PUBLIC

Code Assessment

of the Core Protocol V1 Smart Contracts

October 18, 2022

Produced for

MYSO FINANCE

by

CHAINSECURITY

Contents

1 Executive Summary

Dear Myso Team,

Thank you for trusting us to help MYSO Finance with this security audit. Our executive summary provides an overview of subjects covered in our audit of the latest reviewed contracts of Core Protocol V1 according to Scope to support you in forming an opinion on their security risks.

MYSO Finance implements a borrowing system which does not expose borrowers to liquidation risks. Each loan has the same duration and does not rely on any price oracle or curve-based pricing.

The most critical subjects covered in our audit are asset solvency, functional correctness, access control, and precision of arithmetic operations. Security regarding all the aforementioned subjects is high. In the first iteration of the engagement, we uncovered a few medium-severity issues related to the functional correctness that were addressed in the updated codebase.

The general subjects covered are upgradeability, documentation, trustworthiness, gas efficiency and code complexity. The contracts in scope of this review are not upgradable and do not have any privileged account, hence the security regarding upgradeability and trustworthiness is high. The project has extensive documentation and inline code specification. We reported possibilities to improve the gas efficiency which were acknowledged by MYSO Finance but not adopted due to code size restrictions. Regarding code complexity, we highlighted a functionality that implements a complex logic to optimize storage costs and could be simplified, see Optimizations at the cost of added complexity.

It is important to note that security audits are time-boxed and cannot uncover all vulnerabilities. They complement but don't replace other vital measures to secure a project.

The following sections will give an overview of the system, our methodology, the issues uncovered and how they have been addressed. We are happy to receive questions and feedback to improve our service.

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	0
Medium-Severity Findings	4
• Code Corrected	2
• Specification Changed	2
Low-Severity Findings	16
• Code Corrected	9
• Specification Changed	1
• Code Partially Corrected	2
• Risk Accepted	1
• Acknowledged	3

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 following source code files inside the Core Protocol V1 repository based on the documentation files:

- `BasePool.sol`
- `interfaces/IBasePool.sol`
- `interfaces/IPAXG.sol`
- `pools/paxg-usdc/PoolPaxgUsdc.sol`
- `pools/usdc-weth/PoolUsdcWeth.sol`
- `pools/weth-cusdc/PoolWethCusdc.sol`
- `pools/weth-dai/PoolWethDai.sol`
- `pools/weth-usdc/PoolWethUsdc.sol`

The table below indicates the code versions relevant to this report and when they were received.

V	Date	Commit Hash	Note
1	29 Aug 2022	eaa5c68f0238a45bfbaa51351bace4c55543c059	Initial Version
2	26 Sep 2022	b3a80dfd3f989688ada7bed341ca7ad775e96211	Version 2
3	11 Oct 2022	67b472411ced07d282dc7c757bb183e1ee5e74f5	Version 3
4	18 Oct 2022	810069f12e0056062ca2f7ef2ed6cb006badb0a1	Version 4

For the solidity smart contracts, the compiler version `0.8.17` was chosen.

2.1.1 Excluded from scope

Any file not listed above, third party libraries, and any external token that Core Protocol V1 interacts with were outside the scope of this code assessment.

2.2 System Overview

This system overview describes the received version (Version 2) of the contracts as defined in the Assessment Overview.

Furthermore, in the findings section, we have added a version icon to each of the findings to increase the readability of the report.

MYSO Finance offers a borrowing system with a fixed interest rate and no liquidation. Moreover, it does not rely on any price oracle or curve-based pricing.

To do so, MYSO Finance uses pools, `BasePool.sol`, that hold a loan token `L` and collateral token `C`, where borrowers can take a loan against the collateral that they need to repay before a tenor period. If

the loan is repaid during the tenor period, LP can claim the repaid amount in `L` token. If the tenor period expires, the loan cannot be repaid and the LP can claim the collateral.

When a loan is taken for a transferred amount `X` of collateral, the final loan amount is derived from `X"=X'-X'*protocolFee` with `X'=X-transferFee(X)`, where `transferFee` is the fee, if any, taken by the token transfer, e.g. PAXG. Then, the final loan amount is `Y(X")=(X"*mlc*al)/(X"*mlc+al)`, where `mlpc` is the fixed maximum amount of loan per collateral and `al` is the maximum available liquidity in the pool, i.e., `al=totalLiquidity-MIN_LIQUIDITY`.

The fixed interest rate at maturity depends on the liquidity of the pool before (`P`), and after the loan (`P-Y(X")`). It is computed as the arithmetic average of `r(P)` and `r(P-Y(X"))`, where `r(x)` is defined as follows:

$$\frac{r_1 * B_1}{x} \quad if \quad x < B_1$$

$$r_2 + \frac{(r_1 - r_2) * (B_2 - x)}{B_2 - B_1} \quad if \quad B_1 \leq x \leq B_2$$

$$r_2 \quad if \quad B_2 < x$$

where `B_1` and `B_2` are two liquidity bounds (in loan token decimals), and `r_1` and `r_2` are the rate parameters (in 18 decimals). With this formula, the rate is parabolic when the total liquidity is below `B_1`, linear when the liquidity is between the target bounds, and constant with rate `r_2` when there is a lot of liquidity. This ensures the LP a minimal interest rate of `r_2`. In some cases when the interest rate moves from flat to linear, or from linear to parabolic for a loan, it becomes cheaper to borrow two smaller loans.

The intention of MYSO Finance is to deploy several pools per token pair with different `mlpc` or tenor durations, in order to cover the range of each supported token pair price changes. Each `BasicPool` has an internal loan counter `loanIdx`, starting at `1`, so that it can give each loan a unique ID. The counter is incremented by `1` after each new loan.

Every token transfer is done with OpenZeppelin's `safeTransfer` library.

2.2.1 Pool owner

The pool owner (`poolCreator`) receives the protocol's fee. The actual owner can give ownership to another address with function `proposeNewCreator` and the new owner needs to take ownership with the function `claimCreator`.

2.2.2 Actions approvals

The `setApprovals` function can be used by anyone to allow other addresses to trigger action on their behalf. The possible approval actions are `REPAY`, `ROLLOVER`, `ADD_LIQUIDITY`, `REMOVE_LIQUIDITY` and `CLAIM`. The function resets every time the approvals from `msg.sender` for an `approvee`.

2.2.3 Collateral and repayments aggregation

Aggregation of loans into buckets is a feature of the system that makes it possible for LPs to claim their repayments and collateral from a batch of loans in a single call and without the need of iterating through all the loans. The contract supports three sizes of buckets: `baseAggrBucketSize`, `10 x baseAggrBucketSize`, and `100 x baseAggrBucketSize`, where `baseAggrBucketSize` is set in the constructor and should be a multiple of `100`. An LP should always claim from buckets with bigger size to make the claiming cheaper in terms of gas. However, claiming should always be performed in ascending order based on loan Ids, as claiming recent loans or buckets prevents the LP from claiming any previous loan. In order to claim from a bucket, an LP should have had the same amount of shares for all loans in the bucket.

2.2.4 Liquidity Providers

The actions of the liquidity providers are tracked in two arrays. The first one, `sharesOverTime`, tracks the history of the shares updates and the second one, `loanIdxsWhereSharesChanged`, tracks the history of the `loanIdxs` where such update occurred. The second array always has one less element than the first one, because the first liquidity addition for an LP is not counted as an update. That way, LPs know how many shares apply to each loan and how much they can claim. Moreover, a pointer is used to indicate which index of the `sharesOverTime` array must be considered for the next claim. In addition to the pointer, the field `fromLoanIdx` indicates the lower bound index from which LP can claim.

- `addLiquidity`: LP can use this function to initially add liquidity or for top-up. The senders can call this function for themselves or on behalf of someone else if approved (`ADD_LIQUIDITY`). The LP will receive a number of shares based on the pool's state and the amount of liquidity they provide. For a liquidity amount `X`, the number of shares is given by:

$$\begin{cases} \frac{X}{1000} & if \quad totalLpShares = 0 \\ \frac{X * totalLpShares}{totalLiquidity} & otherwise \end{cases}$$

Note that if `totalLpShares==0` and `totalLiquidity!=0`, the remaining dust is swept from the pool and sent to the treasury before adding new liquidity.

- `removeLiquidity`: LP can remove liquidity based on their current shares with this function. The senders can call this function for themselves or on behalf of someone else if approved (`REMOVE_LIQUIDITY`). It is not possible to remove liquidity before the `MIN_LPING_PERIOD` which is set to 120 seconds to avoid flash liquidity provision. Caller need to provide the number of shares `Y` they want to remove, the liquidity they receive back is given by the formula `(Y * (totalLiquidity - MIN_LIQUIDITY)) / totalLpShares`.
- `claim`: LP or approved addresses (`CLAIM`) can claim their share on repaid (loan tokens) or defaulted (collateral tokens) loans with this function. They need to provide an ordered array of each loan index they want to claim where they have non-zero shares. The loan indices in the array must correspond to a period where the number of shares has not been updated, i.e. the indices must be between two values in the `loanIdxsWhereSharesChanged` array. Once the checks for the indices validity has passed, the LP's `fromLoanIdx` and `currSharePtr` are updated to point to the next valid index that is claimable, so no double claiming is possible. The claimed collateral is always transferred to the `msg.sender`. The `claim` function allows to directly reinvest the claimed `L` tokens in the pool, it can be done by the LP or an approved address (`ADD_LIQUIDITY`), otherwise the claimed `L` tokens are transferred to the `msg.sender`. If a claimable loan index is missed, the claim is skipped and cannot be claimed later.
- `claimFromAggregated`: This function allows LPs to claim repayments and collateral from multiple loans (repaid or defaulted), whose tenor period has passed. The contract sets consecutive loans in groups, called buckets, to optimize the gas costs of claiming hundreds or thousands of loans without the need to iterate through them one by one. The function takes as input an array of indices `_aggIdxs` which should be ordered and represent the limits of consecutive buckets to be claimed. Similarly as in `claim`, if the `_isReinvested` flag is set to `true`, tokens are reinvested into the pool. The caller must specify the starting and ending indices of the buckets they want to claim from and must be entitled to claim every `loanIdx` in the bucket. The difference between two indices of the `_aggIdx` must match the size of the target bucket, and cross bucket claims are not possible.
- `overrideSharePointer`: LP can use this function in order to skip claims. However, once a claim is skipped there is no going back.

2.2.5 Borrowers

- `loanTerms`: borrowers can call this function by specifying the amount of collateral (after token transfer fees) and it will return the following values:

- the loan and repayment amounts, in loan token decimals, that would apply if the loan was taken with the current total liquidity,
- the pledge amount, which is the collateral amount minus the protocol fee
- the protocol fee, in collateral token decimals
- the current total liquidity, in loan token decimals

The `loanTerms` function will revert if one of the following conditions is satisfied: the pre- or post-loan liquidity is under the pool's `MIN_LIQUIDITY`; the loan amount is smaller than the pool's minimum loan amount; the repayment amount is smaller than the loan amount; or the repayment amount per LP share goes down to 0 due to rounding error.

- `borrow`: borrowers call this function when they want to take a loan. The final loan amount is computed by the `loanTerms` function and is given by the formula detailed above. After the loan is taken, the total liquidity of the pool is reduced by the loan amount. Upon calling this function, borrowers must provide the following parameters:

  - the address that will take the loan, either themselves or someone else without constraints
  - the amount of collateral the borrower will send
  - the minimum loan limit, in loan token decimals
  - the maximum amount the borrower agrees to repay, in loan token decimals
  - a deadline before which the borrow must occur
  - a referral code, currently not used but can serve later for a possible referral program

  Once the loan terms have been computed, the aggregation buckets are updated accordingly (+ collateral amount per share), the global loan counter is incremented, the collateral is transferred from the `msg.sender`, the protocol fee is transferred to the treasury and the loan amount is sent to the `msg.sender`.

  It is important to note that a loan can be taken on behalf of anyone, the collateral amount is transferred from `msg.sender` and the loan amount is sent to the `msg.sender`. However, only the loan owner can repay it or issue the respective approvals.

- `repay`: this function allows borrowers to repay their loan before the tenor period ends. They must provide the target `loanIdx`, the recipient, which must be either the approved sender (`REPAY`) or the loan owner, and the amount of `L` tokens they are sending back. A loan cannot be repaid in the same block the loan was taken to avoid flashloans behaviors. The sent amount after transfer must be at least equal to the amount that must be repaid, but also cannot be too big. There is a `1%` margin on the maximum repayment amount that can be sent, any extra payment will benefit the LP. There is no partial repayment. Once all the checks have passed, the aggregation buckets are updated (- collateral amount per share, + (repayment amount + extra) per share), the loan tokens are transferred from the `msg.sender` and the collateral is sent to the specified recipient.
- `rollOver`: the `rollOver` function allows borrowers to repay an old loan and take a new one without moving the collateral. The sender must be the loan owner or must be approved (`ROLLOVER`). The parameters are the `loanIdx` that need to be repaid, a minimum loan limit for the new loan, a maximum repay limit for the new loan, a deadline before which the rollover must take place, and the amount of `L` token that will cover the roll over cost. As for `repay`, a roll over cannot happen after the loan expiry or in the same block that it was taken. The amount sent must at least cover the roll over cost, but also must be in a `1%` margin for maximum roll over cost coverage, any extra payment will profit the LP. The roll over cost is determined by the difference between the old loan's repayment amount and the new loan amount. The new loan amount is computed by the `loanTerms` function and is given by the formula detailed above. Once the repayment checks have passed, the aggregation buckets are updated (- old collateral amount per share, + (repayment amount + extra) per share), the new loan info is stored, the aggregation buckets are updated with the new loan (+ new collateral amount). Note that the new collateral amount is equal to the old one

minus the protocol fee. After the new loan is taken, the total liquidity of the pool is reduced by the amount of the new loan, the sent amount of loan token is transferred from the `msg.sender`, and the protocol fee is sent to the treasury. This function is particularly beneficial for loans where the collateral is a token with transfer fees since the collateral is not transferred from the borrower to the pool again.

2.2.6 Trust model

Pool contracts do not have any privileged account with special permission and are not upgradable, however MYSO Finance is trusted to deploy the pools with the correct parameters. Anyone can deploy pools with arbitrary tokens, hence LPs and borrowers should validate the pools they interact with. The addresses that have been approved for actions are trusted to behave in a non-adversarial way for the approver.

Tokens: Any external token used in the system is considered fully trusted and pool deployer should carefully assess supported tokens. Only ERC20-compliant tokens without special behavior (e.g., transfer callbacks like ERC777, or inflationary/deflationary tokens) and implementing function `decimals` are supported by the system. This review is limited to pools that integrate with the following tokens:

- `PAXG`: 0x45804880De22913dAFE09f4980848ECE6EcbAf78
- `DAI`: 0x6B175474E89094C44Da98b954EedeAC495271d0F
- `WETH`: 0xC02aaA39b223FE8D0A0e5C4F27eAD9083C756Cc2
- `USDC`: 0xA0b86991c6218b36c1d19D4a2e9Eb0cE3606eB48

Finally, for upgradable tokens such as `USDC` we assume new implementations are always trusted and according to the existing specifications.

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 the 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 defined in the engagement letter. We assessed whether the project follows the provided specifications. These assessments are based on the provided threat model and trust assumptions. 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 that over the life cycle of any software, changes to the product itself or to the environment in which it is operated can have an impact leading to operational behaviors other than those 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 severity. 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. The findings are split into these different categories:

- 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	6

5.1 Missing Event for `poolCreator` Update

Design Low Version 3 Acknowledged

The functions that allow an update of the pool creator perform important state change without emitting an event.

Acknowledged:

MYSO Finance has acknowledged this issue, but has decided to keep the functions as-is due to limitations on the code size.

5.2 Gas Optimizations

Design Low Version 2 Code Partially Corrected Acknowledged

1. State variables `r1`, `r2`, `liquidityBnd1`, `liquidityBnd2`, and `minLoan` are set in the constructor and are read-only afterwards, thus they can be declared as `immutable` to save gas.
2. In function `removeLiquidity`, the `SLOAD` to access `totalLiquidity` when emitting the event could be avoided if memory variables are used.
3. In function `borrow`, the storage field `totalLpShares` is passed to `updateAggregations`. Even if it is a hot address, accessing it again costs 100 gas, a memory variable would be more

efficient as `MLOAD` costs 3 gas. It is also the case for `loanIdx` in the `borrow` function and in the `rollOver` function.

4. `rollOver` function computes `_sendAmount - getLoanCcyTransferFee(_sendAmount)` multiple times. Storing the result in a memory variable will save gas.
5. In function `updateAggregations`, `repaymentUpdate` is always computed but is only needed when `_isRepay` is `true`.
6. In Version 2, the constant variable `treasury` was changed into a state variable `poolCreator` which could be declared as `immutable`.
7. In Version 3, function `borrow` performs an unnecessary `SLOAD` to get the loan index when emitting the event `Borrow`.

Code partially corrected:

3. The storage variables `totalLpShares` and `loandIdx` are stored in memory variables.
4. The logic has been moved in function `checkAndGetSendAmountAfterFees` and the result of the subtraction is cached.

Acknowledged

MYSO Finance replied:

```
We acknowledge that certain variables could be made immutable and also within
functions a few cases where storing a repeatedly used variable as a memory variable
would also save gas, but we were running against byte code limits and stack too deep
errors, and instead of significantly refactoring, we decided against implementing many
of the optimizations.
```

5.3 Force Other LPs to Sell Cheap Loans

Design | Low | Version 1 | Code Partially Corrected | Risk Accepted

Liquidity providers have the guarantee that they receive a minimum interest (flat rate `r2`) from the repaid loans. If there is enough demand for borrowing from a pool, the interest rate goes up which makes it more attractive for LPs to provide liquidity into it. However, one can implicitly force LPs to lend tokens at a lower interest rate. To achieve that, an attacker needs to add liquidity into a pool and then borrow.

For example, if the available liquidity in a pool is between `liquidityBnd1` and `liquidityBnd2`, the attacker adds enough liquidity, so the interest rate gets lowered. Taking a loan immediately after this operation, the attacker consumes part of its liquidity and part of other LPs liquidity with a lower interest rate than the market rate. The attacker borrows enough tokens such that the interest rate is back to the one before the attack started. This way, the liquidity added by the attacker is not exposed to lower interest rates, while other LPs effectively were forced to sell loans with low interest rates.

Code partially corrected:

MYSO Finance implemented two mitigation measures to reduce the likelihood of such attacks:

1. Smart contracts (or EOA) cannot add liquidity and borrow from the pool in the same transaction (or block), as functions `addLiquidity` and `borrow` track `tx.origin`. This complicates but does not eliminate the risk of the attack described above. Instead of using

one single contract to atomically provide liquidity and borrow, an attacker would need to take the risk of carrying the attack non-atomically, or use flashbots, which require more work.

2. Increase the minimum LP-ing period from 30sec to 120sec to increase the exposure of the attacker's liquidity to the same attack vector.

Risk accepted:

MYSO Finance is aware that the attack is inherent to the system's architecture and states that the two mitigation measures described above will reduce the likelihood of such attacks but not fully prevent them. Furthermore, the attack does not lower the interest rates below the flat rate of a pool (`r2`), hence LPs still earn a minimum yield.

5.4 Optimizations at the Cost of Added Complexity

Design | Low | Version 1 | Risk Accepted

The function `updateLpArrays` considers 7 different cases when an LP updates its position and optimizes the storage usage by avoiding storing redundant data. This optimization of the storage comes with added complexity in the logic of the function `updateLpArrays` although the majority of cases (4 out of 7) are expected to happen rarely.

Risk accepted

The client accepts the risk associated with the code complexity to optimize storage gas costs and will consider refactoring the function in a future version of the codebase.

5.5 Rollover Not Allowed in Certain Situations

Design | Low | Version 1 | Acknowledged

Function `rollOver` in `BasePool` reverts if a borrower renews its loan and the new loan amount is higher than the repayment of the previous loan. This might be the case if the pool has more available liquidity when rollover happens than when the loan was initially taken. The restriction is enforced in the following check:

```
if (loanAmount >= loanInfo.repayment) revert InvalidRollOver();
```

Acknowledged

MYSO Finance has decided to keep the code unchanged as this scenario is expected to happen rarely, and users still have an alternative to perform the same operation, as explained in their response:

```
For bytecode reasons we refrained from supporting this use case as it would require an
additional if-else to distinguish between calling transferFrom (regular case where
borrower pays to rollOver) and transfer (rare case where borrower receives a refund).
The situation where a rollOver would lead to a refund is expected to occur - if at
all - rather rarely, hence not supporting it isn’t deemed a significant loss in
functionality. Moreover, if necessary a borrower could also independently emulate a
rollOver for this situation by atomically repaying and borrowing using a flashloan.
```

5.6 Unclaimed Tokens Remain Locked

Design | Low | Version 1 | Acknowledged

Liquidity providers specify the loan indices for their claims and are allowed to skip loans that are not sufficiently profitable. Once an LP skips a loan, it cannot claim it anymore. Hence, a pool continuously holds loan and collateral tokens amounts that cannot be claimed by LPs and are locked. The only way to recover loan token funds is if all LPs remove their liquidity from a pool (`totalLpShares == 0`) and then one adds liquidity which triggers the transfer of dust to the treasury. However, there is no way to recover collateral amounts left in the pool from skipped claims.

Acknowledged

MYSO Finance acknowledges the issue and does not plan on adding a functionality to track the unclaimed loans as it would increase significantly the gas costs. However, MYSO Finance will simplify the UI for claiming and promote aggregate claims to reduce the number of unclaimed loans.

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	0
Medium-Severity Findings	4

- Mismatch of Implementation With Specification Specification Changed
- Missing Loan Owner Sanity Check When Borrowing Code Corrected
- Protocol Fee Computation Can Overflow Code Corrected
- Total LP Shares Are Capped in Pools Specification Changed

Low-Severity Findings	10

- Emission of ApprovalUpdate Event Can Be Tricked Code Corrected
- Deletion of Timestamps From Mapping Code Corrected
- Redundant Events Emitted Code Corrected
- Disabled Optimizer Code Corrected
- Inaccessible TREASURY Account Code Corrected
- Insufficient Check for Minimal Loan Given Total LP Shares Specification Changed
- Inverted NewSubPool Event Token Fields Code Corrected
- Misleading ApprovalUpdate Event Code Corrected
- Missing Precision of Pool Parameters Code Corrected
- Non-indexed Events Code Corrected

6.1 Mismatch of Implementation With Specification

Correctness Medium Version 1 Specification Changed

The specifications of the `borrow` function state:

```
In this case the collateral is deducted from the 3rd party ``msg.sender`` address but
the ``_onBehalfOf``  address receives the loan and is registered as the loan owner
(including the ability to repay and reclaim the pledged collateral).
```

However, the function takes the collateral from `msg.sender` and also sends the loan amount to `msg.sender` in violation with the specifications:

```
IERC20Metadata(collCcyToken).safeTransferFrom(msg.sender, address(this), _sendAmount);
...
IERC20Metadata(loanCcyToken).safeTransfer(msg.sender, loanAmount);
```

Specification changed

The specification in section 'Calling Functions on Behalf' of the gitbook has been revised to reflect the code behavior:

```
In this case the collateral is deducted from msg.sender and msg.sender also receives
the loan but the_onBehalfOf address is registered as the loan owner (including the
ability to repay and reclaim the pledged collateral). This allows wrapping and
unwrapping of tokens through a peripheral contract.
```

6.2 Missing Loan Owner Sanity Check When Borrowing

Design | Medium | Version 1 | Code Corrected

A borrower can take a loan on behalf of anyone without restriction and this can result in the loan never being repaid. If a borrower calls `borrow` with an `_onBehalf` address they do not control or is aware that it will be the owner of a loan, the loan will default since the borrower is not the loan owner and is probably not allowed to repay it. E.g., `borrow` is called with `_onBehalf=address(0)`, then the loan will default for sure.

Code corrected

The function `borrow` has been updated to perform a sanity check that address `_onBehalf` is not set to `addr(0)` by mistake. However, the caller is still responsible for providing a correct address for `_onBehalf` which repays the loan if required.

6.3 Protocol Fee Computation Can Overflow

Design | Medium | Version 1 | Code Corrected

The protocol fee computation in `loanTerms` can overflow if the `protocolFee` is non-zero. The multiplication in `_protocolFee = uint128((_inAmountAfterFees * protocolFee) / BASE)` is carried in `uint128` and might overflow. Example is with `protocolFee = 5 * 10**5` which is also the maximum allowed fee and `_inAmountAfterFees=uint128(uint256(2**128) / uint256(5 *10**15))+1=68056473384187692692675` which may seem to be a lot but could be a realistic amount for collateral tokens with 18 decimals and low value.

Code corrected

In the second version of the codebase, the variable `protocolFee` was renamed `creatorFee` and its type was changed to `uint256` to avoid possible overflows in the computation highlighted in the issue above.

6.4 Total LP Shares Are Capped in Pools

Design | Medium | Version 1 | Specification Changed

The function `_addLiquidity` performs two checks to guarantee that an LP will get non-zero token amounts from a small loan, both on repay and default. The checks are implemented as follows:

```
if (
    ((minLoan * BASE) / totalLpShares) * newLpShares == 0 ||
    (((10**COLL_TOKEN_DECIMALS * minLoan) / maxLoanPerColl) * BASE) /
        totalLpShares == 0
) revert PotentiallyZeroRoundedFutureClaims();
```

The first condition evaluates to `true` whenever `totalLpShares > minLoan * BASE`. Since both `minLoan` and `Base` are fixed for a pool, the `totalLpShares` is capped for a pool.

Similarly, the second condition evaluates to true whenever `totalLpShares > ((10**COLL_TOKEN_DECIMALS * minLoan) / maxLoanPerColl) * BASE)` sets another restriction on the maximum `totalLpShares`.

Capping the `totalLpShares` prevents adding liquidity to pools that are attractive to users and have high activity.

Specification changed

The specifications have changed and the checks described above have been removed, hence the unintended capping on total LP shares is not present anymore.

6.5 Emission of `ApprovalUpdate` Event Can Be Tricked

Design | Low | Version 3 | Code Corrected

There is no restriction on the parameter `_packedApprovals` of function `setApprovals`. One could set the 6th bit to `1` even if no approval is updated and the event will be emitted. Moreover, if bits higher than the 6th are set, they will be shown in the emitted event.

Code corrected:

The input parameter `_packedApprovals` has been sanitized to consider only the 5 least significant bits.

6.6 Deletion of Timestamps From Mapping

Design | Low | Version 2 | Code Corrected

The timestamp stored in `lastAddOfTxOrigin` are never deleted from the mapping although they are used only to disallow LPs from adding liquidity and borrowing in the same block. The entries of this mapping can be deleted, e.g., when LP remove their liquidity, to get gas refunds.

Code corrected:

The entry for an address in the mapping `lastAddOfTxOrigin` is deleted when liquidity is removed.

6.7 Redundant Events Emitted

Design Low Version 2 Code Corrected

The function `setApproval` iterates through all approval types and emits an event independently if an approval status is updated or not. Therefore, even if only one approval type is changed for an `_approvee`, five events will be emitted.

Code corrected

Function `setApproval` has been updated to emit the event when at least one of the approvals changes state.

6.8 Disabled Optimizer

Design Low Version 1 Code Corrected

In `hardhat.config.js` the optimizer is not explicitly enabled and the default value for hardhat is `enabled: false`. Enabling the optimizer may help to reduce gas cost.

Code corrected

The optimizer has been enabled and the `runs` are set to `1000`.

6.9 Inaccessible TREASURY Account

Design Low Version 1 Code Corrected

The TREASURY address is declared as constant and set to `0x1234567890000000000000000000000000000001` which is not in the control of the developers, hence all protocol fees collected by the system will be locked forever. MYSO Finance is aware of this issue and will use a multisig account for the treasury on deployment.

Code corrected

The constant variable TREASURY is replaced with the state variable `poolCreator` which is assigned to `msg.sender` in constructor.

6.10 Insufficient Check for Minimal Loan Given Total LP Shares

Design Low Version 1 Specification Changed

The second condition in the following code is supposed to check that repayment amount for a loan is big enough that all LPs can claim non-zero amounts if the loan is repaid given their share:

```
if (
    ... ||
    ((repaymentAmount * BASE) / totalLpShares) == 0
) revert ErroneousLoanTerms();
```

The check might not work as intended for loan tokens with low decimals, e.g., USDC (6 decimals), as BASE is a constant with value `10**18`. For example, if `repaymentAmount` is `10**7` (10 USDC) and `totalLpShares` is `10**8` (2 LPs with `5 * 10**7` shares each) the check would still pass.

Specification changed

MYSO Finance has changed the specifications and decided to remove the check above as it effectively would increase the minimum loan amount over time as total LP shares increase.

6.11 Inverted `NewSubPool` Event Token Fields

Correctness | Low | Version 1 | Code Corrected

The `NewSubPool` event definition in `IBasePool.sol` specifies that the first two fields are `collCcyToken` and `loanCcyToken`, but when the event is emitted in the constructor, the two fields are set to `_loanCcyToken` and `_collCcyToken`.

Code corrected

The definition of event `NewSubPool` in `IBasePool` is updated and the parameters are in line with the code that emits the event:

```
event NewSubPool(
    address loanCcyToken,
    address collCcyToken,
    ...
);
```

6.12 Misleading `ApprovalUpdate` Event

Design | Low | Version 1 | Code Corrected

The function `setApprovals` emits an event only when an approval type is set to `true`, even if it was previously the case, and nothing is emitted when an approval is unset. An example is: current approvals are `10101` and the updated approvals are `10100`. The event is misleading in the sense that it will be emitted for indices `0` and `2`, which have not been updated, and no `ApprovalUpdate` event is emitted for the actual update of the index `4`.

Code corrected

The event `Approval` is now emitted for every index with the status `true` or `false` and independently if it was changed from the previous state.

6.13 Missing Precision of Pool Parameters

Design | Low | Version 1 | Code Corrected

The documentations and inline specifications do not describe the precision of the pool parameters. To improve the readability of the code and avoid possible mistakes, the decimals used for all pool parameters such as `r1`, `r2`, `liquidityBnd1` and `liquidityBnd2` should be stated clearly.

Code corrected

Inline code comments were added for the variables mentioned above, which specify the precision of expected values:

```
uint256 r1; // denominated in BASE and w.r.t. tenor (i.e., not annualized)
uint256 r2; // denominated in BASE and w.r.t. tenor (i.e., not annualized)
uint256 liquidityBnd1; // denominated in loanCcy decimals
uint256 liquidityBnd2; // denominated in loanCcy decimals
```

6.14 Non-indexed Events

Design | Low | Version 1 | Code Corrected

No parameters are indexed in the events of contracts `BasePool`. It is recommended to index the relevant event parameters to allow integrators and dApps to quickly search for these and simplify UIs.

Code corrected

MYSO Finance has evaluated the events used in `BasePool` and has indexed parameters that they deem useful for future UI and dashboard integrations.

Several events such as `NewSubPool` and `Approval` have non-indexed parameters, however, the client intentionally kept them unchanged.

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 LP Shares Dilute Over Time

Note Version 2

The shares of an LP dilute over time as more activity happens in a pool by users that borrow and LPs that add more liquidity. Therefore, LPs should monitor their proportion of LP shares to the total LP shares and remove their liquidity from a pool when their share to loan repayments or collateral becomes insignificant.

7.2 LPs Get Slightly Less Token for Their Shares

Note Version 1

The pool keeps a minimum of loan tokens and it does not allow LPs to fully empty a pool. When removing liquidity, LPs get slightly less tokens than their fair share to maintain the minimum liquidity in the pool. The relevant code is:

```
uint256 liquidityRemoved = (numShares *
    (_totalLiquidity - MIN_LIQUIDITY)) / _totalLpShares;
```

7.3 LPs Should Be Careful When Claiming

Note Version 1

LPs can claim their share of repayments and collateral tokens via function `claim` or `claimFromAggregated`. It is important to note that LPs are responsible for claiming loans always in order. Otherwise, any loan skipped during a claim is impossible to be claimed in the future.

Furthermore, LPs can skip all loans during a time window via the function `overrideSharePointer`. Similarly, if an LP calls this function, they cannot claim anymore the repayments and collateral for all loans linked with the skipped shares.

7.4 Limitations on Claiming Batch of Loans

Note Version 1

Both functions `claim` and `claimFromAggregated` allow LPs to claim loans in batches over a period during which the LP has not changed its shares in a pool. LPs should be aware that modifying their position in a pool by topping up or removing liquidity, will require them to perform multiple transactions for the claiming which increases gas costs and potentially prevents LPs from using aggregate claims.

7.5 Locked Tokens

Note Version 1

ERC20 tokens could be accidentally/intentionally sent to the pool contracts. In that case the tokens will be locked, with no way to recover them. Incidents (https://coincentral.com/erc223-proposed-erc20-upgrade/) in the past showed this is a real issue as there always will be users sending tokens to the token contract.

7.6 Minimum Loan Amount Allowed

Note Version 2

The constructor of `BasePool` does not enforce any restriction on the minimum allowed amount for loans. Therefore, the pool deployer should carefully set this value depending on the specific token used as loan token.

7.7 Positions in a Pool Are Non Transferrable

Note Version 1

All positions in pools held by liquidity providers or borrowers are tracked in the contract `BasePool` and they are non-transferable. Users can approve other addresses to act on their behalf, but there is no support for transferring ownership of positions.

7.8 Possible to Overpay Loans

Note Version 1

Functions `repay` and `rollOver` check that the user always pays at least the due amount. However, both functions allow users to overpay their loans by `1%` in case users cannot precisely calculate the sending amount for tokens with transfer fees.

7.9 Profits of a Pool Are Not Equally Distributed

Note Version 1

The profits of a pool from loan repayments are not equally distributed among liquidity providers. The system is designed such that profits for an LP depend on loans that borrow most of their liquidity. For example, if a pool starts with an interest `i` and over time the interest rate goes to `3 x i`, initial LPs will earn payments from loans with interest `i`, while LPs joining later will have higher profits (as the interest rate tripled to `3 x i`).

7.10 Transfer Fee for Upgradable Tokens

Note Version 1

The function `getLoanCcyTransferFee` in contracts `PoolPaxgUsdc` and `PoolWethUsdc` is hard-coded to return `0` as fee for the loan token, namely USDC. We would like to highlight that the pools would not work as expected if upgradable tokens were to introduce fees in new implementations.

7.11 `if` Blocks Without Curly Braces

Note **Version 1**

It is generally good practice to enclose every `if/else` block into curly braces. It increases code readability and lowers possibilities for bugs like the famous `goto fail;` bug in Apple SSL code https://blog.codecentric.de/en/2014/02/curly-braces/.

8 Monitoring

A thorough code audit is just one important part of a comprehensive smart contract security framework.

Next to proper documentation/specification, extensive testing and auditing pre-deployment, security monitoring of live contracts can add an additional layer of security. Contracts can be monitored for suspicious behaviors or system states and trigger alerts to warn about potential ongoing or upcoming exploits.

Consider setting up monitoring of contracts post-deployment. Some examples (non-exhaustive) of common risks worth monitoring are:

1. Assumptions made during protocol design and development.
2. Protocol-specific invariants not addressed/mitigated at the code level.
3. The state of critical variables
4. Known risks that have been identified but are considered acceptable.
5. External contracts, including assets your system supports or relies on, that may change without your knowledge.
6. Downstream and upstream risks - third-party contracts you have direct exposure to (e.g. a third party liquidity pool that gets exploited).
7. Privileged functionality that may be able to change a protocol in a significant way (e.g. upgrade the protocol). This also applies to on-chain governance.
8. Protocols relying on oracles may be exposed to risks associated with oracle manipulation or staleness.

8.1 Project-specific monitoring opportunities

We have identified some areas in Core Protocol V1 that would be well suited for security monitoring.

We classify these into two categories: *invariants* and suspicious *changes*. If an invariant of the system doesn't hold anymore, there has been unexpected behavior requiring immediate investigation. If a change of a suspicious condition has been observed, something has happened which could change the behavior of the system and requires timely investigation to ensure the continued safety.

The following monitoring opportunities have been identified:

Identified suspicious change: The functions `getCollCcyTransferFee` and `getLoanCcyTransferFee` are hardcoded to return a transfer fee of 0 for tokens that currently do not have such fees. However, for upgradable tokens such as USDC, this could change in new implementations, hence this change can be monitored and trigger an alert if fees ever change.

Identified suspicious change: All pools have a finite number of cycles for borrowing and adding liquidity until a potential overflow on the total LP shares may happen. Therefore, the value of `totalLpShares` can be monitored and trigger an alert if it becomes large enough to overflow, e.g., larger than `2**240`, so a new pool can be deployed.