updated version of the contract has new checks if the pool is still valid in the respective functions.

6.5 Flawed Fee and Premium Structure

Design High Version 1 Specification Changed

When swapping, a fee is charged and borrows pay a premium on their position. Both amounts end up in the pool without separate accounting. This has various implication. All arising from the fact that the underlying calculation are based on the pool's reserve, which includes the fees and premiums. All operations that pay out a proportion of the reserve amounts - also pay out parts of the fees and premiums. Hence, each time `remove`, `repay`, `swap` or `borrow` is called, fees and premiums are payed out. Regardless of the callee is entitled to receive these fees.

The most severe issue is the `swap` function. Swapping on the pool's reserve, which includes the collected fees, will nullify all previous fees and prevent fee accumulation. Hence, liquidity providers will not earn fees collected during the lifetime of the pool, but only the fee from the last swap.

Other examples for issues are shared (even with non-eligible users) premiums and fees, participating on fees and premiums repeatedly. E.g. a liquidity provider that does not take the risk of lending their token gets a share of the premium. A borrower gets part of the premium and fees of others. All this can be leveraged through repeating the operation.

Version 2 **Specification changed**

This version of the code introduces a novel fee structure.

Version 3 **Specification changed**

The respective code has been updated according to the new specifications of Version 3 which assume fees only during swaps.

6.6 No Slippage Protection

Design High Version 1 Code Corrected

All transactions have a lag between the time they are sent and the time they are executed as they remain in the mem pool for some time prior to being executed. Between sending and execution, other transaction might change the contract's state. This is critical for all transaction where the user receives or has to pay funds. In Version 1 all action function in the Engine contract except for `supply` and `claim` do not offer any protection against slippage. In VERSION4 this affects `allocate`, `remove`, and `swap`. Without checking if the transaction is still executed under the desired conditions, the user may suffer losses.

This issue can be maliciously exploited by front running certain transactions. However, as the system is designed to interact with smart contracts, the slippage protection could be implemented on their side.

Code corrected

The user is now able to define the delta in and delta out when swapping. Hence, the user either gets the defined values or the swap will revert due to a violation of the invariant check. The check verifies that the invariant can only increase.

6.7 Violation of Maximum Ratio of Float Liquidity

Security High Version 1 Specification Changed

The amount of liquidity supplied as float should be less than a threshold value, hard coded to 80% in the current version of the contract. This restriction is enforced in the `function addFloat()` as follows:

```
function addFloat(Data storage reserve, uint256 delLiquidity) internal {
        reserve.float += delLiquidity.toUint128();
        if ((reserve.float * 1000) / reserve.liquidity > 800) revert LiquidityError();
    }
```

This restriction is enforced only when a liquidity provider supplies its liquidity as float, but it does not hold always as any liquidity provider can freely remove available liquidity from the pool. For example, if the float liquidity is at its maximum level (i.e., 80%), one liquidity provider could call the `function remove()` to remove the 20% of the remaining liquidity, thus putting the pool reserve in a state with 100% of its liquidity as float.

```
function remove(
        Data storage reserve,
        uint256 delRisky,
        uint256 delStable,
        uint256 delLiquidity,
        uint32 blockTimestamp
    ) internal {
        reserve.reserveRisky -= delRisky.toUint128();
        reserve.reserveStable -= delStable.toUint128();
        reserve.liquidity -= delLiquidity.toUint128();
        update(reserve, blockTimestamp);
    }
```

Version 2 **Code corrected**

The `Reserve` library now defines a function `checkUtilization()` which checks if the invariant holds whenever float is added, float is payed, or the liquidity is removed.

Version 3: **Specification changed** The respective code has been removed according to the new specifications of Version 3.

6.8 Token Decimal Validation

Design Medium Version 3 Code Corrected

The engine contract supports tokens with different decimals. However, tokens with very few decimals and more than 18 decimals cause severe issues but are allowed to be deployed by the factory.

Code corrected

The factory now validates decimals for both tokens before deploying an engine. Tokens with 6 to 18 decimals are supported.

6.9 Explicitly Handling Positive Invariant Restriction

Design Medium Version 1 Specification Changed

The current implementation checks that the invariant grows assuming it is negative and, when updated by a swap, grows closer to zero: `if (invariant > nextInvariant && nextInvariant.sub(invariant) >= Units.MANTISSA_INT)`.

A zero invariant implies a balanced pool at the time of the swap. Typically, the invariant should not become positive but could happen in specific scenarios such as high trading frequency and high fee accumulation. If this is the case, besides major other problems, the pool cannot recover as the invariant needs to decrease back to zero to be in balance again. This is because of the check, that the invariant is only allowed to grow after a trade. However, in case of a positive invariant, it should not become more positive.

Version 2: **Code changed**

The updated version of the code checks explicitly if the invariant is positive and prevents the invariant to grow in the wrong direction.

Version 3: **Specification and code changed** The issue theoretically exists in version 3. However, according to Primitive Finance, a positive invariant is not an undisired state any more and a swap should not lead to a decreasing invariant - even if it is positive.

6.10 Incorrect Tracking of Cumulative Values for Pool Reserves

Design Medium Version 1 Code Corrected

Primitive Finance pointed out this issues while the audit was ongoing. They are aware that calling the function `update()` after the pool reserve values are updated, results in incorrect cumulative values.

Code corrected

The function `update()` is called before the new amounts have been applied to the pool reserves.

6.11 Liquidity Providers Get Rewards Without Supplying Float Liquidity

Design Medium Version 1 Specification Changed

In order for users to borrow liquidity from the pools, liquidity providers should allocate liquidity to a pool and then supply it as float which can be borrowed. Users pay a premium when borrowing liquidity and the float liquidity of the pool reserve is deducted. After the borrowers pay their debt, the liquidity providers should claim at first their share of liquidity as float, and then remove it from the reserve.

Since the liquidity providers do not get tokens for their supplied liquidity, the premiums paid by borrowers go to the pool reserve. This way, all liquidity provider get tokens out according to their share of liquidity and independently if they have supplied float liquidity to the pool. Hence, a liquidity provider that supplies

float liquidity and is more exposed (cannot remove liquidity unless borrowers repay it, or the maturity has passed) do not get any additional reward.

Version 2 Specification and code changed

Version 2 introduces a novel fee structure that collects fees when `borrow()` function is called and distributes the collected fees to liquidity providers that have supplied float. In case of a positive invariant, swap fees are also distributed to float providers.

Version 3 Specification changed

The respective code has been removed according to the new specifications of Version 3.

6.12 Possible Overflows

Security Medium Version 1 Code Corrected

Primitive Finance pointed out these issues while the audit was ongoing. They are aware that the following expressions could overflow:

```
res.cumulativeRisky += res.reserveRisky * deltaTime;
uint256 reserveRisky = (res.reserveRisky * 1e18) / res.liquidity;
```

Code corrected

The overflow is avoided by casting the variables to `uint256` as follows:

`res.cumulativeRisky += uint256(res.reserveRisky) * deltaTime;`
`delRisky = (delLiquidity * reserve.reserveRisky) / reserve.liquidity;`, where `delLiquidity` is of type `uint256`.

6.13 Possible to Frontrun on Claim Request

Design Medium Version 1 Specification Changed

In some situations, e.g., the price of the underlying token changes significantly, one (or more) liquidity providers might want to exit their positions and call `function claim()` to remove their liquidity from float. However, an attacker might frontrun this transaction and call `function borrow()` and prevent the liquidity provider from exiting their position.

Version 3 Specification changed

The respective code has been removed according to the new specifications of Version 3.

6.14 Redundant and Improper Revert Condition

Design Medium Version 1 Code Corrected

The two functions `balanceRisky()` and `balanceStable()` verify if the call returns the balance correctly by checking: `if(!success && data.length < 32)` and revert if the condition is true. This makes sense if `success` is `false` or the call did not return a value (`data.length < 32`). With using `&&` instead of `or` the condition would not revert for the combination of `success = false` and `data.length > 32` (which is possible). As the function needs the balance to work properly, we do not see a case where this should not revert if `data.length < 32`. The `data.length` check is also

redundant because it is also performed in `abi.decode(data, (uint256))`. Additionally, it might make sense to reevaluate if the `less than` condition makes sense or an equal condition would be more suitable.

Code corrected

The client has updated the check as follows: `if (!success || data.length < 32)`.

6.15 Inconsistency of Input Parameters

Design Low Version 3 Code Corrected

The engine contract is not consistent on the number of decimals an input value should have when called externally. More precisely, the function `create()` expects the strike price to have 18 decimals, independently from the decimals of the stable token. However, the function `swap()` expect the `deltaIn` to have the same decimals as the token being swapped in. A similar format is expected by functions `deposit()` and `withdraw()`.

Code corrected

The function `create()` expects the strike price to have the same number of decimals as the stable token. Also, the specification has been updated accordingly.

6.16 Unused Error Definition

Design Low Version 3 Code Corrected

The error `ZeroLiquidityError` is defined in `IPrimitiveEngineErrors` but not used.

Code corrected

The `ZeroLiquidityError` error is now used in `remove` and `allocate`.

6.17 Redundant Storage Read

Design Low Version 2 Specification Changed

Reading from state storage consumes more gas than reading from memory. The compiler often handles redundant storage reads. To avoid paying multiple times for storage reads, storage variables can be buffered in memory if used more than once. This is tha case for `precisionStable` and `precisionRisky` in `create` and `swap` as they are accessed more than once from storage.

Specification changed

The decimal accounting was changed in Version 3. This issue does not exist anymore.

6.18 Code Duplication

Design Low Version 1 Code Corrected

The following functions share the same code which could be reused:

In `borrow`, `repay`, `remove` and `allocate`:

```
delRisky = (delLiquidity * reserve.reserveRisky) / reserve.liquidity; // amount of risky from removing
delStable = (delLiquidity * reserve.reserveStable) / reserve.liquidity; // amount of stable from removing
```

Code corrected

The duplicated statements are moved into a function `getAmounts()` in the `Reserve` library.

6.19 Duplicated Calculation of Invariant

Design Low Version 1 Code Corrected

The `function swap()` after updating the timestamp of the pool, calculates the invariant for the new time until expiry (tau): `int128 invariant = invariantOf(details.poolId);`. Afterwards, depending on the value of `riskyForStable` parameter, the function calls either `getStableGivenRisky()` or `getRiskyGivenStable()`. Both these functions call `function invariantOf()`, which recalculates the invariant for the same pool and the same timestamp. Although recalculating the invariant is reasonable for external calls, it increases the gas consumption for calls from the swap function.

Code corrected

The updated function `swap()` calls the `getRiskyGivenStable()` and `getStableGivenRisky()` functions from the `ReplicationMath` library which take the invariant as an argument and do not recalculate it.

6.20 Implementation of `getStableGivenRisky` Function

Correctness Low Version 1 Code Corrected

The function specification for calculating the `stablePerLiquidity` do not include the value of the last invariant. However, the function implementation adds the last invariant in the computed stables, therefore resulting in this formula: `stablePerLiquidity = K*CDF(CDF^-1(1 - riskyPerLiquidity) - sigma*sqrt(tau)) + invariantLastX64`.

Code corrected

The code comment has been updated correctly.

6.21 Possible Gas Optimization in the Deposit Function

Design | Low | Version 1 | Code Corrected

The `function deposit()` in the engine contract allows users to deposit a single token to their margin account, therefore the function calls the `balanceof()` function only for the token with a positive delta. However, after the callback function for the transfer executes, the function checks the balance of both tokens (performs two `balanceOf()` calls). The function can optimize the gas consumption again by calling the `balanceOf()` only for the token added.

Code corrected

The updated version of the function `deposit()` now checks if the delta value is greater than 0 before reading the balance (before and after the transfer) for the stable and the risky tokens.

6.22 Return Value in `safeTransfer`

Design | Low | Version 1 | Code Corrected

The function `safeTransfer()` in the library `Transfers` checks if an ERC20 token transfer completed successfully, otherwise the function reverts. Currently, the function returns a boolean value but it is never checked in the caller functions.

Code corrected

The return value in `safeTransfer()` has been removed.

6.23 `MANTISSA_INT` Constant

Design | Low | Version 1 | Code Corrected

`MANTISSA_INT` is a constant defined in units library and its value in 64x64 format corresponds to 10x the value of the constant variable `Mantissa` in units library.

Code corrected

The unused `MANTISSA_INT` has been removed.

7 Notes

We leverage this section to highlight further findings that are not necessarily issues. The mentioned topics serve to clarify or support the report, but do not require an immediate modification inside the project. Instead, they should raise awareness in order to improve the overall understanding.

7.1 Engine Shall Not Have Privileges in Other Smart Contracts

Note Version 1

With the current setup, for security reasons, the engine contract must not have privileges in any other smart contracts. The main reason is that the engine contract supports several callbacks which potentially could have the same function signature as a sensitive function in the contract being called.

7.2 Limited Supported Token Pairs

Note Version 1

PrimitiveFactory contract deploys a unique PrimitiveEngine contract for a pair of ERC20 tokens. The `function deploy()` takes as arguments the addresses of the two ERC20 tokens and assumes that they are implemented correctly and behave as expected. The `function deploy` checks only if the provided addresses of the risky token and the stable one are not the same and that they are different from `address(0)`.

Technically, it is possible to deploy a PrimitiveEngine with any arbitrary pair of tokens, such as: compromised/malicious tokens, two risky tokens, two stable tokens, or with switched addresses for the risky and stable tokens. Therefore, the filtering of the `bad` or malicious engines and their respective pools should happen on the application level (outside the audited smart contracts) in order to protect users from interacting with incorrect engines.

We explicitly mention that the contract ONLY works with standard ERC20 tokens that do not have any unusual behavior like inflation, deflation, locking, fees, two addresses etc. Users needs to carefully evaluate if the pool's token fulfill the requirements!

A check in the factory before deploying the engine might at least prevent accidentally adding a token with unsupported decimals.

7.3 More Testing for CDF

Note Version 1

The correctness of the cumulative normal distribution function and its inverse function are important for the whole protocol. The functions `getCDF()` and `getInverseCDF()` ensure that the pool maintains the correct values of stable and risky tokens at any time. Both functions compute approximate values and the current testing shows that the error falls below a chosen threshold. However, the code calls these functions in the pattern `getCDF(getInverseCDF(x) + volatility)` (refer to `getRiskyGivenStable()` and `getStableGivenRisky()` functions), therefore it is highly recommended to expand the testing for checking how the combine error changes when the functions are called as in the above example.

7.4 Oracle Usage

Note Version 1

In case any project uses the current marginal prices as oracles, the oracle price would be an easy target to price manipulation.

7.5 Stuck Funds

Note Version 1

When the `PrimitiveEngine` contract calls callbacks from other contracts to make a token transfer, the engine only checks that its token balance increased by a value equal or greater than an expected amount. Afterwards, the engine updates the reserve balances with the expected amount. However, if the external contract transfers more token than expected, the difference (tokens transferred - expected tokens) are locked in the engine contract and neither pools, nor liquidity providers can access them. Below is a code example from the `function create()`:

```
if (balanceRisky() < delRisky + balRisky) revert RiskyBalanceError(delRisky + balRisky, balanceRisky());
if (balanceStable() < delStable + balStable) revert StableBalanceError(delStable + balStable, balanceStable());
```

Funds can also be locked during liquidity allocation if either `delRisky` or `delStable` does not match the `delLiquidity` that the user intents to allocate. The function `allocate()` computes the respective delta liquidities for both tokens (risky and stable) and rewards the smallest delta liquidity to the user. The code is shown below:

```
uint256 liquidity0 = (delRisky * reserve.liquidity) / uint256(reserve.reserveRisky);
uint256 liquidity1 = (delStable * reserve.liquidity) / uint256(reserve.reserveStable);
delLiquidity = liquidity0 < liquidity1 ? liquidity0 : liquidity1;
...
liquidity[recipient][poolId] += delLiquidity; // increase position liquidity
```

The same is true for all other funds that are accidentally send to the contract or intentionally forced into the contract.

7.6 Timestamp Conversion Limit

Note Version 1

The `_blockTimestamp` function converts the `block.timestamp` from a `uint256` to a `uint32`. Hence, limiting the maximum value for the timestamp to Sunday, February 7, 2106. This is in 84 years. The contract will have issues in case it is used this long at that point in time.