PUBLIC

Code Assessment

of the MethLab DLV Smart Contracts

July 5, 2024

Produced for

METHLAB

by

CHAINSECURITY

Contents

1 Executive Summary

Dear MethLab team,

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

MethLab implements the second version fo their lending protocol. The most important change from the previous version is the introduction of strategies which can automatically determine specific parameters, e.g., the strike price of a loan to-be-issued.

The most critical subjects covered in our audit are security of the funds in the vaults and the loans, potential price manipulations, and the functional correctness of the protocol. A medium severity issue was uncovered, where the owner of a vault can modify loan parameters resulting in loan issuance of unexpected specification for a borrower. Most of the issues have been addressed some low severity issues remain open.

The general subjects covered are access control, gas-efficiency and, testing. The security of all aforementioned subjects is high.

In summary, we find that the codebase provides a high level of security.

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	2
• Code Corrected	2
Low-Severity Findings	6
• Code Corrected	4
• No Response	2

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 DLV 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	12 June 2024	bf96c9cd5f447bc879c277bbfe9fdc08ccaffbcc	Initial Version
2	21 June 2024	d1f2b188612e5e5b41b9b3c8ad02d1cfa48f1dcf	Intermediate fixes
3	2 July 2024	ac86f3153178b58a5f66c64850c44f4703ac5ec7	Second round of fixes
4	3 July 2024	29cca0cd5891859385304414c9c88582d62fe53a	Final version

For the solidity smart contracts, the compiler version `0.8.19` was chosen.

The files in-scope are the following `packages/contracts/src/`:

- `V2/`:
    - `DelegatedLenderVault.sol`
    - `LoanV2.sol`
    - `PythWrapper.sol`
    - `RegistryV2.sol`
    - `Strategy.sol`
    - `PythUtilsExtra.sol`
    - `periphery/LoanCreator.sol`
    - `periphery/LoanRepayer.sol`
- `Loan.sol`
- `periphery/UniswapV3SwapperMultiRoute.sol`

SHA256 hashes are provided for convenience.

```
15ec35792e647e0499645f82c0f955f116abdc1f51717a5210c2df7f50b11759 V2/DelegatedLenderVault.sol
0d11cabdcae44e4b19c6ed12507ea8287bf2048583057c4a131de1618c770b76 V2/LoanV2.sol
9f96d7690b7c36527b6b436e2cbc5f2d273dc4689c9f0b82d192c8248288c7c0 V2/PythWrapper.sol
6a070963f3cf1dc7e2ab89f70e734dcfea96c3bfc4ebfb940224d57832eb98ea V2/RegistryV2.sol
11e9ed875c584077624511fa92a2324c6e618acc3347a0ea39e27e6ad6c15a33 V2/Strategy.sol
5b17987c5f9cb10295a00ab7c9a1d029da77e68ca8b815beec5599196c2ce905 V2/PythUtilsExtra.sol
e9de31467d438e9d90645ea27d760f5cbd2309930d16b13dfa687339f7dc62c1 V2/periphery/LoanCreator.sol
eaa4d1329674d3e951243493259da44241b0ae1f15c3ef346112e1d9e69c51e8 V2/periphery/LoanRepayer.sol
dba721d9b3e8c683983a83f06f119b2d4885cb50e9386764ad12dc250d5e392b Loan.sol
8fd6c59c446464df13a6046210f11e5f0d2f49b150e1bdcb575d3d5b2eb2ba68 periphery/UniswapV3SwapperMultiRoute.sol
```

2.1.1 *Excluded from scope*

Any file not included in Scope is out-of-scope. In particular, third-party libraries, including patched versions, and deployment scripts are not in scope. The evaluation of the economic model of the implemented protocol is beyond the scope of this review. The protocol doesn't implement any restrictions on the configurations of the various modules such as the Lender's vault. Therefore, lenders are responsible for configuring their vaults and borrowers are responsible for correctly choosing the lender vaults from which they'll take loans. The owner of the protocol can set the protocol fees. Attacks concerning a malicious owner were considered out of scope. The smart contracts are aimed to be deployed on the Mantle L2 rollup. We assume that the semantics of the EVM opcodes are the same as on Ethereum. Moreover, the Mantle sequencer is considered to be fully trusted and to function as expected. Moreover, the protocol interacts with a UniswapV3-like contract. We assume that the implementation of the contract and the contracts it interacts with are identical to UniswapV3. As this second version of the protocol builds on top of the first one, all the relevant issues and concerns mentioned in the previous report are also valid for this one.

2.2 System Overview

This system overview describes the initially received version (**Version 1**) 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.

MethLab offers the second version (V2) of a lending protocol for loans with fixed interest, fixed duration, and fixed repayment prices i.e., borrowers can repay for the price that originally was determined during the opening of a position. Funds are not pooled. Instead, each lender owns their pool from which users can borrow money at some pre-defined terms.

Each Vault (lending pool) belongs to one lender and can issue loans of some pre-specified configurations (intents). In V2, the vaults (`DelegatedLenderVault`) define a single intent, therefore, they allow for loan issuance in one borrowed token for one specific collateral token. The lender can define the rest of the parameters of the loan using a strategy. In the current scope, the implemented strategy defines a fixed loan duration, a fixed interest rate, and a fixed strike price that is calculated from the current spot price of the collateral token and some static factor. During the loan duration, the principal and interest can be repaid in any amount at any time returning the corresponding share of collateral as per the strike price.

After a loan has ended, repayments are no longer possible and any remaining collateral can be seized by the lender. The protocol does not contain a liquidation mechanism. Borrowers can choose to sell their borrowed tokens to go short for the full duration of the loan without fears of being liquidated while lenders are always long on the borrowed token.

2.2.1 *RegistryV2*

`RegistryV2` is a simple registry contract that allows to deploy clones of `DelegatedLenderVault` and `Loan`. A vault can be deployed by anyone, loans are deployed from registered vaults. Additionally, the protocol governance can set maker (lender) and taker (borrower) fees that are charged during loan creation.

2.2.2 DelegatedLenderVault

Each lender can call `RegistryV2.deployVault()` to create a Vault for a specific collateral / borrow token pair using a user-supplied strategy contract. The lender can then set an expiry date for the vault using `setCollectionExpiry()` after which no new loans can be created. With `setMinMaxLoanAmt()`, an interval for the amounts a user can borrow can be set. Additionally, the lender can enable /disable the vault with `setCollectionStatus()` or with the `pause()` / `unpause()` functionality.

A lender can `deposit()` any amount of to-borrowed tokens which become instantly available to be loaned out. They can also `withdraw()` the borrow token, as well as any other token, from the contract.

Borrowers can call `createLoan()` to deploy a new `Loan` contract. Collateral is supplied on loan creation and stored in the `Loan` contract until the loan is repaid or the collateral seized.

2.2.3 Loan

A `Loan` is deployed for each individual loan between a lender and a borrower. The borrower can `repay()` any amount (up until the total owed) and receive the respective amount of collateral back at any time. Once the end of the loan duration has been reached, anyone can call `seizeCollateral()` to transfer the remaining collateral back to the lender's vault.

Borrowers can also transfer the debt to another address in case they are not capable to repay anymore or want to sell their debt.

2.2.4 Strategy

In general a strategy is a contract implementing an arbitrary logic returns an intent. Intents are a tuple consisting of information about an offered loan:

- The strike price.
- The interest rate.
- The duration.

The default implementation of a strategy provided specifies the function `setStrategy()` which allows the owner of the contract to set these three values. The strike price, however, is not set directly. The function rather allows the owner to set a certain `adjustmentFactor` (between 0.1% and 95%) that is applied to the spot price at the time of the creation of any given loan. The factor increases the required collateral in comparison to the lent amounts and therefore sets a maximum loss for the borrower and a maximum gain for the lender:

1. If the borrowed token loses market value, the borrower is always incentivized to repay the loan, making the lender exposed to a normal long position in the borrow token.
2. If the borrow token gains market value, the borrower is only incentivized to repay the loan up until the point the borrow position value is equal to the collateral value. If the borrower goes short (by selling the borrow token for the collateral token on the open market), their losses are capped at this point. If they stayed longer, the borrowed tokens would be more profitable than the collateral position. Due to this, the lender's profits are capped as they won't receive the borrowed token back, not profiting from its further gains.

2.2.5 PythWrapper

`Strategy.getIntentData()` creates the strike price for a loan by multiplying the current spot price for a given pair with the `adjustmentFactor`. To reliably retrieve the current spot price, the `PythWrapper` contract is used as an oracle.

It utilizes the pull-based *Pyth* oracle service that allows any user to publish very recent pricing information on-chain themselves. Before calling a function that fetches price data from Pyth, users must call the function `forcePythPriceUpdate()` (or `updatePriceFeeds()` directly on the Pyth contract) for both assets. The `getPrice()` function then uses the prices of the two updated feeds to generate a price and validate that the price is not stale and that the confidence of the providers does not reach a given threshold.

2.2.6 LoanCreator

Borrowers who want to go short instantly can use the `LoanCreator` peripheral contract. Calling `createLoan()` allows the user to define a swap on a third-party exchange that is performed with the received borrow tokens. When loans are created, the borrow tokens are first sent to the `msg.sender` after which a special function `sourceCollateral()` is called on the `msg.sender`. `LoanCreator` performs a swap in this callback to partially pay for the required collateral.

Users using this contract only pay the premium and extra collateral (according to the `adjustmentFactor`) to get access to a loan that gives them the option to later convert the tokens at the strike price.

2.2.7 LoanRepayer

If borrowers went short on a loan and want to repay without holding the extra borrow tokens in advance, they can use the `LoanRepayer``peripheral to simply repay with the collateral tokens by converting them on a third-party exchange. The process follows the same scheme as the ``LoanCreator`.

2.2.8 Changes in version 2

The following changes have been made in Version 2 of the code:

1. Expiry, min/max loan amounts and the collection status have been removed from `DelegatedLenderVault`.
2. Strategies are now whitelisted in `RegistryV2`. Vaults can no longer be deployed with arbitrary strategies.

2.2.9 Changes in version 3

The following changes have been made in Version 3 of the code:

1. Strategies are now controlled by the admin and a special strategist role. Both roles are supposed to be timelocked.
2. The implementation of loans deployed by a specific vault can now be upgraded by the owner of the vault via `upgradeLoanImplementation()`.
3. The implementation of vaults deployed by the registry as well as the default implementation of loans for the new vaults can now be changed by the admin of the registry.
4. In `RegistryV2`, the taker and maker fees and the strategy status are set by the manager role. Moreover, the manager can withdraw the fees.
5. A strategy can be (un)paused by the stragist role or the (un)pause guardian.

2.2.10 Roles & Trust Model

Governance can transfer any tokens sent to the `RegistryV2` contract. In addition, governance can change maker and taker fees at any time.

`PythWrapper` allows the owner to change staleness and confidence thresholds at any time.

`Strategy` allows the owner to update loan and operational parameters at any time.

Vaults can be deployed with arbitrary Strategy contracts (and therefore also arbitrary Oracle contracts).

Contracts are generally immutable except `PythWrapper`.

In Version 3 the following roles were introduced:

- The `RegistryV2` admin: can modify the access control of the contract, and upgrade the default implementations for the vaults and the loans, and whitelist loan implementations and loan upgrades. It is controlled by a long timelock (24-28h).
- The `RegistryV2` manager: can modify the fees, set the strategy status, and withdraw fees. It is controlled by a short timelock.

The two roles are set initially to the same address.

- The Strategy strategist: It is controlled by a 3rd party. It can modify the loan duration, interest rate adjustment factor, and other operational parameters as well as pause and unpause a strategy. It is controlled by a long timelock (24-28h).
- The Strategy admin: can modify the access control of the contract, set the oracle used by the strategy, or perform any of the operations of a strategist. It is controlled by a medium timelock (12 -24h)
- The Strategy pause/(un)pause guardians: can (un)pause a strategy. It is controlled by a short timelock.

All the roles are fully trusted.

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:

- 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	2

- Few Decimals for the Denomination Asset
- Sending Arbitrary ETH in forcePythPriceUpdate

5.1 Few Decimals for the Denomination Asset

Correctness Low Version 1

CS-MLV2-001

`PythWrapper.getPriceInDenom()` calculates the relative price between collateral and borrowed asset, i.e., it calculates the collateral price denominated in the borrowed asset. The price uses the decimals of the borrowed asset. Let's assume that the denomination asset (for example GUSD) has only two decimals. Moreover, let's assume that the collateral asset is worth very little compared to the borrowed asset. Due to rounding errors, the relative price will be reported as 0.

5.2 Sending Arbitrary ETH in `forcePythPriceUpdate`

Design Low Version 1

CS-MLV2-002

A user can update the Pyth price feed by calling `PythWrapper.forcePythPriceUpdate()`. The call forwards the full amount of ETH to the Pyth oracle. However, Pyth exposes an `updateFee()` view function that informs the caller on the required native amount needed. It is best practice to calculate the fee in the same transaction as the fees could change from transaction creation until its execution.

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	2

- Wrong Implementation Code Corrected
- Loan Creation Frontrunning Code Corrected

Low-Severity Findings	4

- Approval Abuse With Reentrant Tokens Code Corrected
- Max Loan Amounts Are Not Enforceable Code Corrected
- Missing Check Code Corrected
- Vault Pausing Functionality Code Corrected

6.1 Wrong Implementation

Correctness Medium Version 3 Code Corrected

CS-MLV2-016

When calling `RegistryV2.deployLoan()`, the caller has the opportunity to specify the `impl` address of the implementation of the loan they want to deploy. However, the loan is created as a clone of `loanImpl`. In other words, `impl` address is never used.

Code corrected:

A clone of `impl` is now created.

6.2 Loan Creation Frontrunning

Design Medium Version 1 Code Corrected

CS-MLV2-017

Lenders create their vaults by passing an arbitrary strategy contract to `RegistryV2.deployVault()`. The strategy contract then determines some important loan parameters during loan creation. Since borrowers are not able to set any slippage parameters during calls to `DelegatedLenderVault.createLoan()`, changes to these parameters that are executed after the borrower has published their transaction but before it is executed lead to undesired effects.

This is easily possible with the default `Strategy` contract by just updating the `interestRate` parameter. It is also possible to use any other contract as a Strategy or supply a proxy contract that is then upgraded to a new version with different intent.

Consider the following example:

1. A lender's vault allows to create loans for WBTC, with WETH as collateral.
2. The WETH price is 3,000 and the WBTC price 60,000.
3. The vault's strategy contains an intent with an `interestRate` of 5% and an `adjustmentFactor` of 80%. Fees are 0.
4. A borrower wants to supply 20 ETH to the vault's `createLoan()` function. The used UI simulates the transaction and displays that the borrower will receive a loan of 0.76 WBTC. The borrower submits the transaction.
5. The lender sees the borrower's transaction in the mempool and instantly creates another transaction to `Strategy.setStrategy()` that sets the interest rate to 30%. The transaction gets executed before the user's transaction but after the frontend calculated the transaction details for the user.
6. When the borrower's transaction is executed, the user now pays the higher interest rate of 30%.

Additionally, this could become an even bigger problem if the protocol contracts are deployed on a chain that does not order transactions based on FIFO but uses a public mempool and a fee market. It is also possible that Mantle changes from FIFO to a fee market in the future. In this case, borrowers can set up honeypot vaults and automatically frontrun any borrower, setting the interest rate to ~99.99%.

Code partially:

In Version 2, only whitelisted strategies can be used. Note however, the problem still persists for such strategies, if they allow for unrestricted change of their parameters. Consider the current strategy implementation. A trusted owner could change the strategy's parameters and accidentally frontrun a loan issuance.

In Version 3, the parameters of the strategies are controlled by a timelock therefore unexpected changes are highly unlikely.

6.3 Approval Abuse With Reentrant Tokens

Security Low Version 1 Code Corrected

CS-MLV2-014

`LoanRepayer.repayLoan()` allows borrowers to repay their loan by swapping collateral tokens to borrow tokens. For this, borrowers have to give approval of the collateral token to the `LoanRepayer` contract. During repayment, the collateral tokens are optimistically sent to the borrower address, after which the `sourceCollateral()` function of the `LoanRepayer` is called. This function then uses the approval granted by the borrower to transfer the previously sent collateral tokens from the borrower to an exchange.

The borrower address is requested from the `Loan` both in `repayLoan()` and `sourceCollateral()`. If the collateral token executes a callback to the receiver after a transfer has occurred, the borrower is able to call `Loan.transfer()` and move the loan to a new borrower address once the `Loan.repay()` function transfers collateral tokens to them.

When `sourceCollateral()` requests the loan data, this new address is then passed as the borrower address. This results in a `safeTransferFrom()` call from the new address. If it has any open approvals to the `LoanRepayer` in the given collateral token, the tokens can be transferred out to the user-supplied swapper address and thus be stolen.

Therefore, any approvals to the `LoanRepayer` using tokens that contain callback functionality are unsafe.

Code corrected:

`Loan.transfer()` is now non-reentrant.

6.4 Max Loan Amounts Are Not Enforceable

Design | Low | Version 1 | Code Corrected

CS-MLV2-019

Lenders can set a parameter `maxSingleLoanAmt` in their `DelegatedLenderVault` that limits the amount a single loan can reach. Borrowers can, however, always create multiple loans as the `createLoan()` function is unpermissioned.

Code correct:

Max amounts are no longer enforced. They remain in the storage and interface for backwards compatability but are always set to the maximum.

6.5 Missing Check

Design | Low | Version 1 | Code Corrected

CS-MLV2-015

When a vault has been created with a given `Strategy` and the strategy has not been initialized with `setStrategy()` yet, it is possible to create loans with `0` duration and an `adjustmentFactor` of `0`. Neither `Strategy.getIntentData()` nor `DelegatedLenderVault.createLoan()` revert in this case.

Code corrected:

Strategy parameters are now set and checked during deployment.

6.6 Vault Pausing Functionality

Design | Low | Version 1 | Code Corrected

CS-MLV2-018

`DelegatedLenderVault` exposes three different methods for disabling vaults. Lenders can:

1. Call the `pause()` function.
2. Call the `setCollectionExpiry()` function with a value smaller than the current block's timestamp.
3. Call the `setCollectionStatus()` function with a `false` value.

These checks cover similar cases, they are therefore redundant and could be removed.

Code corrected:

The functions `setCollectionExpiry()` and `setCollectionStatus()` have been removed.

7 Informational

We utilize this section to point out informational findings that are less severe than issues. These informational issues allow us to point out more theoretical findings. Their explanation hopefully improves the overall understanding of the project's security. Furthermore, we point out findings which are unrelated to security.

7.1 Code Duplication

Informational **Version 1**

CS-MLV2-003

`DelegatedLenderVault._matchBorrowParams()` contains logic that could be safely replaced by calling the `intent()` function instead as long as there is no reason why the additional custom errors should not be replicated in the `intent()` function.

7.2 Gas Optimizations

Informational **Version 1**

CS-MLV2-004

The following parts of the code can be optimized for gas efficiency:

1. `registry` in `Loan` and `DelegatedLenderVault` is stored in storage. The value can not be changed later-on and is the same across all clones deployed with a given registry contract. It can therefore be defined as `immutable` and set during deployment of the implementation contracts.
2. `IntentCollection` packing can be improved by placing the `isEnabled` boolean behind the addresses.
3. In general, some storage variable sizes can be decreased. For example, a variable holding a timestamp does not require full 256 bits of storage.
4. `PythOracle.forcePythPriceUpdate()` could be used to update the prices of two feeds at the same time since getting an oracle observation always involves two feeds at a time. Otherwise, the function is redundant as the price update can also be performed directly on the Pyth contract.
5. The conversion operations for confidence and price in `PythWrapper.isPriceInvalid()` are not necessary as the `skew` is merely a ratio of the two values and both are already in the same format. The additional checks are also performed in `getPriceInDenom()` so it would suffice to just check if the confidence is negative.
6. `oracle`, `collToken` and `borrowToken` in `Strategy` can be defined as `immutable`.
7. The call to `safeDecreaseAllowance()` in `LoanCreator.createLoan()` can be safely replaced by `safeApprove(0)`, omitting two calls to `allowance()`.
8. `LoanCreator.sourceCollateral()` checks that the vault equals `msg.sender`. However, `vault` is set arbitrarily and never used in the call which means that the check is redundant.
9. `LoanRepayer.sourceCollateral()` checks that `lrp.loan == msg.sender` and then calls `ILoan(lrp.loan).loanData()`. However, `ILoan(msg.sender).loanData()` could be called directly.

7.3 Implementations Not Initialized

Informational Version 1

CS-MLV2-005

The implementation contracts of `DelegatedLenderVault` and `Loan` are not automatically initialized, allowing third party users to set storage variables.

7.4 Incorrect Documentation

Informational Version 1

CS-MLV2-006

NatSpec of `pauseVault()` and `unpauseVault()` in `DelegatedLenderVault` mentions that the functions can be called by pause / unpause guardian roles. This is, however, incorrect, as no such roles exist.

7.5 Misleading Naming

Informational Version 1

CS-MLV2-007

`LoanCreator.executeLoans()` does not, in fact, execute loans.

7.6 Missing Event Indexes

Informational Version 1

CS-MLV2-008

The following event fields could benefit from indexation:

1. `CreateLoan.borrower` in `DelegatedLenderVault`.
2. `Withdraw.token` in `DelegatedLenderVault`.
3. `LoanCreated.loan` in `LoanCreator`.
4. `LoanRepayed.loan` in `LoanRepayer`.

It is noticeable that indexes are not set consistently. For example, `WithdrawFees.token` in `RegistryV2` is indexed while `Withdraw.token` in `DelegatedLenderVault` is not.

7.7 Missing Events

Informational Version 1 Code Partially Corrected

CS-MLV2-013

Some state-changing functions do not emit events. This can be problematic for off-chain applications trying to notice state changes. Some examples include:

1. `DelegatedLenderVault.initialize()`.
2. `DelegatedLenderVault.setCollectionExpiry()`.
3. `DelegatedLenderVault.setMinMaxLoanAmt()`.
4. `DelegatedLenderVault.setCollectionStatus()`.

5. `Loan.initialize()`.
6. `Strategy.setStrategy()`.
7. `Strategy.setOperationalParams()`.
8. `PythWrapper.initFeed()`.
9. `PythWrapper.updateFeedMaxSkew()`.
10. `PythWrapper.updateFeedTimeTolerance()`.

Code partially corrected:

Missing events (and additional events for newly introduced functions) have been added to the `Strategy` contract. The remaining events mentioned above are still missing.

7.8 Positive Pyth Exponents

Informational **Version 1**

CS-MLV2-009

`PythUtilsExtra.convertToUint()` reverts when the exponent of a Pyth feed is greater than 0. While currently no Pyth feeds contain such exponents, the Pyth team does not rule out that such feeds might be made available in the future.

7.9 Pyth Prices in Future

Informational **Version 1**

CS-MLV2-010

Since timestamps for Pyth oracle prices are not generated on-chain, it is generally possible that a timestamp of a price feed is further in the future than `block.timestamp`. It is best-practice amongst pull-based oracle providers such as Pyth or RedStone to reject prices that are too far in the future. `PythWrapper.isPriceValid` currently does not check for such cases.

7.10 Reverting Oracle Observations

Informational **Version 1**

CS-MLV2-011

`PythWrapper.getPrice()` returns a boolean indicating that something is wrong with the price feed (e.g., the price is stale). This works, however, not in all cases. During the call, `isPriceInvalid()` and `getPriceInDenom()` perform additional checks during unit conversions using the library function `PythUtilsExtra.convertToUint()`. This function reverts, for example, when a negative price is returned.

7.11 Typographical Errors

Informational **Version 1**

CS-MLV2-012

The following typographical errors have been found in the code during this review:

1. `DelegatedLenderVault` contains the following comment: "*Integratoor's Getters*".
2. `LoanCreator.sourceCollateral()` contains the following comment: "*and it we swap only the amount required*".
3. The event `LoanRepayed` in `LoanRepayer` contains the term "*repayed*".

8 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.

8.1 Denomination Asset of Price Feeds

Note Version 1

`PythWrapper` calculates the relative price of two assets by quering the individual prices of the assets against some denomination. As the owner initializes the feeds, they are responsible to make sure that all the price feeds indeed report the price feed against the same denomination asset. While Pyth currently only offers feeds in USD denomination, this might change in the future.

8.2 Misleading Interest Rate Calculation

Note Version 1

Lenders define an `interestRate` in their `Strategy` that is then used to calculate the premium for a given loan. It is, however, misleading as it does not define the actual interest rate of the loan but the share of the premium in the total amount that has to be repaid:

```
uint256 totalPremium = (loanAmount * interestRate) / INTEREST_BASE_UNIT;

...

borrowAmtReceived = loanAmount - totalPremium;
```

If, for example, the interest rate is set to 50%, the borrower has an effective interest rate of 100%.

8.3 Pyth Oracle Time Tolerances

Note Version 1

`PythWrapper` allows to set a time tolerance for each individual price feed. Prices are accepted only if the price of a feed has been observed in the given time interval. Since users can update the price feed themselves, it is important to keep the time tolerance parameters tight, especially because prices in the PythWrapper are always a combination of two feeds.

Consider the following example:

1. A vault does not charge premium and fees (for simplification of the example).
2. The vault's strategy has an `adjustmentFactor` of 95%.
3. The Pyth feeds for WETH and WBTC are initialized with a time tolerance of 2 hours.
4. Both feeds are not regularly updated on the given chain. Their last update occurred several hours ago.
5. A user creates a loan for WBTC, supplying WETH as collateral.

6. The user first has to update both price feeds to a new version. They are able to choose any of the published Pyth prices as soon as they are newer than the last published ones and not older than the 2 hour time tolerance.

    - The current spot price of WETH is 3,000. In the last two hours, the price fluctuated with a high of 3,200.
    - The current spot price of WBTC is 60,000. In the last two hours, the price fluctuated with a low of 57,000.

7. The user now chooses the most preferential prices to update the feed and then commences with creating their loan.
8. The oracle computes a price of ~0.05614. Adjusted by the adjustment factor, the strike price is ~0.05334.
9. For 20 supplied WETH, the user can borrow ~1.06666 WBTC.
10. The 20 WETH have been bought for exactly 1 BTC on the spot market. The user now made an instant profit of ~0.06666 BTC.

As long as there is enough volatility in the assets, this is possible even if WBTC and WETH followed the same curve during the last 2 hours as the user is not required to pick both prices at the same point in time.

8.4 Repayment Rounding Errors

Note **Version 1**

The amount of collateral that is returned after a repayment in `Loan` is calculated in the function `previewRepay()`. Due to the way, the amount is calculated, very small repayments can result in not collateral being returned at all:

```
(loanData.collAmt * repayAmt) / loanData.borrowAmt)
```

If there is a discrepancy between the decimals of the collateral and the borrow token, minimal repayments will lead to a result of 0 in the shown calculation if the repay amount is noticeably lower than 1 full token. Users are, however, assumed to not repay such low amounts.

8.5 Unsupported ERC-20 Tokens

Note **Version 1**

The system doesn't apply any restrictions on what tokens are used as collateral and principal tokens. However, it is important to emphasize that tokens with rebases should not be used as collateral. Moreover, tokens with negative rebases, fees on transfer or callbacks should not be used as either collateral or principal. Lenders should be mindful of the tokens they choose to allow in their intents.