STATE MIND

MYSO V2

22-05-2023 - 15-08-2023

Table of contents

1. Project Brief 7

2. Finding Severity breakdown 9

3. Summary of findings 10

4. Conclusion 10

5. Findings report 12

High

Impossible to swap when protocolFee is set 12

Inability to Disable Whitelist for Borrowers When Granted Using Signatures 13

Reuse signature mechanic 14

Collateral can be locked forever, if the loan amount is zero 15

Medium

Potential reclaiming of whitelist through signature reuse | P2Pool 16

Misuse of _repaymentScheduleCheck function 17

Insufficient check in _checkAndReturnLatestRoundData 18

Requirment to set Sequncer if project will be multichain 18

Transfers in Erc721 wrapper can be blocked 18

Griefing wrapping tokens 19

Last repayment can be blocked 20

transferFrom with IERC20 interface unsupports early ERC20 implementation 21

Users can lose tokens when minting a wrapped token 22

Compartment access after a loan default 23

Using SafeERC20 library for tokens approve 23

Medium

Singleton state violation 23

Meaningless condition in the rollback function 24

UniswapV2 LP-Token price oracle can be manipulated 24

Rounding error in repayAmount calculation 25

Possible lose of money by lenders 25

Reentrancy in borrowing process 26

Potential DOS of repay due to open repayCallback 28

Harmful condition in acceptLoanTerms 29

Borrower can avoid repayment 30

Informational

Possible excessive amount of collToken transferred in compartment contracts 31

Opportunities for gas optimizations: extended | P2Peer 32

Use safeTransferFrom() 32

Complex structure of the comment 33

Cache state variables 33

Possible redundant event emissions 34

Improper interestRatePctInBase checks 34

Incomplete whitelisting for a block | P2Peer 34

Insufficient checks for offChainQuote 35

Opportunities for gas optimizations: extended v2 | P2Peer 35

Use OpenZeppelin's Math.mulDiv() 36

Unnecessary expressions/variables v3 | P2Peer 36

Zero address check for recipient 36

Unnecessary expressions/variables v4 | P2Peer 37

Issues with TwapGetter implementation 37

Informational	Opportunities for gas optimizations | P2Pool	38
	Redundant Ownable contract implementation	38
	Unnecessary expressions/variables	38
	Erroneous conditional check	39
	Potential reversion in repaymentSchedule check	40
	Incomplete whitelisting for a block | P2Pool	41
	Redundant expressions/variables	41
	Add pagination to getFullOnChainQuoteHistory()	41
	Redundant option to set allowAllPairs to false	41
	Improper calculation of the correctness of Borrower Whitelist Status	42
	Gas optimizations	42
	Improving newOnwer checks	43
	Improper Check of Number of signatures	43
	Insufficient check in updateOnChainQuote	44
	Insufficient check in _isValidOnChainQuote	44
	Redundant variable in _checkTokensAndCompartmentWhitelist signature	45
	LoanProposalImpl has an unreasonable limit for arrangerFee	45
	Empty lenders for updateLenderWhitelist | P2Pool	45
	Any can wrap tokens from wrapper balance	46
	Interfaces don't contain methods of implementation	46
	Own implementation for common patterns	46
	Setting the same parameters for set functions	47
	UnlockCollateral lock collateral	47
	baseCurrency zero address check	48
	IStakingHelper contains function from Curve and GLP contracts	48

Informational	outOfGas for tokensCreated	48
	getReclaimableBalance has redundant arguments	49
	Redundant calculations for reclaimCollAmount	49
	UpfrontFee from non-collateral tokens	50
	Redundant safeCast	51
	Wrapped ERC redeem with allowance	51
	Unchecked maxLoan off-chain quote parameter	52
	Wrapped redeem optimization	52
	Inconsistent instructions structures	53
	SLOAD reduction via existing memory	53
	Oracle whitelist check	53
	Unnecessary logic separation	54
	Verification code duplication	54
	Address to uint160 cast	54
	Redundant SLOAD CurveLPStakingCompartment	55
	Unindexed Event topics	55
	Immutable to memory	55
	Storage to memory gas reduction	56
	Redundant MSTORE	56
	Redundant quoteTuples check	56
	Redundant reentrancy guard	57
	Misleading comment for Ownable2Step inheritance	57
	Gas optimization via unchecked	57
	Redundant isLoan field in TransferInstructions struct	57
	Redundant loan calculations for swap quote	58

Informational

Adjust subscription amount instead of revert 58

Checking protocol fee for zero 58

Optimizing protocol logic when repayment is fully converted 58

Checking for zero address before assigning 58

Missing state check 59

Unchanged value of _lenderExercisedConversion 59

Restricting length of repayments 59

Using calldata variable instead of storage one 59

Redundant approve 59

Set allowance to zero 60

Double whitelist state changing 60

Code duplication while borrowing 60

Transfer fee on protocol fee 60

Making checks before assigning 61

QuoteTuple index is not checked 61

Parameter minLoan can be zero 61

Using ERC-2098 standard for signatures 62

Misleading name of a function 62

No function that returns the length of array in a contract 62

OpenZeppelin's ECDSA library can create Ethereum Signed Message 62

Gas optimization 63

Use clone instead of cloneDeterministic to save gas 63

Infinite unsubscribe period 64

The sufficient condition is always satisfied 64

Positive and Negative MysoTokenManager hooks 64

Informational

No protection from malicious proposal 65

No option to set the exact number of subscriptions 66

Unnecessary expressions/variables v2 | P2Peer 66

Blacklisted tokens will revert the whole transaction 66

Lack of zero check 66

Adding a Sweep Function to the redeem Function of Wrapped ERC20 67

loans can be defaulted if sequencer down 67

Redundant re-entrance check 67

Calldata reduction 68

Pragma of interfaces 68

Using ReentrancyGuardUpgradeable contract 68

EnumerableSet.Address instead of mappings and arrays 68

Redundant checks 69

7. Appendix B. Slither 70

8. Appendix C. Tests 81

1. Project Brief

Title	Description
Client	MYSO
Project name	MYSO V2
Timeline	22-05-2023 - 15-08-2023
Initial commit	c0536c1ad650805bdf5d68390de0434eb570e694
Final commit	ac3f28fca0637a46d57a73331ff743dbe61cd366

Short Overview

MYSO Finance is a DeFi protocol for “Zero-Liquidation Loans” (ZLLs). Project consist of two main subsystems: Peer-to-Peer, Peer-to-Pool.

MYSO's v2 Peer-to-Peer system is a decentralized lending platform that enables lenders to offer loans to borrowers on a peer-to-peer basis. It is built on the Ethereum blockchain and consists of several smart contracts that work together to facilitate the lending process. The core components of the system are the Address Registry, Borrower Gateway, Lender Vault Factory, Lender Vault Implementation, and Quote Handler contracts.

MYSO's v2 Peer-to-Pool system is a decentralized borrowing and lending system built on the Ethereum blockchain that empowers lenders and borrowers to engage in peer-to-pool loan transactions.

Project Scope

The audit covered the following files:

- LoanProposalImpl.sol
- DataTypesPeerToPool.sol
- ILoanProposalImpl.sol
- WrappedERC20Impl.sol
- ERC721Wrapper.sol
- ChainlinkBasic.sol
- UniV2Chainlink.sol
- BorrowerGateway.sol
- VoteCompartment.sol
- CurveLPStakingCompartment.sol
- IWrappedERC20Impl.sol
- IWrappedERC721Impl.sol
- IUniV2.sol
- IBalancerAsset.sol
- IBalancerVault.sol
- IBorrowerGateway.sol
- IQuoteHandler.sol
- IMysoTokenManager.sol
- Factory.sol
- IFactory.sol
- Errors.sol
- ERC20Wrapper.sol
- DataTypesPeerToPeer.sol
- ChainlinkBasicWithWbtc.sol
- BalancerV2Looping.sol
- LenderVaultFactory.sol
- GLPStakingCompartment.sol
- BaseCompartment.sol
- IERC20Wrapper.sol
- IVaultCallback.sol
- AggregatorV3Interface.sol
- BalancerDataTypes.sol
- ILenderVaultFactory.sol
- IStakingHelper.sol
- ILenderVaultImpl.sol
- FundingPoolImpl.sol
- IFundingPoolImpl.sol
- Ownable.sol
- WrappedERC721Impl.sol
- AddressRegistry.sol
- OlympusOracle.sol
- UniV3Looping.sol
- QuoteHandler.sol
- AaveStakingCompartment.sol
- LenderVaultImpl.sol
- IERC721Wrapper.sol
- IOlympus.sol
- IAddressRegistry.sol
- ISwapRouter.sol
- IOracle.sol
- IBaseCompartment.sol
- Constants.sol

2. Finding Severity breakdown

All vulnerabilities discovered during the audit are classified based on their potential severity and have the following classification:

Severity	Description
Critical	Bugs leading to assets theft, fund access locking, or any other loss of funds to be transferred to any party.
High	Bugs that can trigger a contract failure. Further recovery is possible only by manual modification of the contract state or replacement.
Medium	Bugs that can break the intended contract logic or expose it to DoS attacks, but do not cause direct loss of funds.
Informational	Bugs that do not have a significant immediate impact and could be easily fixed.

Based on the feedback received from the Customer regarding the list of findings discovered by the Contractor, they are assigned the following statuses:

Status	Description
Fixed	Recommended fixes have been made to the project code and no longer affect its security.
Acknowledged	The Customer is aware of the finding. Recommendations for the finding are planned to be resolved in the future.

3. Summary of findings

Severity	# of Findings
Critical	0 (0 fixed, 0 acknowledged)
High	4 (4 fixed, 0 acknowledged)
Medium	20 (19 fixed, 1 acknowledged)
Informational	103 (90 fixed, 13 acknowledged)
Total	127 (113 fixed, 14 acknowledged)

4. Conclusion

During the audit of Myso V2 codebase, 146 issues were found in total:

- 5 high severity issues (5 fixed, 0 acknowledged)
- 24 medium severity issues (23 fixed, 1 acknowledged)
- 117 informational severity issues (104 fixed, 13 acknowledged)

The final reviewed commit is ac3f28fca0637a46d57a73331ff743dbe61cd366

Deployment

File name	Contract deployed on mainnet
AddressRegistry (ethereum)	0xd8b132A0abA610D0AaA18716A2b26e14141f112C
AddressRegistry (mantle)	0x196A6649e2A7eb225Ee53BA507552cCc8BcdB07a
BorrowerGateway (ethereum)	0x3EAf1BA2C14B2d353fd63c3881bbcc7583E665f9
BorrowerGateway (mantle)	0xd8b132A0abA610D0AaA18716A2b26e14141f112C

QuoteHandler (mantle)	0x3EAf1BA2C14B2d353fd63c3881bbcc7583E665f9
LenderVaultImpl (ethereum)	0x4522BF023e4ca2A3875B73EBB21696cB30EBC17d
LenderVaultImpl (mantle)	0x71E1cc2F7574C798ED893Ef04D55D1E573bE95B1
LenderVaultFactory (ethereum)	0x1874a08F7975b25944FEb989BbAaA464f61aB3bc
LenderVaultFactory (mantle)	0x4522BF023e4ca2A3875B73EBB21696cB30EBC17d

5. Findings report

HIGH-01	Impossible to swap when protocolFee is set	Fixed at 11f106

Description

During the swap process, there are currently two conditions that need to be met as part of the validation:

1. LenderVaultImpl.sol#L137-L142

```
if (
    borrowInstructions.collSendAmount <
    upfrontFee + borrowInstructions.expectedTransferFee
) {
    revert Errors.InsufficientSendAmount();
}
```

2. BorrowerGateway.sol#L276-L280

```
if (
    borrowInstructions.collSendAmount < protocolFeeAmount + upfrontFee
) {
    revert Errors.InsufficientSendAmount();
}
```

Additionally, the upfrontFee is calculated using the following formula:

```
upfrontFee =
    (borrowInstructions.collSendAmount *
        quoteTuple.upfrontFeePctInBase) /
    Constants.BASE;

// Also, we know that
quoteTuple.upfrontFeePctInBase == Constants.BASE

// Hence
upfrontFee == borrowInstructions.collSendAmount
```

Therefore, based on condition 2, it can be inferred that in order to avoid a revert, the **protocolFeeAmount** must be set to **0**. Furthermore, from condition 1, it is apparent that **borrowInstructions.expectedTransferFee** must also be set to **0**. Consequently, it becomes impossible to process the swap when there are any transfer fees or if the protocol fee is set. The underlying issue lies in the fact that for a simple loan with a **quoteTuple.upfrontFeePctInBase** value close to **Constants.BASE**, it is feasible to create the loan. However, it subsequently becomes impossible to utilize the loan due to the aforementioned reasons.

Recommendation

It is recommended to reconsider the calculation of **upfrontFee** to not solely rely on **borrowInstructions.collSendAmount**, but also take into account transfer fees and protocol fees. Alternatively, it is recommended to establish external conditions for the swap functionality to ensure its usability even in the presence of non-zero **borrowInstructions.expectedTransferFee**.

HIGH-02 | Inability to Disable Whitelist for Borrowers When Granted Using Signatures | Fixed at f73d21

Description

Currently, in the AddressRegistry contract, borrowers have the ability to update their whitelist status using a signature, which is not directly associated with quotes. All parametrs for **payloadHash** (msg.sender, whitelistedUntil, block.chainid, salt) can be reused.

Whitelist authorities have the capability to change a borrower's status, as well, but borrowers can exploit the system by backrunning the authority's call and preserving their existing whitelist status.

Borrorwers can save their status until **block.timestamp** becomes greater than **whitelistedUntil** timestamp specified in the signature.

Recommendation

We recommend implementing a map to keep track of used signatures and modifying the logic to revert the transaction if a signature has already been used. For example:

```
function claimBorrowerWhitelistStatus(
    address whitelistAuthority,
    uint256 whitelistedUntil,
    bytes calldata signature,
    bytes32 salt
) external {
    if (signatureIsInvalidated[signature]) {
        revert Errors.InvalidSignature();
    }
    signatureIsInvalidated[signature] = true;
    // ...
}
```

HIGH-03	Reuse signature mechanic	Fixed at f73d21

Description

Protocol uses signature check EIP-712 in several places:

- **Factory claimLenderWhitelistStatus** L135-L141
- **AddressRegistry claimBorrowerWhitelistStatus** L175-181
- **QuoteHandler _areValidSignatures** hanL246-L275

Moreover, **payloadHash**, **messageHash** are identical in the data structure, and **recoveredSigner** is **whitelistAuthority**. Thus, the signature issued for **claimLenderWhitelistStatus** can be used in **claimBorrowerWhitelistStatus**. The same **whitelistAuthority** for **Factory** and **AddressRegistry** can't restrict user registration in different whitelists.

Recommendation

We recommend adding the contract address in the message payload as the first argument for separating action.

```
bytes32 payloadHash = keccak256(
    abi.encode(address(this), msg.sender, whitelistedUntil, block.chainid, salt)
);
```

HIGH-04 | Collateral can be locked forever, if the loan amount is zero | Fixed at ecddfa

Description

This issue will arise if a borrower tries to borrow a loan using a quote with specific parameters. This quote should have **GeneralQuoteInfo.minLoan** and **QuoteTuple.loanPerCollUnitOrLtv** parameters equal to zero. Also, the borrower should set **BorrowTransferInstructions.minLoanAmount** to zero too.

To borrow, the borrower will call the function **BorrowerGateway.borrowWithOnChainQuote** or **BorrowerGateway.borrowWithOffChainQuote**, it calls **QuoteHandler.checkAndRegisterOffChainQuote**, and then it calls **LenderVaultImpl.processQuote**.

In this case, if these parameters are zero, all checks inside **QuoteHandler.checkAndRegisterOffChainQuote** function will pass, and inside the function **LenderVaultImpl.processQuote** variables **loanAmount** and **repayAmount** will be equal to zero. After that, the contract **BorrowerGateway** will transfer the collateral tokens to the **LenderVaultImpl** contract or the **Compartment** and will transfer zero amount loan to the borrower.

Everything would be fine if the borrower could return his collateral tokens. But he isn't able to do that in this case. In a standard case, he would call **BorrowerGateway.repay** function to return his collateral tokens. But in this case, this require statement will revert the repayment transaction because he would need to repay zero tokens out of zero amount dept.

Moreover, even the lender won't be able to get this collateral using the **LenderVaultImpl.unlockCollateral** function. Variable **totalUnlockableColl** will equal to zero because **_loan.initRepayAmount - _loan.amountRepaidSoFar** will equal zero.

A borrower and a lender can easily set parameters **GeneralQuoteInfo.minLoan**, **QuoteTuple.loanPerCollUnitOrLtv**, and **BorrowTransferInstructions.minLoanAmount** to zero values because of an error inside scripts that the lender and borrower used or some other factors.

Recommendation

Add additional checks inside the **LenderVaultImpl.processQuote** function that the loan is bigger than zero.

```
if (
    loanAmount < borrowInstructions.minLoanAmount ||
    loanAmount == 0
) {
    revert Errors.TooSmallLoanAmount();
}
```

MEDIUM-01	Potential reclaiming of whitelist through signature reuse | P2Pool	Fixed at f73d21

Description

Within the **Factory** contract, the **claimLenderWhitelistStatus()** function enables the claiming of a whitelist status using **whitelistAuthority**'s signature. Concurrently, the **whitelistAuthority** possesses the ability to update the whitelist status via the **updateLenderWhitelist()** function.

Consider a scenario where:

1. **whitelistAuthority** grants whitelist status to lender up to **timestamp = x**.
2. **lender** successfully claims the whitelist.
3. **whitelistAuthority** opts to revoke the lender's whitelist status by setting **timestamp = 0**.
4. **lender** can reclaim the whitelist using the same signature.

This sequence of actions suggests a potential vulnerability as it allows the lender to reclaim a revoked whitelist status using a previously issued signature.

Recommendation

It is recommended to implement a mechanism that ensures each signature is used only once. This could be achieved by storing signatures after they are used and adding a check to ensure that a signature is not reused.

MEDIUM-02	**Misuse of _repaymentScheduleCheck function**	**Fixed at 2b51f4**

Description

In the **LoanProposalImpl** contract, the **acceptLoanTerms()** function includes a **_repaymentScheduleCheck()** function call, which seems to be used improperly for two reasons:

1. The **_repaymentScheduleCheck()** function found here, involves a thorough examination of the entire **repaymentSchedule**. However, its application within the context of the **acceptLoanTerms()** function seems redundant, as this check is also performed during the execution of the **proposeLoanTerms()** function.
2. The function contains a conditional clause that verifies if the first repayment **dueTimestamp** is in the correct timeframe:

```
if (
    _loanTerms.repaymentSchedule[0].dueTimestamp <
    block.timestamp +
        Constants.LOAN_TERMS_UPDATE_COOL_OFF_PERIOD +
        Constants.LOAN_EXECUTION_GRACE_PERIOD +
        Constants.MIN_TIME_UNTIL_FIRST_DUE_DATE
) {
    revert Errors.FirstDueDateTooCloseOrPassed();
}
```

However, it mistakenly includes the **LOAN_TERMS_UPDATE_COOL_OFF_PERIOD** variable, which denotes a period that has already elapsed. As a result, attempts to execute **acceptLoanTerms()** will perpetually fail in cases where:

```
dueTimestamp -
    (block.timestamp +
    Constants.LOAN_TERMS_UPDATE_COOL_OFF_PERIOD +
    Constants.LOAN_EXECUTION_GRACE_PERIOD +
    Constants.MIN_TIME_UNTIL_FIRST_DUE_DATE) <= Constants.LOAN_TERMS_UPDATE_COOL_OFF_PERIOD //
during the `proposeLoanTerms()` execution
```

In such instances, the only viable remedy is to propose new loan terms.

Recommendation

It is recommended to revise the check with the following code snippet:

```
if (
    _loanTerms.repaymentSchedule[0].dueTimestamp <
    block.timestamp +
        Constants.LOAN_EXECUTION_GRACE_PERIOD +
        Constants.MIN_TIME_UNTIL_FIRST_DUE_DATE
) {
    revert Errors.FirstDueDateTooCloseOrPassed();
}
```

MEDIUM-03	Insufficient check in _checkAndReturnLatestRoundData	Fixed at a6c5f1

Description

In the function **_checkAndReturnLatestRoundData** there are checks to ensure that the data is recent enough. However, there is no check for the lower bound of **updatedAt**, and the other checks do not cover it at all.

Recommendation

We recommend adding a check for the lower bound of **updatedAt**. For example, you can set a constant **MAX_BLOCK_DIVERGENCE** and modify the code as follows:

```
if (
roundId == 0 ||
answeredInRound < roundId ||
answer < 1 ||
updatedAt > block.timestamp ||
updatedAt < block.timestamp - MAX_BLOCK_DIVERGENCE
)
```

MEDIUM-04	Requirment to set Sequncer if project will be multichain	Fixed at d1a2f2

Description

As stated in the official documentation of Chainlink, Layer 2 solutions require an additional Sequencer contract to provide data.

Recommendation

If the project will be deployed on some L2 chains, we recommend considering adapting the solution to accommodate Layer 2 scenarios with the use of a Sequencer contract.

MEDIUM-05	Transfers in Erc721 wrapper can be blocked	Fixed at 63c9e6

Description

User can withdraw a tokens from the vault only if all collections and token are transferable. However, some collections can block transfers for some tokens.

Recommendation

We recommend including functionality for withdrawing all possible tokens from the vault. For untransferable tokens there are several ways:

- emitting event about token
- adding **sweep()** function
- not burning supply token

MEDIUM-06	Griefing wrapping tokens	Fixed at d3d85f

Description

Any participant can wrap ERC20/ERC721 tokens via **AddressRegistry createWrappedTokenForERC721s** L206, **AddressRegistry createWrappedTokenForERC20s** L228. This function passes **minter == msg.sender** L216, L238. However, griever can call **ERC721Wrapper createWrappedToken** L36, **ERC20Wrapper createWrappedToken** L37 directly. In case, the user gives approval in **Tx1**, but creates a wrapped token in **Tx2**, griefer can create a wrapped token with own params **tokensToBeWrapped**, **name**, **symbol**. The User can return tokens via **redeem** function L44, L51, but the created wrapped token isn't registered in **AddressRegistry**. So, it requires a reissue of the wrapped token. This may be relevant for competing protocols that want to ruin the user experience.

Recommendation

We recommend adding **onlyAddressRegistry** check for the call **AddressRegistry createWrappedTokenForERC721s**, **AddressRegistry createWrappedTokenForERC20s**

```
function onlyAddressRegistry() internal view {
    if (addressRegistry != address(0) && msg.sender != addressRegistry) {
        revert Errors.InvalidSender();
    }
}
```

So, the variable **addressRegistry** should have an additional flag for whitelist check and should always be set in **constructor**.

MEDIUM-07	Last repayment can be blocked	Fixed at 67a1e9

Description

A borrower can repay the loan using **BorrowerGateway repay** function. **LenderVaultImpl validateRepayInfo** has a check of over repayment amount.

```
if (
    loanRepayInstructions.targetRepayAmount == 0 ||
    loanRepayInstructions.targetRepayAmount + _loan.amountRepaidSoFar >
    _loan.initRepayAmount
//        borrower can't pay more than loan --^
) {
    revert Errors.InvalidRepayAmount();
}
```

In the next step, the reclaim amount is calculated as follows.

```
reclaimCollAmount = (loan.initCollAmount * loanRepayInstructions.targetRepayAmount)
    / loan.initRepayAmount;
if (reclaimCollAmount == 0) {
    revert Errors.ReclaimAmountIsZero();
}
```

If **reclaimCollAmount > 0** therefore **initCollAmount** * **targetRepayAmount >= initRepayAmount**
In case, if a borrower returns almost a loan, the last repayment can be blocked.

```
borrower take loan
collToken.decimals = 6
loanToken.decimals = 6
initCollAmount = 1_000 * 10^6
initRepayAmount = 1_000_000_000 * 10^6
repayTokenPerCollateral = 1_000_000_000 * 10^6 / 1_000 * 10^6 = 1_000_000
repaymentReequiredMore = 1_000_000 - 1 = 1_000_000 - 1 = 999_999

borrower returns almost loan in 1 repayment
targetRepayAmount = 1_000_000_000 * 10^6 - 999_999
reclaimCollAmount = (1_000 * 10^6) * (1_000_000_000 * 10^6 - 999_999) / (1_000_000_000 * 10^6) = 999_999_999

borrower try to return the leftover loan in 2 repayment
targetRepayAmount = 999_999
reclaimCollAmount = (1_000 * 10^6) * 999_999 / (1_000_000_000 * 10^6) = 0
reclaimCollAmount always will be reverted
borrower can't return leftover loan
```

A loan proposal can determine any exchange rate **repayTokenPerCollateral** and the amount blocked value in token or $ can be big. Moreover, the loan obliges the protocol to return **collateral >= initCollAmount**.

Recommendation

We recommend calculating reclaimed amounts and returning left amounts if
targetRepaymentAmount == initRepayAmount – amountRepaidSoFar

```
uint reclaimCollAmount;
uint leftRepaymentAmount = loan.initRepayAmount - loan.amountRepaidSoFar;
if (leftRepaymentAmount == targetRepaymentAmount) {
    reclaimCollAmount = loan.initCollAmount - loan.reclaimedAmount
} else {
    reclaimCollAmount = (loan.initCollAmount * loanRepayInstructions.targetRepayAmount)
        / loan.initRepayAmount;
    if (reclaimCollAmount == 0) {
        revert Errors.ReclaimAmountIsZero();
    }
}
```

MEDIUM-08	transferFrom with IERC20 interface unsupports early ERC20 implementation	Fixed at d04329

Description

ERC20 token standard has some implementation differences. It is described in detail in **SafeERC** article LINK. Some implementations revert if they can't execute the function, while others return false. Also, an earlier ERC20 implementation may have differences in interfaces.
EIP-20 standard

```
interface IERC20 {
    function transferFrom(address from, address to, uint256 amount) external returns (bool); // <-- declared return value
}
```

Earlier tokens like USDT - 0xdac17f958d2ee523a2206206994597c13d831ec7, doesn't have a return value. However, it's crucial for protocol because USDT is the most popular stable coin in crypto.

```
interface EarlyERC20 {
    function transferFrom(address from, address to, uint256 amount) public; // <-- don't have return value
}
```

So, directly using the ERC20 interface for operation approval, transfer, and permit is non-recommended.
It is actually for:

- **WrappedERC20Impl::mint()** L132

Recommendation

We recommended using **SafeERC20** library for the transfer operation

MEDIUM-09	Users can lose tokens when minting a wrapped token	Fixed at d04329

Description

Any user can mint wrapper token using **WrappedERC20Impl::mint()** function. Also, **ERC20Wrapper::createWrappedToken** allow creating one wrapper token for single underlying token L54-L58.
Attack scenario:

1. Attacker creates **WrappedERC20Impl** token with a single underlying token. Also, the attacker can wait for users to burn all issued wrapped tokens, and **totalSupply** wrapped token temporarily becomes 0.

   ```
   Underlying - USDT
   totalInitialSupply - 1 wei
   underlyingBalance - 1 wei
   ```

2. Wait when the victim tries to mint some tokens using **WrappedERC20Impl::mint()**.

   ```
   mintAmount - 10000 * 10^18
   ```

3. Front-run victim transaction. Attacker transaction should increase the underlying token balance by a mint amount. So, the attacker directly deposits in the Wrapper contract.

   ```
   transferAmount == 10000 * 10^18
   ```

4. Contract is calculating the mint amount, but **tokenPreBal > amount** => always 0.

   ```
   _mint(
       recipient,
       currTotalSupply == 0
           ? amount
           : Math.mulDiv(amount, currTotalSupply, tokenPreBal)
   // 10000 * 1 / 10000 + 1 = 0
   );
   ```

Recommendation

We recommended adding a variable, which is tracking the virtual balance for the underlying token in **WrappedERC20Impl** contract and using it for the mint amount

MEDIUM-10 | Compartment access after a loan default | Fixed at 970e5f

Description

Currently, a borrower has full access to the compartment's token functionality even after the loan expires.

```
if (msg.sender != loan.borrower && !approved[msg.sender]) {
    revert Errors.InvalidSender();
}
```

Hence, a borrower can manage tokens that logically are not in his possession unless the lender unlocks them into the vault.

Recommendation

We recommend restricting access to the compartment's token management functionality after the loan is expired.

```
if (msg.sender != loan.borrower && !approved[msg.sender]) {
    revert Errors.InvalidSender();
}
if (block.timestamp >= loan.expiry) {
    revert Errors.LoanExpired();
}
```

MEDIUM-11 | Using SafeERC20 library for tokens approve | Fixed at 9af2fe

Description

At Line 45 **approve()** is called. It is better to use safe version to avoid problems with specific tokens, which could return bool instead of revert.

Same issue in this function, which have several approves:

- Function repayCallback()
- Function borrowCallback()
- Function repayCallback()
- Function stake()

Recommendation

It is recommended to use **forceApprove()** or **safeIncreaseAllowance()** / **safeDecreaseAllowance()** from **SafeERC20** library.

MEDIUM-12 | Singleton state violation | Fixed at 57a424

Description

Function _updateSingletonState() does not check if there is already address with one of the singleton states. So, it is possible to have two address with same singleton state. Based on comments at Lines 87-88:

```
// note (1/2): ERC721WRAPPER, ERC20WRAPPER and MYSO_TOKEN_MANAGER state can only be "occupied" by
// one addresses ("singleton state")
```

This situation breaks protocol rule.

Recommendation

It is recommended to check if wrappers and token manager are set and in case of changing states reset state for old addresses.

MEDIUM-13	Meaningless condition in the rollback function	Fixed at d9bc59

Description

In the contract **LoanProposalImpl** in the function **rollback**, there is a condition responsible for the case when a borrower doesn't want to accept a loan proposal. The condition **block.timestamp < lenderInOrOutCutoffTime** doesn't make sense because if a borrower doesn't want to accept a loan proposal after **lenderInOrOutCutoffTime**, he won't call the **finalizeLoanTermsAndTransferColl** function, and because only he can do it, everybody will have to wait for **Constants.LOAN_EXECUTION_GRACE_PERIOD** amount of time after **lenderInOrOutCutoffTime** to call the **rollback** function and unsubscribe from this loan proposal.

Recommendation

It is better to remove this condition to allow a borrower to call the **rollback** function after **lenderInOrOutCutoffTime**, not to make everyone wait for **Constants.LOAN_EXECUTION_GRACE_PERIOD** to remove their's funds from this loan proposal.

MEDIUM-14	UniswapV2 LP-Token price oracle can be manipulated	Fixed at 590348

Description

The contract is using the following formula to calculate LP-Token price:
price = 2 * (sqrt(reserve0 * reserve1) * sqrt(price0 * price1)) / totalSupply

The method used to calculate LP-Token price holds well if the reserve pools stay balanced according to constant product formula: **x * y = k**. However UniswapV2 Pair contract allows to arbitrary transfer tokens to reserve without minting new LP-Tokens or adhering to contstant product formula via **sync()**.

The attacker can increase the amount of tokens in one of the reserves by directly transferring the tokens to the UniswapV2 Pair contract and then executing **sync()** function, which will update the reserve balances.

This manipulation can only increase the amount of tokens in the reserves and consecutively the price of LP-Token, but not to decrease. So to attack the protocol, the attacker would need to use their own LP-Tokens as the collateral (not for borrowing LP-Tokens).

It should be noted that sending tokens to the UniswapV2 Pair this way won't mint any new LP-shares and the attacker won't be able to fully return their funds, however, skewing reserves in such way creates an arbitrage opportunity that the attacker can use to partially reclaim the funds sent to uniswap pool.

To summarize a potential attack scenario:

1. Mint or borrow LP-Tokens
2. Send large amount of TokenA or TokenB to UniswapV2 Pair
3. Call **sync()** on UniswapV2 Pair, imbalancing the reserves
4. Use inflated LP-Tokens to borrow assets
5. Partially return the sent tokens from UniswapV2 Pair by rebalancing reserves back from self-created arbitrage opportunity

The economic feasibility of this attack largely depends on size of reserves pre-attack and further complicated by need of having LP-Tokens to use in borrowing positions.

Recommendation

It is recommended not to use this oracle for pairs with low liquidity or ERC20 tokens with small amount of decimals.

MEDIUM-15 | Rounding error in repayAmount calculation | Fixed at 13687d

Description

In the _getLoanAndRepayAmount method of **LenderVaultImpl** contract, there is a rounding error in the price calculation:

```
loanAmount =
    (loanPerCollUnit * (collSendAmount - expectedTransferFee)) /
    (10 ** IERC20Metadata(generalQuoteInfo.collToken).decimals());

// ...

repayAmount = (loanAmount * interestRateFactor) / Constants.BASE;
```

The error occurs cause division in **loanAmount** is performed before multiplication in **repayAmount** calculation Consider the following situation

1. **loanPerCollUnit = 10**
2. **collSendAmount = 692308**
3. **interestRateFactor = 130% in BASE**
4. the resulting **repayAmount** is **7**
5. actual **repayAmount** should be **9**

Recommendation

We recommend performing multiplication before division like

```
uint256 unscaledLoanAmount = loanPerCollUnit * (collSendAmount - expectedTransferFee);
loanAmount =
    unscaledLoanAmount / (10 ** IERC20Metadata(generalQuoteInfo.collToken).decimals());
repayAmount = (unscaledLoanAmount * interestRateFactor) / Constants.BASE /
            (10 ** IERC20Metadata(generalQuoteInfo.collToken).decimals());
```

MEDIUM-16 | Possible lose of money by lenders | Fixed at 7929f8

Description

In **LoanProposalImpl.getAbsoluteLoanTerms() loanTokenDue** is considered relatively to **_finalLoanAmount**, but in case, there's a **ArrangerFee** and loanTokenDue sum is 100% not more (example from tests), lenders will lose part of their money, due to new **loanTokenDue** calculations. But this is strange, isn't it? So, sum of repayments should include all fees. In other way, lenders will constantly lose their funds after claiming repayments.
In another case, it turns out that the lenders pay for the loan service by the borrower, which is strange too.

Recommendation

We recommend redefining some logic, cause **loanTokenDue** should be calculated considered based on **totalSubscriptions**, not **_finalLoanAmount**.

MEDIUM-17	Reentrancy in borrowing process	Fixed at 173b8a

Description

The **LenderVaultImpl** contract contains a reentrancy vulnerability that allows an owner to spend a borrower's locked funds by using **Compartment** implementation, that makes call to **owner** address in its initializer. The vulnerability arises due to a possible external call between checking the available balance and sending funds to the borrower.

```
function processQuote(
    address borrower,
    DataTypesPeerToPeer.BorrowTransferInstructions calldata borrowInstructions,
    DataTypesPeerToPeer.GeneralQuoteInfo calldata generalQuoteInfo,
    DataTypesPeerToPeer.QuoteTuple calldata quoteTuple
)
    external
    returns (
        DataTypesPeerToPeer.Loan memory _loan,
        uint256 loanId,
        uint256 upfrontFee,
        address collReceiver
    )
{
    // ...


    (uint256 loanAmount, uint256 repayAmount) = _getLoanAndRepayAmount(
        borrowInstructions.collSendAmount,
        borrowInstructions.expectedTransferFee,
        generalQuoteInfo,
        quoteTuple
    ); // <-- checks the available amount of funds for the loan, taking into account locked funds

    // ...

    if (generalQuoteInfo.borrowerCompartmentImplementation == address(0)) {
        collReceiver = address(this);
        lockedAmounts[_loan.collToken] += _loan.initCollAmount;
    } else {
        collReceiver = _createCollCompartment(
            generalQuoteInfo.borrowerCompartmentImplementation,
            _loans.length
        );
        // <-- potential reentrancy: owner can provide compartment with external call in initializer.
        _loan.collTokenCompartmentAddr = collReceiver;
    }
    loanId = _loans.length;
    _loans.push(_loan);
    emit QuoteProcessed(borrower, _loan, loanId, collReceiver);
    // further loanAmount will be used to transfer to borrower without check for available balance.
}
```

The reentrancy vulnerability allows the contract owner to steal tokens provided as collateral in the **LenderVaultImpl** contract. The following steps describe how the exploit can be carried out:

1. Borrowers initiate loans by providing **token** as collateral, which is stored in the **LenderVaultImpl** balance but locked in the **lockedAmounts[token]** mapping.
2. The contract owner sends a sufficient amount of the same token to the **LenderVaultImpl** contract, causing its balance to be double the value of **lockedAmounts[token]**.
3. The owner creates a loan quote using **token** as the loan asset, with the loan amount set to **lockedAmounts[token]**. Additionally, the owner specifies a **borrowerCompartmentImplementation** that contains a call to the owner's address in its **initialize** function. To prevent front-running attempts, the owner should restrict the whitelist status for this quote.
4. The owner proceeds to borrow using the created quote by calling either the **BorrowerGateway.borrowWithOffChainQuote** or **BorrowerGateway.borrowWithOnChainQuote** function. During the borrowing process, when the compartment initializer is called, a call is made to the owner's address. The owner includes a callback function, which invokes the **LenderVaultImpl.withdraw** function with an amount of **token** equal to **lockedAmounts[token]**.
5. After the borrow completes, the **LenderVaultImpl** balance of the **token** becomes zero, indicating that all locked tokens have been withdrawn.

By exploiting this reentrancy vulnerability, the contract owner can effectively steal the tokens that were provided as collateral in the **LenderVaultImpl** contract.

Recommendation

We recommend adding check for available balance in **LenderVaultImpl.transferTo** in case of loan token transfer (set **isLoan=true** in **BorrowerGateway._processTransfers**):

```
function transferTo(
    address token,
    address recipient,
    uint256 amount,
    bool isLoan
) external {
    _senderCheckGateway();
    if (isLoan && amount > IERC20Metadata(token).balanceOf(address(this)) - lockedAmounts[token]) {
        revert Errors.InvalidTransferAmount();
    }
    IERC20Metadata(token).safeTransfer(recipient, amount);
}
```

MEDIUM-18	Potential DOS of repay due to open repayCallback	Fixed at 9af2fe

Description

In the contract **BalancerV2Looping** and **UniV3Looping** contracts, the function **repayCallback** is open and can be called by anyone. It can lead to Denial-of-Service (DOS) vulnerability, if it is called after token transfer to the contract and before its call in **BorrowerGateway** contract, because of **minSwapReceive** condition in swap.

```
        loan.collTokenCompartmentAddr == address(0)
            ? ILenderVaultImpl(lenderVault).transferTo(
                loan.collToken,
                callbackAddr == address(0) ? loan.borrower : callbackAddr,
                reclaimCollAmount
            )
            : ILenderVaultImpl(lenderVault).transferCollFromCompartment(
                loanRepayInstructions.targetRepayAmount,
                loan.initRepayAmount - loan.amountRepaidSoFar,
                loan.borrower,
                loan.collToken,
                callbackAddr,
                loan.collTokenCompartmentAddr
            ); // <-- can be possibility to call repayCallback after transfer
        if (callbackAddr != address(0)) {
            IVaultCallback(callbackAddr).repayCallback(loan, callbackData); // <-- will revert due to minSwapReceive
condition
        }
```

A possible scenario could be the use of a compartment **CurveLPStakingCompartment**. In this compartment, after collateral token transfer comes the transfer of reward tokens. In case of malicious reward tokens, that have external calls during **safeTransfer** it is possible to make call to **repayCallback**.

Recommendation

We recommend restricting the **repayCallback** function to only be callable by the **BorrowerGateway** contract.

MEDIUM-19	Harmful condition in acceptLoanTerms	Fixed at d9bc59

Description

In the function **LoanProposalImpl.acceptLoanTerms** there is a condition:

```
function acceptLoanTerms(uint256 _loanTermsUpdateTime) external {
    // ...
    if (totalSubscriptions < _loanTerms.minTotalSubscriptions) { <-- the condition doesn't allow to accept terms due to
insufficient number of subscriptions
        revert Errors.NotEnoughSubscriptions();
    }
    // ...
}
```

This condition isn't required, because lenders can subscribe to proposal after loan terms is accepted. Moreover, is harmful because many lenders may not subscribe to proposal until it is accepted, because unaccepted loan terms can be changed and lender will have to unsubscribe in limited time period. Otherwise, their funds will be used in unwanted loan proposal. So it is better to remove the condition making process of accepting terms simpler.

Recommendation

We recommend removing the harmful condition.

MEDIUM-20	Borrower can avoid repayment	Acknowledged

Description

Loan proposal calculates absolute amounts for loan proposal in **getAbsoluteLoanTerms** L544. **loanTokenDue** only has zero value check L636-L638. So, **loanTokenDue**, **totalSubscription** can be value**[0, 2 ^ 256 – 1]**. However, the arranger can create a proposal with incorrect **loanTokenDue**, **totalSubscription** which truncates the absolute value for **loanTokenDue**. Thus, it allows the borrower to avoid repayment.

Code of calculation

```
_tmpLoanTerms.repaymentSchedule[i].loanTokenDue = SafeCast.toUint128(
    (_finalLoanAmount * _tmpLoanTerms.repaymentSchedule[i].loanTokenDue)
    / Constants.BASE
);
```

Test scenario with the transaction

```
TestToken
decimals = 6

Case 1
finalLoanAmount = 9 * 10 ^ 6
loanTokenDue = 1 * 10 ^ 11
Constants.BASE = 10 ^ 18

brokenRepayment
brokenRepayment.reploanTokenDue = (9 * 10 ^ 6) * (1 * 10 ^ 11) / 10 ^ 18 = 0.9 = 0
```

Any combination with **finalLoanAmount** * **loanTokenDue < 10 ^ 18** leads to truncation

Recommendation

We recommend adding a zero check for **loanTokenDue**.

```
uint128 absoluteLoanTokenDue = SafeCast.toUint128(
    (_finalLoanAmount * _tmpLoanTerms.repaymentSchedule[i].loanTokenDue)
    / Constants.BASE
);
uint256 relativeLoanTokenDue = _tmpLoanTerms.repaymentSchedule[i].loanTokenDue;
if (relativeLoanTokenDue != 0 absoluteLoanTokenDue == 0) { // <-- zero loanTokenDue means truncation
    revert Errors.LoanTokenDueIsZero();
}
```

INFORMATIONAL-01	Possible excessive amount of collToken transferred in compartment contracts	Fixed at 4f1fb5

Description

During the analysis of the **CurveLPStakingCompartment** and **GLPStakingCompartment** contracts, it has been identified that the transfer mechanism for hardcoded tokens may result in the borrower receiving more tokens than expected if the **collToken** is the same as the hardcoded one.

Let's consider the following scenarios to illustrate this issue:

1. Scenario: **GLPStakingCompartment** with **WETH** as `collToken
    - The lender selects the **GLPStakingCompartment** as the compartment and sets **WETH** as the **collToken**.
    - The borrower initiates a loan and decides to repay 50% of the **loanToken**.
    - In the **_transferCollFromCompartment** function, 50% of the entire **collToken** balance is transferred.
    - Subsequently, an additional transfer of 50% of the initial **WETH** balance is made, which effectively transfers 25% of the initial **WETH** balance.
    - Consequently, the borrower receives 75% of their **collToken** (**WETH**) while only repaying 50% of the **loanToken**.
2. Scenario: **CurveLPStakingCompartment** with various token configurations
    - Case 1: **lp_token** (**collToken**) is **CRV**
    - Case 2: **lp_token** (**collToken**) is some **reward_token**
    - Case 3: **CRV** is some **reward_token**
    - Case 4: Duplicate **reward_token** entries exist

Recommendation

It is recommended to implement a mapping structure to control the allowed **collToken** tokens for each compartment and introduce balance caching mechanisms to accurately track the balances of the tokens.

INFORMATIONAL-02	Opportunities for gas optimizations: extended | P2Peer	Fixed at b7ef17

Description

1. Avoiding Redundant Defaults: Setting variables to their default values may not be necessary and could be avoided to optimize gas consumption:
   - AddressRegistry.sol#L105
   - AddressRegistry.sol#L142
   - AddressRegistry.sol#L258
   - LenderVaultImpl.sol#L65
   - LenderVaultImpl.sol#L251
   - LenderVaultImpl.sol#L347
   - QuoteHandler.sol#L261
   - QuoteHandler.sol#L349
   - CurveLPStakingCompartment.sol#L212
   - ChainlinkBasic.sol#L33
   - UniV2Chainlink.sol#L36
   - ERC20Wrapper.sol#L54
   - WrappedERC20Impl.sol#L37
   - WrappedERC20Impl.sol#L57
   - ERC721Wrapper.sol#L53
   - ERC721Wrapper.sol#L71
   - WrappedERC721Impl.sol#L33
   - WrappedERC721Impl.sol#L48
   - WrappedERC721Impl.sol#L49
2. Redundant calculations to perform a comparison:
   - LenderVaultImpl.sol#L291
3. Redundant conversions to perform a comparison:
   - QuoteHandler.sol#L266
   - ERC20Wrapper.sol#L63
   - ERC721Wrapper.sol#L66
4. Redundant data are given: It could be changed to "":
   - BalancerV2Looping.sol#L43
   - BalancerV2Looping.sol#L83

Recommendation

It is recommended to implement the suggested changes.

INFORMATIONAL-03	Use safeTransferFrom()	Fixed at 6d0060

Description

The **WrappedERC721Impl** contract currently has a **redeem** function that transfers NFTs from the contract to the burner. However, it does not have a check in place to ensure that if the burner is a contract, it is properly prepared to receive the NFTs. This could result in the NFTs being lost forever.

Recommendation

It is recommended to use the **safeTransferFrom()** function instead of **transferFrom()**.

INFORMATIONAL-04	Complex structure of the comment	Fixed at b7ef17

Description

The QuoteHandler contract includes a comment that has a complex structure, making it difficult to understand its meaning at first glance. The comment is as follows:

```
// If the oracle address is set, the LTV can only be set to a value > 1 (undercollateralized)
// when there is a specified whitelist authority address.
// Otherwise, the LTV must be set to a value <= 100% (overcollateralized).
```

This comment represents a conditional tree structure that is not immediately apparent from the comment alone. The conditional tree is:

```
if (onChainQuote.generalQuoteInfo.oracleAddr != address(0)) {
    if (onChainQuote.generalQuoteInfo.whitelistAuthority == address(0)) {
        if (onChainQuote.quoteTuples[k].loanPerCollUnitOrLtv > Constants.BASE) {
            return false;
        } else {
            // it is fine
        }
    } else {
        // LTV is any
    }
} else {
    // loanPerCollUnit is any
}
```

Recommendation

It is recommended to revise and clarify the comment to explicitly convey the conditional logic.

INFORMATIONAL-05	Cache state variables	Fixed at da738f

Description

Reading from state variables can be an expensive operation, and there are several places in the codebase where state variables could be cached to optimize gas costs and improve performance. The following locations can benefit from caching state variables:

1. Cache **vaultAddr**
   - CurveLPStakingCompartment.sol#L165
   - CurveLPStakingCompartment.sol#L195
   - CurveLPStakingCompartment.sol#L203
2. Use **_liqGaugeAddr** instead of **liqGaugeAddr**
   - CurveLPStakingCompartment.sol#L56
3. Use **msg.sender** instead of **vaultAddr**
   - BaseCompartment.sol#L60
4. Use **msg.sender** instead of **vaultAddr**
   - BaseCompartment.sol#L60

Recommendation

It is recommended to cache the state variables at the specified locations.

INFORMATIONAL-06 | Possible redundant event emissions | Fixed at f63c85

Description

The current codebase contains potential instances of redundant event emissions in multiple locations:

1. **setAllowedTokensForCompartment():AllowedTokensForCompartmentUpdated** - The code does not include a check to verify if the allowances were actually modified.
2. **updateBorrowerWhitelist():BorrowerWhitelistUpdated** - There is no validation performed to confirm if the length of the whitelist is zero before emitting the event.
3. **addSigners():AddedSigners** - Similar to the previous case, the code lacks a length check for the signers before emitting the event.

Recommendation

It is recommended to implement the following checks before emitting the respective events.

INFORMATIONAL-07 | Improper interestRatePctInBase checks | Fixed at 5aa38e

Description

In DataTypesPeerToPeer.sol at line 35, the **interestRatePctInBase** variable is defined to allow a value of **-BASE**. However, there are several places in the codebase where this value is not properly checked:

- QuoteHandler.sol#L368 - It should be **int(Constants.BASE) <** instead of **int(Constants.BASE) <=**.
- LenderVaultImpl.sol#L432 - It should be **if (_interestRateFactor < 0) {** instead of **if (_interestRateFactor <= 0) {**.

Recommendation

It is recommended to make the following changes.

INFORMATIONAL-08 | Incomplete whitelisting for a block | P2Peer | Fixed at f73d21

Description

Within the **AddressRegistry** contract, the function **isWhitelistedBorrower()** indicates whether an address is currently whitelisted. However, when inspecting **claimBorrowerWhitelistStatus()**, we observe that while the whitelist status can be claimed for the current block, **isWhitelistedBorrower()** would still return false for this block.

Recommendation

It is recommended to adjust to consider the block timestamp as part of the whitelist validation. Here's a suggested code revision:

```
return
    _borrowerWhitelistedUntil[whitelistAuthority][borrower] >=
    block.timestamp;
```

INFORMATIONAL-09	Insufficient checks for offChainQuote	Fixed at 1d6ea7

Description

The **QuoteHandler** contract includes a function named **_isValidOnChainQuote** that contains various checks for **onChainQuote** objects. However, not of them are applied throughout the entire flow for **offChainQuote** objects:

-
```
if (
    onChainQuote.generalQuoteInfo.maxLoan == 0 ||
    onChainQuote.generalQuoteInfo.minLoan >
    onChainQuote.generalQuoteInfo.maxLoan
) {
    return false;
}
```

Recommendation

It is recommended to add the necessary checks to the flow for **offChainQuote** as well. This will help maintain consistency and ensure that the appropriate validations are performed for both types of quotes.

INFORMATIONAL-10	Opportunities for gas optimizations: extended v2 | P2Peer	Fixed at 6da731

Description

1. Redundant staticcall: It could be cached:
    - UniV2Chainlink.sol#L109 - it is constantly equal to **18**
2. Avoiding Redundant Defaults: Setting variables to their default values may not be necessary and could be avoided to optimize gas consumption:
    - CurveLPStakingCompartment.sol#L169

Recommendation

It is recommended to implement the suggested changes.

INFORMATIONAL-11	Use OpenZeppelin's Math.mulDiv()	Fixed at 6d0060

Description

There are several places that could theoretically revert because of overflow in your codebase. They include:

- LenderVaultImpl.sol#L419
- LenderVaultImpl.sol#L436
- BaseCompartment.sol#L46
- CurveLPStakingCompartment.sol#L112
- CurveLPStakingCompartment.sol#L191
- CurveLPStakingCompartment.sol#L209
- CurveLPStakingCompartment.sol#L223
- GLPStakingCompartment.sol#L46
- OlympusOracle.sol#L55
- ChainlinkBasic.sol#L62
- UniV2Chainlink.sol#L70
- UniV2Chainlink.sol#L106
- WrappedERC20Impl.sol#L61

Recommendation

It is recommended to use OpenZeppelin's **Math.mulDiv()** function that allows 512-bit math.

INFORMATIONAL-12	Unnecessary expressions/variables v3 | P2Peer	Fixed at 02095d

Description

There are several unnecessary expressions or variables in your codebase. They include:

- ERC20Wrapper.sol#L93 - **addressRegistry != address(0) &&**
- ERC721Wrapper.sol#L99 - **addressRegistry != address(0) &&**
- LenderVaultImpl.sol#L399 - **!_addressRegistry.isWhitelistedERC20(tokens[i])** (token could be whitelisted but then removed from the list)

Recommendation

It is recommended to remove these unnecessary expressions or variables.

INFORMATIONAL-13	Zero address check for recipient	Fixed at 9cebc0

Description

There are no zero address checks for the **recipient** variable in the following functions:

- WrappedERC20Impl.sol#L54 - **redeem()**
- WrappedERC721Impl.sol#L47 - **redeem()**

Recommendation

It is recommended to add zero address checks for the recipient variable in these functions to ensure the validity of the recipient address.

INFORMATIONAL-14	Unnecessary expressions/variables v4 | P2Peer	Fixed at a477b6

Description

There are several unnecessary expressions or variables in your codebase. They include:

- TwapGetter.sol#L54-L56 - **(, tick, lastIndex, , , ,) = IUniswapV3Pool(uniswapV3Pool).slot0();**
- FundingPoolImpl.sol#L23 - **mapping(address => uint256) public subscriptionUnlockTime;**

Recommendation

It is recommended to remove these unnecessary expressions or variables.

INFORMATIONAL-15	Issues with TwapGetter implementation	Fixed at a477b6

Description

During the audit of the **TwapGetter** contract, the following incorrectnesses were identified:

1. In the function **getSqrtTwapX96()**, there is a check for preventing overflow of the **twapInterval** variable over **int32** type, as seen in this check. However, this check is currently ineffective and does not provide the intended protection against overflow. As a result, the function may produce incorrect results.
2. The function **getPriceFromSqrtPriceX96()** has a potential overflow issue. It does not follow the implementation used by Uniswap in their official repository (OracleLibrary.sol#L49), which could lead to transaction revert.

Recommendation

It is recommended to address the following issues in the TwapGetter contract:

1. Revise the check in the **getSqrtTwapX96()** function to ensure it effectively prevents overflow of the **twapInterval** variable. Thoroughly test the function to verify that it produces accurate results and handles all possible edge cases.
2. Align the implementation of the **getPriceFromSqrtPriceX96()** function with the official implementation used by Uniswap. This will help ensure consistent and correct computation of prices and avoid potential overflow issues.

INFORMATIONAL-16	Opportunities for gas optimizations | P2Pool	Fixed at 1d6ea7

Description

1. Iteration Incrementor: Consider using ++i instead of i++ to potentially optimize gas usage:
   - Factory.sol#L183
   - LoanProposalImpl.sol#L212
   - LoanProposalImpl.sol#L590
   - LoanProposalImpl.sol#L647
2. Array Length Caching: Rather than fetching the array length each time, it could be more efficient to cache this value:
   - Factory.sol#L170
   - LoanProposalImpl.sol#L202
   - LoanProposalImpl.sol#L571
   - LoanProposalImpl.sol#L635
3. Avoiding Redundant Defaults: Setting variables to their default values may not be necessary and could be avoided to optimize gas consumption:
   - Factory.sol#L170
   - LoanProposalImpl.sol#L202
   - LoanProposalImpl.sol#L571
   - LoanProposalImpl.sol#L635
4. In **LoanProposalImpl.proposeLoanTerms()** there's a resetting storage value, which can be the same. It can be optimized by adding **if** condition. This will save a little bit gas, during the function call.

Recommendation

It is recommended to implement the suggested changes.

INFORMATIONAL-17	Redundant Ownable contract implementation	Fixed at 49ee34

Description

Your Ownable contract shares a very similar implementation to the Ownable2Step contract provided by OpenZeppelin. This redundancy can be avoided.

Recommendation

It is recommended to directly use OpenZeppelin's Ownable2Step contract.

INFORMATIONAL-18	Unnecessary expressions/variables	Fixed at cfb4af

Description

There are several unnecessary expressions or variables in your codebase. They include:

- Ownable.sol#L56 - **_newOwnerProposal == address(this)**
- Factory.sol#L48 - **_collToken == address(0)**
- Factory.sol#L31 - **Ownable()**
- Factory.sol#L136 - **salt**
- FundingPoolImpl.sol#L37 - **_factory == address(0)**
- LoanProposalImpl.sol#L49 - **_factory == address(0)**
- LoanProposalImpl.sol#L50 - **_fundingPool == address(0)**
- LoanProposalImpl.sol#L174 - **totalSubscriptions > _unfinalizedLoanTerms.maxTotalSubscriptions**

Recommendation

It is recommended remove these unnecessary expressions or variables.

INFORMATIONAL-19	Erroneous conditional check	Fixed at d9bc59

Description

In the **LoanProposalImpl** contract, the **acceptLoanTerms()** function includes a conditional check. The purpose of this check appears to be to prevent a borrower from accepting loan terms that were set by the arranger prematurely. However, this condition doesn't function as intended due to a potential front-running issue. In this scenario, the borrower's **acceptLoanTerms()** transaction could be prioritized over the arranger's **proposeLoanTerms()** transaction, thereby undermining the effectiveness of the check.

Recommendation

It is recommended to:

1. Eliminate this check, given that it currently doesn't serve its intended purpose effectively.
2. Consider adding gap in the check to safeguard the arranger's transaction, as shown in the example below:

```
if (
    block.timestamp <
    lastLoanTermsUpdateTime +
        Constants.LOAN_TERMS_UPDATE_COOL_OFF_PERIOD +
        Constants.GAP
) {
    revert Errors.WaitForLoanTermsCoolOffPeriod();
}
```

INFORMATIONAL-20	Potential reversion in repaymentSchedule check	Fixed at e5f656

Description

In the **LoanProposalImpl** contract, the **_repaymentScheduleCheck()** function performs a check on **repaymentSchedule**. It starts iterating from index zero with the **prevDueDate** not set, which can cause the function to revert under specific circumstances. For instance:

```
Constants.LOAN_TERMS_UPDATE_COOL_OFF_PERIOD + Constants.LOAN_EXECUTION_GRACE_PERIOD +
Constants.MIN_TIME_UNTIL_FIRST_DUE_DATE = 2 days + 1 hours
Constants.MIN_TIME_BETWEEN_DUE_DATES = 7 days

block.timestamp = 0;
repaymentSchedule[0].dueTimestamp at (2 days + 1 hours, 7 days)

// will revert
if (
    currDueDate < prevDueDate + Constants.MIN_TIME_BETWEEN_DUE_DATES
) {
    revert Errors.InvalidDueDates();
}
```

Recommendation

It is recommended to revise the loop in the check. Below is a code snippet illustrating how to rectify this issue:

```
        uint256 prevDueDate = repaymentSchedule[0].dueTimestamp;
        if (repaymentSchedule[0].loanTokenDue == 0) {
            revert Errors.LoanTokenDueIsZero();
        }
        uint256 currDueDate;
        for (uint i = 1; i < repaymentSchedule.length; ) {
            if (repaymentSchedule[i].loanTokenDue == 0) {
                revert Errors.LoanTokenDueIsZero();
            }
            currDueDate = repaymentSchedule[i].dueTimestamp;
            if (
                currDueDate < prevDueDate + Constants.MIN_TIME_BETWEEN_DUE_DATES
            ) {
                revert Errors.InvalidDueDates();
            }
            prevDueDate = currDueDate;
            unchecked {
                i++;
            }
        }
```

INFORMATIONAL-21	Incomplete whitelisting for a block | P2Pool	Fixed at f73d21

Description

Within the **Factory** contract, the function **isWhitelistedLender()** indicates whether an address is currently whitelisted. However, when inspecting **claimLenderWhitelistStatus()**, we observe that while the whitelist status can be claimed for the current block, **isWhitelistedLender()** would still return false for this block.

Recommendation

It is recommended to adjust to consider the block timestamp as part of the whitelist validation. Here's a suggested code revision:

```
return
    _lenderWhitelistedUntil[whitelistAuthority][lender] >=
    block.timestamp;
```

INFORMATIONAL-22	Redundant expressions/variables	Fixed at ac3f28

Description

There are several redundant expressions or variables in your codebase. They include:

- AddressRegistry.sol#L30 - **address public quotePolicyManager;**
- LenderVaultImpl.sol#L417-L419 - **function totalNumSigners() external view returns (uint256)**

Recommendation

It is recommended remove these redundant expressions or variables.

INFORMATIONAL-23	Add pagination to getFullOnChainQuoteHistory()	Fixed at ac3f28

Description

There are functions that could revert because of an out-of-gas error:

- QuoteHandler.sol#L295 - **getFullOnChainQuoteHistory()**

Recommendation

It is recommended to add pagination to these functions to avoid potential out-of-gas errors.

INFORMATIONAL-24	Redundant option to set allowAllPairs to false	Fixed at ac3f28

Description

Within the **setGlobalPolicy()** function, there exists an option to set the global policy with the **allowAllPairs** parameter set to **false**. However, considering the flow of the **isAllowed()** function, if **globalPolicy.allowAllPairs** is **false**:

1. When a **pairPolicy** is present, it takes precedence.
2. When no **pairPolicy** is present, a given quote is disallowed.

Consequently, the utilization of **globalPolicy** becomes obsolete when **allowAllPairs** is set to **false**.

Recommendation

It is recommended to conduct a review of the **globalPolicy** workflow, followed by appropriate adjustments. For instance, consider removing the **allowAllPairs** parameter from **globalPolicy**.

INFORMATIONAL-25	Improper calculation of the correctness of Borrower Whitelist Status	Fixed at f73d21

Description

In AddressRegistry.sol there is view function **isWhitelistedBorrower** that checks if a borrower is whitelisted. If we call this function when **block.timestamp == _borrowerWhitelistedUntil[whitelistAuthority][borrower]**, it will return false. But in function **claimBorrowerWhitelistStatus** we can update whitelist status until **block.timestamp** and it will pass all checks.

Recommendation

We recommend unifying this check and changing
_borrowerWhitelistedUntil[whitelistAuthority][borrower] > block.timestamp; to
_borrowerWhitelistedUntil[whitelistAuthority][borrower] => block.timestamp;
Alternatively, you can forbid updating the whitelist status for the current **block.timestamp**.

INFORMATIONAL-26	Gas optimizations	Partially fixed at https://github.com/mysofinance/v2/commit/ac3f28fca0637a46d57a73

Description

1. Use **++i** instead of **i++**

- ERC20Wrapper.sol#L80
- WrappedERC20Impl.sol#L40
- WrappedERC20Impl.sol#L65
- ERC721Wrapper.sol#L92
- WrappedERC721Impl.sol#L36
- WrappedERC721Impl.sol#L58
- WrappedERC721Impl.sol#L65
- AddressRegistry.sol#L105 here better to use unchecked {++i}
- AddressRegistry.sol#L150
- AddressRegistry.sol#L271
- LenderVaultImpl.sol#L85
- LenderVaultImpl.sol#L261
- QuoteHandler.sol#L272

2.Caching array length

- ERC20Wrapper.sol#L54
- WrappedERC20Impl.sol#L57
- ERC721Wrapper.sol#L53
- ERC721Wrapper.sol#L71
- WrappedERC721Impl.sol#L48-49
- AddressRegistry.sol#L105
- AddressRegistry.sol#L142
- AddressRegistry.sol#L258
- LenderVaultImpl.sol#L65
- LenderVaultImpl.sol#L251
- LenderVaultImpl.sol#L269
- QuoteHandler.sol#L261
- QuoteHandler.sol#L349

Recommendation

We recommend optimizing the following areas.

INFORMATIONAL-27	Improving newOnwer checks	Fixed at 704c5d

Description

In the provided function, it is advisable to include additional checks to ensure that **newOwner** is not equal to **owner** and **newOwner** is not equal to 0.

Recommendation

We recommend adding two checks in function. Example:

```
function _newOwnerProposalCheck(
    address _newOwnerProposal
) internal view virtual {
    if (
        _newOwnerProposal == address(this) || _newOwnerProposal == _newOwner ||
        _newOwnerProposal == _owner || _newOwnerProposal == address(0)
    ) {
        revert Errors.InvalidNewOwnerProposal();
    }
}
```

INFORMATIONAL-28	Improper Check of Number of signatures	Fixed at f73d21

Description

In function _areValidSignatures() we have check of length for v array and **minNumOfSigners()** variable from Vault contract. But quote will be valid with more than minimum number of signers.

Recommendation

We recommend changing conditions to:

```
if (
    v.length != r.length ||
    v.length != s.length ||
    v.length < ILenderVaultImpl(lenderVault).minNumOfSigners()
) {
    return false;
}
```

INFORMATIONAL-29	Insufficient check in updateOnChainQuote	Fixed at 227a2c

Description

There is no check in **updateOnChainQuote** to verify if **newOnChainQuote** has already been added, unlike the **addOnChainQuote** function which includes such a check.

Recommendation

We recommend adding check, for example:

```
if (!isOnChainQuoteFromVault[onChainQuoteHash]) {
    revert Errors.UnknownOnChainQuote();
}
isOnChainQuoteFromVault[onChainQuoteHash] = false;
emit OnChainQuoteDeleted(lenderVault, onChainQuoteHash);
onChainQuoteHash = _hashOnChainQuote(newOnChainQuote);
if(isOnChainQuoteFromVault[onChainQuoteHash]) { <-- add this
    revert Errors.OnChainQuoteAlreadyAdded();
}
isOnChainQuoteFromVault[onChainQuoteHash] = true;
emit OnChainQuoteAdded(lenderVault, newOnChainQuote, onChainQuoteHash);
```

INFORMATIONAL-30	Insufficient check in _isValidOnChainQuote	Fixed at 227a2c

Description

As stated in the comments

```
    // If the oracle address is set, the LTV can only be set to a value > 1 (undercollateralized)
    // when there is a specified whitelist authority address.
    // Otherwise, the LTV must be set to a value <= 100% (overcollateralized).
```

However, there is no check implemented for the "Otherwise" case, where the LTV must be set to a value less than or equal to 100% (overcollateralized). "Oracle address setted" and "specifed whitelist authority address" case handled correctly.

Recommendation

We recommend making the comments and code consistent, adding check for the "Otherwise" case.

INFORMATIONAL-31	Redundant variable in _checkTokensAndCompartmentWhitelist signature	Fixed at 4c36a3

Description

There is **address _addressRegistry** in the calldata, but **addressRegistry** is immutable and may not be pushed to the calldata

Recommendation

We recommend erasing **_addressRegistry** from signature. For example:

```
function _checkTokensAndCompartmentWhitelist(
    address collToken,
    address loanToken,
    address compartmentImpl
) internal view {
    IAddressRegistry registry = IAddressRegistry(addressRegistry);
```

INFORMATIONAL-32	LoanProposalImpl has an unreasonable limit for arrangerFee	Fixed at 7929f8

Description

LoanProposalImpl initialize requires that **arrangerFee** must be in range **[Constants.MIN_ARRANGER_FEE, Constants.MAX_ARRANGER_FEE]** L56-L57. An upperbound check is reasonable, because **Myso** protects users from unfair fees, but lower bounds check is unreasonable because the arranger can have rewards in other protocols. So, the arranger readies to make work without the arranger fee. It makes the proposal more attractive to lenders and borrowers.

Recommendation

We recommend removing the lower bound check for **arrangerFee**

INFORMATIONAL-33	Empty lenders for updateLenderWhitelist | P2Pool	Fixed at 6ab0ef

Description

Factory updateLenderWhitelist allows updating the whitelist for **whitelistAuthority**. However, if **address[] calldata lenders** is empty, the function doesn't do useful work, wasting time and gas L170. In addition, the function logs **LenderWhitelistUpdated** without a real updating whitelist L186.

```
function updateLenderWhitelist(
    address[] calldata lenders,
    uint256 whitelistedUntil
) external {
    for (uint i = 0; i < lenders.length; ) { // <-- do nothing with empty lenders
    ...
    }
    emit LenderWhitelistUpdated(msg.sender, lenders, whitelistedUntil);
    // ^-- emit event without update
}
```

Recommendation

We recommend adding a guard check for empty lenders

INFORMATIONAL-34	Any can wrap tokens from wrapper balance	Fixed at d3d85f

Description

Sometimes users make mistakes and transfer tokens to the wrong address. Participants can wrap ERC20/ERC721 tokens using **ERC721Wrapper createWrappedToken** L36, **ERC20Wrapper createWrappedToken** L37 directly. If the wrapper contains a token on its own balance, any participant can call **createWappedToken** and pass **minter == wrapper_contract_addess** and create a wrapped token.

Recommendation

We recommended extending minter address checks L42, L43

```
if (minter == address(0) II minter == address(this)) {
    revert Errors.InvalidAddress();
}
```

INFORMATIONAL-35	Interfaces don't contain methods of implementation	Fixed at d3d85f

Description

Smart contract interfaces help developers integrate the protocol without diving into the implementation. The preferred approach is adding all public variables and functions to the interface. Developers also can use all function contracts. So, developers can use all the functional contracts.

Recommendation

We recommended extending interfaces:

- **IERC20Wrapper** add function **wrappedErc20Impl**, **addressRegistry**, **tokensCreated**
- **IERC20Wrapper** add function **wrappedErc20Impl**, **addressRegistry**, **tokensCreated**
- **IAddressRegistry** add function **erc721Wrapper**, **erc20Wrapper**

INFORMATIONAL-36	Own implementation for common patterns	Fixed at f15cd1

Description

AddressRegistry has a custom implementation of **Initializable** pattern L22, L45, L65, L368. However, other contracts use **@openzeppelin/contracts/proxy/utils/Initializable.sol** implementation, and **AddressRegistry** can use **@openzeppelin/contracts/proxy/utils/Initializable.sol** implementation for its own initialization.

Recommendation

We recommend using **@openzeppelin/contracts/proxy/utils/Initializable.sol** for **AddressRegistry**

INFORMATIONAL-37	Setting the same parameters for set functions	Fixed at 6d0060

Description

Myso protocol has set functions, which deny setting the same parameters. For example

- **AddressRegistry setWhitelistState** L109, L329
- **Factory setArrangerFeeSplit** L121
- **LenderVaultImpl setMinNumOfSigners** L242
- and others.

However, we found several set functions, which alloy setting the same parameters:

- **AddressRegistry setAllowedTokensForCompartment** allow install **allowTokensForCompartment** flag L143-L151
- **BorrowerGateway setProtocolFee** allow install **_newFee** L201-L203

Re-installing the same parameters wastes gas and time, emits incorrect set/update events. It can also be avoided by checking the current state of the contract.

Recommendation

We recommend adding the same value checks for **setAllowedTokensForCompartment**, **setProtocolFee** functions

INFORMATIONAL-38	UnlockCollateral lock collateral	Fixed at 194498

Description

If a borrower has a defaulted loan, a lender may unlock collateral using **LenderValutImpl unlockCollateral**. Unlock amount for a single loan is calculated as follows.

```
unlockAmount = ((_loan.initRepayAmount - _loan.amountRepaidSoFar) * _loan.initCollAmount) / _loan.initRepayAmount;
```

If **unlockAmount > 0** therefore

(_loan.initRepayAmount – _loan.amountRepaidSoFar) * _loan.initCollAmount >= _loan.initRepayAmount.

In case, if a borrower returns almost a loan, but **unlockAmount** may be 0.

```
borrower take loan
collToken.decimals = 6
loanToken.decimals = 6
initCollAmount = 1_000 * 10^6
initRepayAmount = 1_000_000_000 * 10^6
repayTokenPerCollateral = 1_000_000_000 * 10^6 / 1_000 * 10^6 = 1_000_000
leftLoanAmountReequiredMoreThan = 1_000_000 - 1 = 1_000_000 - 1 = 999_999

borrower returns almost loan in 1 repayment
amountRepaidSoFar = 1_000_000_000 * 10^6 - 999_999
lockedCollateral = 1

the loan was defaulted because the borrower did not paid left amount

lender try unlockCollateral
unlockAmount = (1_000_000_000 * 10^6 - 1_000_000_000 * 10^6 + 999_999) * (1_000 * 10^6) / (1_000_000_000 *
10^6) = 0
```

The loan proposal can determine any exchange rate **repayTokenPerCollateral** and it may affect to unlock amount.

Recommendation

We recommend calculating unlock amounts and returning left amounts

INFORMATIONAL-39	baseCurrency zero address check	Fixed at a6c5f1

Description

ChainlinkBasic constructor L20 allow pass **address(0)** for **baseCurrency**. However, the constructor has a non-0 check L38-L40 for **baseCurrencyUnit**. So, we assume that **baseCurrency** can't be **address(0)**

Recommendation

We recommend adding zero address check for **baseCurrency**

INFORMATIONAL-40	IStakingHelper contains function from Curve and GLP contracts	Fixed at c9bcca

Description

IStakingHelper contains functions from Curve and GLP staking contracts. Although the functions are used separately for **CurveLPStakingCompartment GLPStakingCompartment**, using common interface may lead to bug in the future.

Recommendation

We recommend splitting **IStakingHelper** to **ICurveStakingHelper** and **IGLPStakingHelper** interfaces.

INFORMATIONAL-41	outOfGas for tokensCreated	Fixed at f15a58

Description

IERC20Wrapper and **IERC721Wrapper** create wrapper token contracts and save every address to **address[] internal _tokensCreated;** L52, L23. However, At the current moment, the block limit is **30M** gas, so we can create around **11K** addresses before the block limit. Moreover, **AddressRegistry** allow installing only 1 wrapper contract for one type of token.

```
interface IERC20Wrapper / IERC721Wrapper {
    function tokensCreated() external view returns (address[] memory);
    // outOfGas if wrapper more 11K created addresses
}
```

Recommendation

We recommended adding an additional method for getting by index or pagination.

INFORMATIONAL-42 | getReclaimableBalance has redundant arguments | Fixed at 4f1fb5

Description

IBaseCompartment getReclaimableBalance has 3 arguments, but all implementations used only **collTokenAddr** parameter L34-L40, L114-L125, L51-L57, L79-L85.

```
function getReclaimableBalance(
    uint256 initCollAmount, // <-- unused for all implementation
    uint256 amountReclaimedSoFar, // <-- unused for all implementation
    address collTokenAddr
) external view returns (uint256 reclaimableBalance);
```

Recommendation

We recommended removing **initCollAmount**, **amountReclaimedSoFar**. So, another implementation can get **DataTypesPeerToPeer.Loan** from **vaultAddr** by **loanIdx**

INFORMATIONAL-43 | Redundant calculations for reclaimCollAmount | Fixed at 78d9fd

Description

reclaimCollAmount calculates the following way L187-L195.

```
if (leftRepaymentAmount == loanRepayInstructions.targetRepayAmount) {
    reclaimCollAmount = SafeCast.toUint128(maxReclaimableCollAmount);
} else {
    reclaimCollAmount = SafeCast.toUint128(
        (maxReclaimableCollAmount *
            uint256(loanRepayInstructions.targetRepayAmount)) /
            uint256(leftRepaymentAmount)
    );
}
```

So, if **(leftRepaymentAmount == loanRepayInstructions.targetRepayAmount)** than **(loanRepayInstructions.targetRepayAmount / leftRepaymentAmount) == 1**.

Recommendation

We recommend simplifying calculation

```
reclaimCollAmount = SafeCast.toUint128(
    (maxReclaimableCollAmount *
        uint256(loanRepayInstructions.targetRepayAmount)) /
        uint256(leftRepaymentAmount)
);
```

INFORMATIONAL-44	UpfrontFee from non-collateral tokens	Fixed at 59b1de

Description

The collateral token amount, that the borrower sends to the contract is the value of **BorrowTransferInstructions::collSendAmount**. We can represent **collSendAmount = initCollAmount + upfrontFee + expectedTransferFee**, where:

1. **initCollAmount** is the value that the borrower receives on full repayment
2. **upfrontFee** value is taken from the borrower but counted as collateral to loan token value
3. **expectedTransferFee = protocolFeeAmount + tokenFee**, where **tokenFee** is the native token fee. **expectedTransferFee** does not count in loan token collateral value.

```
loanAmount = (loanPerCollUnit * (collSendAmount - expectedTransferFee)) /
    (10 ** IERC20Metadata(generalQuoteInfo.collToken).decimals());
```

At the same time, **upfrontFee** is calculated as a fee on the full value of the **collSendAmount**, where **expectedTransferFee** is not meant to be the collateral value.

```
upfrontFee = (borrowInstructions.collSendAmount * quoteTuple.upfrontFeePctInBase) /
    Constants.BASE;
```

Let **trueColl = upfrontFee + initCollAmount**, then
loanAmount = (loanPerCollUnit * trueColl) / (10 ** IERC20Metadata(collToken).decimals())

```
initCollAmount = collSendAmount - upfrontFee - expectedTransferFee
initCollAmount = collSendAmount - (collSendAmount * upfrontFeePctInBase) - expectedTransferFee
initCollAmount = collSendAmount * (1 - upfrontFeePctInBase) - expectedTransferFee
initCollAmount = (trueColl + expectedTransferFee) * (1 - upfrontFeePctInBase) - expectedTransferFee
initCollAmount = trueColl * (1 - upfrontFeePctInBase) - expectedTransferFee * upfrontFeePctInBase
```

This means the borrower will receive **expectedTransferFee * upfrontFeePctInBase** less collateral token at the end of the loan. Effectively paying extra protocol fee and token transfer fee (borrower pays for the vault's transfers) but to the vault owner.

Same could be applied to the **protocolFeeAmount**, ideally protocol fee should'nt depend on token transfer fee.

Recommendation

We recommend calculating the **upfrontFee** without including **expectedTransferFee** value.

INFORMATIONAL-45	Redundant safeCast	Fixed at 7929f8

Description

In the **getAbsoluteLoanTerms** function **_finalLoanAmount** value is calculated as

```
uint256 _finalLoanAmount = SafeCast.toUint128(
            totalSubscriptions - _arrangerFee
        );
```

And used after in calculation of **uint128 loanTokenDue**.

```
_tmpLoanTerms.repaymentSchedule[i].loanTokenDue = SafeCast
    .toUint128(
        (_finalLoanAmount *
            _tmpLoanTerms.repaymentSchedule[i].loanTokenDue) /
            Constants.BASE
    );
```

The function **getAbsoluteLoanTerms** returns **_finalLoanAmount** as **uint256**, where **finalizeLoanTermsAndTransferColl** saves that value into the **uint256 finalLoanAmount** field in the **DynamicLoanProposalData** structure.

Recommendation

We recommend removing **SafeCast.toUint128** for the **_finalLoanAmount** value.

INFORMATIONAL-46	Wrapped ERC redeem with allowance	Fixed at d3d85f

Description

A user with allowance to the **WrappedERC20** or the **WrappedERC721** would have to transfer wrapped tokens to himself before redeeming underlying tokens.

Recommendation

We recommend adding redeeming functionality to users with allowance.

```
function redeem(uint256 amount, address user) external nonReentrant {
    if (amount == 0 || balanceOf(user) < amount) {
        revert Errors.InvalidSendAmount();
    }
    if (user != msg.sender)
        _spendAllowance(user, msg.sender, amount)
//        ^-- checks if amount is sufficient (with max allowance)
```

INFORMATIONAL-47	Unchecked maxLoan off-chain quote parameter	Fixed at 898fef

Description

Due to the off-chain nature, the quote parameters cannot be checked beforehand. There are necessary checks (explicit and implicit) along the borrow execution flow, except for the **GeneralQuoteInfo::maxLoan** zero value. Hence, the off-chain quote signers can create a quote with values **maxLoan = 0** and **minLoan = 0**, which in some cases can lead to issue **H-2**.

Recommendation

We recommend implementing a fix from issue **H-2**, which adds an implicit check for **maxLoan != 0**.

```
if (
    loanAmount < borrowInstructions.minLoanAmount ||
    loanAmount == 0
) {
    revert Errors.TooSmallLoanAmount();
}
```

INFORMATIONAL-48	Wrapped redeem optimization	Fixed at d3d85f

Description

The **WrappedERC20Impl::redeem** and **WrappedERC721Impl::redeem** function executes in the order:

1. Check
2. Transfers
3. Burn

Although this function has a non-reentrant modifier. It is beneficial to follow the Checks-Effects-Interaction pattern to reduce gas consumption by moving **_burn** to the beginning of the **redeem** and lifting checks that are present in the internal **_burn** function. (-1 SLOAD in balance map)

Recommendation

We recommend moving **_burn** function to the beginning of the function.

For **WrappedERC20Impl**:

```
if (amount == 0 ) {
    revert Errors.InvalidSendAmount();
}
uint256 currTotalSupply = totalSupply(); //<- save `totalSupply`

_burn(msg.sender, amount); // <- has `accountBalance >= amount` check
...
```

For **WrappedERC721Impl**:

```
_burn(msg.sender, 1);
...
```

INFORMATIONAL-49	Inconsistent instructions structures	Fixed at 173b8a

Description

While borrowing, the user will input data via the **BorrowTransferInstructions** structure which includes fields **callbackAddr** and **callbackData**. At the same, the repay structure **LoanRepayInstructions** is missing these callback fields.

```
function borrowWithOnChainQuote(
    address lenderVault,
    DataTypesPeerToPeer.BorrowTransferInstructions
        calldata borrowInstructions,
    DataTypesPeerToPeer.OnChainQuote calldata onChainQuote,
    uint256 quoteTupleIdx
)
```

```
function repay(
    DataTypesPeerToPeer.LoanRepayInstructions
        calldata loanRepayInstructions,
    address vaultAddr,
    address callbackAddr,
    bytes calldata callbackData
)
```

Recommendation

We recommend moving the **callbackAddr** and **callbackData** fields to the structure.

INFORMATIONAL-50	SLOAD reduction via existing memory	Fixed at 704c5d

Description

Gas consumption could be reduced by replacing storage variables with existing memory copies.

1. In **LenderVaultImpl::unlockCollateral** storage **_loan.collToken** to **collToken**
2. In **LenderVaultImpl::processQuote** move **loanId** before the **if** expression and pass it to the **_createCollCompartment** function instead of **_loans.length**
3. In **CurveLPStakingCompartment::stake** storage **liqGaugeAddr** to **_liqGaugeAddr**
4. In **BaseCompartment::_unlockCollToVault** make a **vaultAddr** memory copy.

Recommendation

We recommend using memory copies of storage variables.

INFORMATIONAL-51	Oracle whitelist check	Fixed at 49ee34

Description

Currently, the oracle whitelist state is checked in the **LenderVaultImpl::_getLoanAndRepayAmount** which is a part of the **LenderVaultImpl::processQuote** function. Although **oracleAddr** is the part of **GeneralQuoteInfo** structure and other whitelist checks are executed in the **QuoteHandler::_checkSenderAndGeneralQuoteInfo**.

Recommendation

We recommend moving oracle whitelist verification to the **_checkSenderAndGeneralQuoteInfo** function.

INFORMATIONAL-52 | Unnecessary logic separation | Fixed at 7992e7

Description

The **repay** function gets loan information via a **loan(targetId)** call to the vault. After, with **loan** information, it calls the vault's view function **validateRepayInfo**, which gets all needed variables from calldata. Furthermore, it makes sense to have repay information logic in the borrower's entry point.

Recommendation

We recommend moving the **validateRepayInfo** function as public to the **BorrowerGateway** to avoid redundant external calls.

INFORMATIONAL-53 | Verification code duplication | Fixed at 49ee34

Description

The verification of the **lenderVault** and its owner could be implemented via an internal function:

```
if (
    !IAddressRegistry(_addressRegistry).isRegisteredVault(lenderVault)
) {
    revert Errors.UnregisteredVault();
}
if (ILenderVaultImpl(lenderVault).owner() != msg.sender) {
    revert Errors.InvalidSender();
}
```

The same code is present in the following functions:

1. **addOnChainQuote**
2. **updateOnChainQuote**
3. **deleteOnChainQuote**
4. **incrementOffChainQuoteNonce**
5. **invalidateOffChainQuote**

Recommendation

We recommend making the new internal function for the vault owner verification.

INFORMATIONAL-54 | Address to uint160 cast | Fixed at f73d21

Description

Casting to **int** is redundant as **address** datatype can be used in **less-than** or **greater-than** comparisons.

1. **QuoteHandler::_areValidSignatures**
2. **ERC20Wrapper::createWrappedToken**
3. **ERC721Wrapper::createWrappedToken**

Recommendation

We recommend removing redundant **address**->**int** castings.

INFORMATIONAL-55	Redundant SLOAD CurveLPStakingCompartment	Fixed at 725a3c

Description

The internal **_withdrawCollFromGauge** function reads **liqGaugeAddr** from storage, although **liqGaugeAddr** is loaded into the memory before **_withdrawCollFromGauge** invocation.

Recommendation

We recommend passing memory **liqGaugeAddr** to the internal function instead of reading it twice.

INFORMATIONAL-56	Unindexed Event topics	Fixed at 227a2c

Description

QuoteHandler events are missing indexed fields.

IQuoteHandler::OnChainQuoteAdded IQuoteHandler::OnChainQuoteDeleted IQuoteHandler::OnChainQuoteInvalidated IQuoteHandler::OffChainQuoteNonceIncremented IQuoteHandler::OffChainQuoteInvalidated IQuoteHandler::OnChainQuoteUsed IQuoteHandler::OffChainQuoteUsed

It would be beneficial to the protocol to mark fields like **lenderVault** and **ChainQuoteHash** as indexed for easy on-chain event filtering.

The same could be applied to events:

ILenderVaultImpl::QuoteProcessed with **borrower** & **loan** fields.

IBorrowerGateway::Borrowed with **loan** field.

IAddressRegistry::AllowedTokensForCompartmentUpdated with **compartmentImpl** field and

IAddressRegistry::BorrowerWhitelistUpdated with **whitelistAuthority** field.

Recommendation

We recommend marking these event fields as indexed.

INFORMATIONAL-57	Immutable to memory	Fixed at 704c5d

Description

Storing immutables variable in memory consumes more gas, than using just immutable variables.

For immutable **addressRegistry** variable:

1. **_checkSenderAndGeneralQuoteInfo**
2. **addOnChainQuote**
3. **updateOnChainQuote**

Recommendation

We recommend using immutable variables without storing them in memory.

INFORMATIONAL-58	Storage to memory gas reduction	Fixed at d9bc59

Description

There are several instances storage variables are read multiple times in the same execution context.

1. In **deposit** storage variable **depositToken**.
2. In **subscribe** storage variable **factory**.
3. In **executeLoanProposal** storage variables **factory** and **depositToken**.
4. In **acceptLoanTerms** storage variable **lastLoanTermsUpdateTime**
5. In **canSubscribe** and **canUnsubscribe** storage variable **dynamicData.status**.
6. In **finalizeLoanTermsAndTransferColl** storage variable **_loanTerms.repaymentSchedule.length** used instead of existing memory variable **_finalizedLoanTerms.repaymentSchedule.length**

Recommendation

We recommend saving storage variables to the memory if it is used in multiple instances.

INFORMATIONAL-59	Redundant MSTORE	Fixed at d9bc59

Description

In the **repay** function the memory variable **_repayment.collTokenDueIfConverted** is saved into the new memory variable **collTokenDueIfAllConverted**. Removing an extra copy of the memory variable will reduce the gas consumption of the function.

Recommendation

We recommend removing redundant copy of memory variable.

INFORMATIONAL-60	Redundant quoteTuples check	Fixed at e3b77b

Description

The **_isValidOnChainQuote** checks if items in **quoteTuples** are valid and/or define swap quote.
The second check guarantees that if there is **swap** quote, then it is the only one in the list.

```
if (isSwapCurr && onChainQuote.quoteTuples.length > 1) {
    return false;
}
```

But the third checks if all items in the list are the same type (**swap** or default).

```
if (k > 0 && isSwap != isSwapCurr) {
    return false;
}
```

Recommendation

We recommend removing the third check in the for loop.

INFORMATIONAL-61	Redundant reentrancy guard	Fixed at 4689e7

Description

The **redeem** and **sweepTokensLeftAfterRedeem** functions are guarded with the **nonReentrant** modifier. At the same time, the custom **_mutex** guard is introduced.

Recommendation

We recommend removing the **_mutex** guard.

INFORMATIONAL-62	Misleading comment for Ownable2Step inheritance	Fixed at 864469

Description

The contracts with **Ownable2Step** inheritance override the **transferOwnership** function with additional checks. Although it says that the input value will be checked against **address(0)** in the parent contract that is not true. In this particular case, the **super** keyword will only call **Ownable2Step::transferOwnership** (no zero check), not **Ownable::transferOwnership** (where zero check exists).

1. AddressRegistry
2. LenderVaultImpl
3. Factory

Recommendation

We recommend removing the comment regarding zero-check.

INFORMATIONAL-63	Gas optimization via unchecked	Fixed at 386e3f

Description

The **subscribe** function can be slightly optimized with the balance update.

```
balanceOf[msg.sender] = _balanceOf - effectiveSubscriptionAmount;
```

Where **effectiveSubscriptionAmount = min(maxSubscriptionAmount, _freeSubscriptionSpace)** and **maxSubscriptionAmount <= _balanceOf**.

Recommendation

We recommend updating the balance without underflow checks.

INFORMATIONAL-64	Redundant isLoan field in TransferInstructions struct	Fixed at f2d51e

Description

Field **isLoan** is only used in function processQuote() and is needed for emitting events. In **BorrowerGateway** there is no logic based on this variable.

Recommendation

It is recommended to remove **isLoan** field from **TransferInstructions** struct and calculate **isLoan** locally in **LenderVaultImpl**.

INFORMATIONAL-65

Redundant loan calculations for swap quote

Fixed at fa1db9

Description

In function _getLoanAndRepayAmount() there are calculations of **interestRateFactor**, **repayAmount** variables which is redundant when it is **swap** quote.

Recommendation

It is recommended to check for **swap** quote to reduce calculations.

INFORMATIONAL-66

Adjust subscription amount instead of revert

Fixed at 2d67a5

Description

At Line 119 subscription amount is checked, if it exceeds the **loanTerms.maxTotalSubscriptions**. Instead of revert, this amount can be recalculated not to go over the limit.

Recommendation

It is recommended to recalculate necessary amount to meet the limit.

INFORMATIONAL-67

Checking protocol fee for zero

Fixed at fe8c3a

Description

At Line 199 **protocolFeeShare** is calculated based on **arrangerFeeSplit**. By protocol design **arrangerFeeSplit** could be zero based on conditions at Lines 119 - 124. In this case **protocolFeeShare** also will zero, and transfer with zero amount may occur.

Recommendation

It is recommended to check **protocolFeeShare** or **arrangerFeeSplit** for zero value before any transfers.

INFORMATIONAL-68

Optimizing protocol logic when repayment is fully converted

Fixed at 335962

Description

During conversion phase of repayment whole loan can be converted. In this case it is not optimal for **borrower** to call **repay** function, because it only updates **currentRepaymentIdx**, and if it is last repayment, transfers collateral for default. Additional condition can be provided in function exerciseConversion() to check if loan is fully converted.

Recommendation

It is recommended to add check if loan is fully converted. If it happened, then **currentRepaymentIdx** should be incremented and if it is last repayment, then collateral tokens for default should be returned to **borrower**.

INFORMATIONAL-70	Missing state check	Fixed at d9bc59

Description

In function claimRepayment() there is no check for current state.

Recommendation

It is recommended to add state check (**LOAN_DEPLOYED** or **DEFAULTED**) for additional safety.

INFORMATIONAL-71	Unchanged value of _lenderExercisedConversion	Fixed at df0123

Description

Inside if - statement at Lines 498 - 506 conversion is handled, but at the end of the function **_lenderExercisedConversion** is not set to **true**.

Recommendation

It is recommended to change value of **_lenderExercisedConversion** to have precise state.

INFORMATIONAL-72	Restricting length of repayments	Fixed at 87f47e

Description

In function _repaymentScheduleCheck() **repaymentSchedule** array is checked for emptiness, but its length is not limited.

Recommendation

It is recommended to add limit for number of repayments.

INFORMATIONAL-73	Using calldata variable instead of storage one	Fixed at 704c5d

Description

At Line 76 **loan.collToken** is used. Before this statement, **loan.collToken** is checked if it equals **collToken** in function params. It is better use **calldata** variable to reduce gas costs.

Recommendation

It is recommended to use variable from function params instead of **storage** one.

INFORMATIONAL-74	Redundant approve	Fixed at 9af2fe

Description

In function borrowCallback() there are 3 approves. But first approve is redundant, because after swap there is a approve which sets to zero allowance.
Same issue:

- Function repayCallback()
- Function borrowCallback()
- Function repayCallback()

Recommendation

It is recommended to remove first approve because after swap and last approve allowance will already zero.

INFORMATIONAL-75	Set allowance to zero	Fixed at 154d9b

Description

In function stake() after the deposit at Line 55 allowance is not set to zero.

Recommendation

It is recommended to set to zero the allowance after a deposit the same way as in callbacks after swap.

INFORMATIONAL-76	Double whitelist state changing	Fixed at 57a424

Description

Function _resetAddrState() is used before changing address state. Firstly, it checks if address is wrapper or token manager, then it resets state. After resetting state in functions _updateSingletonState() and setWhitelistState() it is set again.

Recommendation

It is recommended to make separate checks for wrappers and token manager and make only one whitelist state setting to avoid duplication.

INFORMATIONAL-77	Code duplication while borrowing	Fixed at 6d0060

Description

Functions borrowWithOffChainQuote() and borrowWithOnChainQuote() have two same inner calls: ILenderVaultImpl(lenderVault).processQuote() and _processTransfers().

Recommendation

It is recommended to make one internal function with these two inner calls.

INFORMATIONAL-78	Transfer fee on protocol fee	Fixed at 59b1de

Description

Protocol supports token with fee on transfers, but at Line 274 **protocolFeeAmount** is transferred and there is no check for expected fee on transfer.

Recommendation

It is recommended to leave a comment if it is intended behavior or to include transfer fees for **protocolFeeAmount** in **expectedTransferFee**.

INFORMATIONAL-79	Making checks before assigning	Fixed at fa1db9

Description

At Lines 169 - 172 **_loan.expiry** and **_loan.earliestRepay** are set, but then they are checked. If they don't satisfy the condition, execution reverts. To optimize gas cost in case of error it is better to make checks before assigning values.
Same issue:

- Lines 52 - **onChainQuoteHash** is checked if it already exists, but before it general quote info is checked.

Recommendation

It is recommended to make general checks before assigning values to data structures or making more precise checks.

INFORMATIONAL-80	QuoteTuple index is not checked	Fixed at f15cd1

Description

In function checkAndRegisterOnChainQuote() **quoteTupleIdx** is not checked if it exceeds length of **OnChainQuote.quoteTuples** array.

Recommendation

It is recommended to provide additional check for **quoteTupleIdx**.

INFORMATIONAL-81	Parameter minLoan can be zero	Fixed at 6d0060

Description

In the contract **QuoteHandler** in the function **_isValidOnChainQuote** there are checks that **minLoan <= maxLoan && maxLoan != 0**. A case when **0 == minLoan < maxLoan**, will pass this check but this doesn't make any sense to make a possibility to take zero loan (**minLoan == 0**).

Recommendation

Add additional check that **minLoan != 0** too:

```
if (
    onChainQuote.generalQuoteInfo.maxLoan == 0 ||
    onChainQuote.generalQuoteInfo.minLoan == 0 ||  <---- new condition
    onChainQuote.generalQuoteInfo.minLoan >
    onChainQuote.generalQuoteInfo.maxLoan
) {
    return false;
}
```

INFORMATIONAL-82	Using ERC-2098 standard for signatures	Fixed at f73d21

Description

There is a method that can reduce a signature representation from form (v, r, s) that takes place 65 bytes to form (r, vs) that takes place 64 bytes. By reducing the size of a signature, gas consumption for some functions will reduce. More information about this method can be find here.

Recommendation

Using ERC-2098 standard for signatures will reduce gas consumption.

INFORMATIONAL-83	Misleading name of a function	Fixed at 704c5d

Description

In the contract **Ownable** there is a function **_initialize**. Everything would be fine if it is used only inside this contract, but inside derived contracts this function with this name can mislead the reader about it's purpose.

Recommendation

Maybe it is better to rename this function to the **__Ownable_init**, for example.

INFORMATIONAL-84	No function that returns the length of array in a contract	Fixed at 954301

Description

In the contracts **LenderVaultImpl**, **ERC20Wrapper**, **ERC721Wrapper**, and **Factory** there are no functions that return the length of global variables of type **address[]**:

- In the contract **LenderVaultImpl** the global variable **signers**
- In the contract **ERC20Wrapper** the global variable **tokensCreated**
- In the contract **ERC721Wrapper** the global variable **tokensCreated**
- In the contract **Factory** the global variables **loanProposals** and **fundingPools**

Recommendation

Add corresponding functions that returns length of arrays.

INFORMATIONAL-85	OpenZeppelin's ECDSA library can create Ethereum Signed Message	Fixed at 9eaff6

Description

In the contract **Factory** in the function **claimLenderWhitelistStatus** there is a **Ethereum Signed Message** creation from **payloadHash** variable. OpenZeppelin's **ECDSA** library can do this using function **toEthSignedMessageHash**.

Recommendation

It is recommended to use existing solutions instead of doing all manually.

INFORMATIONAL-86	Gas optimization	Partially fixed at https://github.com/mysofinance/v2/commit/ac3f28fca0637a46d57a733

Description

Here are some improvements that can be done, which will reduce gas consumption:

- Inside **AddressRegistry** contract
  - No need in caching these global variables. Without caching they will be read only once.
  - No need in updating the storage here. Every time after this write, the new value **DataTypesPeerToPeer.WhitelistState.NOT_WHITELISTED** in this storage slot is always overwritten by another value.
- Inside **WrappedERC721Impl** contract
  - Global variables are read multiple times in cycles here and here
- Inside **WrappedERC20Impl** contract
  - Global variable is read multiple times in cycle here
- Inside **FundingPoolImpl** contract
  - In the function **deposit** the global variable **depositToken** is read multiple times
  - In the function **subscribe** the global variable **factory** is read multiple times
  - In the function **unsubscribe** the global variable **subscriptionAmountPerLender[msg.sender]** is read multiple times
  - In the function **executeLoanProposal** the global variables **factory** and **depositToken** are read multiple times
  - In the function **initialize** the argument **_factory** is redundant. It's value can be replaced with **msg.sender**
- Inside **LoanProposalImpl** contract
  - In the function **initialize** the argument **_factory** is redundant. It's value can be replaced with **msg.sender**
  - In the function **acceptLoanTerms** the global variable **lastLoanTermsUpdateTime** is read multiple times
  - In the function **finalizeLoanTermsAndTransferColl** the variable **_loanTerms.repaymentSchedule.length** in the cycle can be replaced with **_unfinalizedLoanTerms.repaymentSchedule.length**. Also, the global variable **staticData.collToken** is read multiple times
  - In the function **canSubscribe** the global variable **dynamicData.status** can be read multiple times

Recommendation

It is better to reduce number of contacts with storage because it is expensive.

INFORMATIONAL-87	Use clone instead of cloneDeterministic to save gas	Fixed at 39b8f3

Description

At Line 55 and Line 101 OpenZeppelin's cloneDeterministic is used to create LoanProposal and FundingPool contracts accordingly.
There is no need to deploy contract to a deterministic address, as predictDeterministicAddress is never used. Even if it was used, determining final addresses would be complicated by use of array lengths in the salt parameter numLoanProposals and fundingPools.length.

Recommendation

We recommend using **clone** instead.

INFORMATIONAL-88	Infinite unsubscribe period	Fixed at 022cdd

Description

In **LoanProposalImpl.intialize()** there's a check, that **_unsubscribeGracePeriod** is bigger than 1 day, due to value in **Constants**. But there's no upper bounds. This condition disallow to call **rollback()**, which is a little bit strange.

Recommendation

We recommend adding upper bound for **_unsubscribeGracePeriod**.

INFORMATIONAL-89	The sufficient condition is always satisfied	Fixed at 43355d

Description

In **LoanProposalImpl.finalizeLoanTermsAndTransferColl()** there's a condition, that if **status != borrower_accepted || block.timestamp < lenderInCutOffTime()**, but there's no need to check second condition, as it shouldn't be true without setting **dynamicData.loanTermsLockedTime**, which is set together with **BORROWER_ACCEPTED** status in **acceptLoanTerms()** function.

Recommendation

We recommend to add some time period in second condition and replace **||** to **&&**, or remove second condition at all, cause of it uselessness.

INFORMATIONAL-90	Positive and Negative MysoTokenManager hooks	Fixed at d9bc59

Description

As you've mentioned before, you want to add future tokenomics providing hooks for **MysoTokenManager**. But in current version of contracts, there's only "positive" hooks (for example **subscribe** and **deposit** functions). But there's no similar hooks on "negative" functions, which will lead to abuse system with scenario like:

1. **Deposit** tokens, call hook, for example minting some tokens or points.
2. **Withdraw** tokens, without calling corresponding hook.
3. Looping for 1-2 steps.

Recommendation

We recommend taking into account hooks only from irreversible functions (for example **LoanProposalImpl.finalizeLoanTermsAndTransferColl()**). Adding additional hooks for **withdraw** or **unsubscribe** functions can be bad idea, for example in case something bad will happen with **MysoTokenManager** users should always have ability to get back their funds, these hooks can revert transactions.

INFORMATIONAL-91	No protection from malicious proposal	Fixed at d9bc59

Description

In the function **FundingPoolImpl.executeLoanProposal** the amounts **finalLoanAmount** and **arrangerFee** are taken from loan proposal contracts.

```
function executeLoanProposal(address loanProposal) external {
        if (!IFactory(factory).isLoanProposal(loanProposal)) {
            revert Errors.UnregisteredLoanProposal();
        }

        (
            uint256 arrangerFee,
            uint256 finalLoanAmount, // <-- amount from loanProposal
            ,
            ,
            ,
            ,

        ) = ILoanProposalImpl(loanProposal).dynamicData();
        DataTypesPeerToPool.LoanTerms memory loanTerms = ILoanProposalImpl(
            loanProposal
        ).loanTerms();
        ILoanProposalImpl(loanProposal).checkAndUpdateStatus();
        IERC20Metadata(depositToken).safeTransfer(
            loanTerms.borrower,
            finalLoanAmount // <-- no check that the amount does not exceed the number of subscriptions
        );
        // ...
    }
```

The amounts **finalLoanAmount** and **arrangerFee** must satisfy equality **finalLoanAmount + arrangerFee = subscriptionAmountOf[loanProposal]**, but there is no such check. In case of malicious loan proposal implementation, it is able to provide **finalLoanAmount** equal to funding pool's balance and withdraw all tokens.

Recommendation

We recommend adding sanity check on **arrangerFee** and **finalLoanAmount**:

```
if (arrangerFee + finalLoanAmount != subscriptionAmountOf[loanProposal]) {
    revert Errors.IncorrectLoanAmount();
}
```

INFORMATIONAL-92 | No option to set the exact number of subscriptions | Fixed at ca2e1b

Description

In the **LoanProposalImpl.proposeLoanTerms** there is a check

```
if (
    newLoanTerms.minTotalSubscriptions == 0 ||
    newLoanTerms.minTotalSubscriptions >=
    newLoanTerms.maxTotalSubscriptions
) {
    revert Errors.InvalidSubscriptionRange();
}
```

Thus, there is no way to set **minTotalSubscriptions = maxTotalSubscriptions**. This condition seems to be redundant.

Recommendation

We recommend removing the redundant condition or leave a comment if it is intended behavior for preventing DOS attacks with small amounts.

INFORMATIONAL-93 | Unnecessary expressions/variables v2 | P2Peer | Acknowledged

Description

There are several unnecessary expressions or variables in your codebase. They include:

- VoteCompartment.sol#L26 - **if (_delegatee == address(0)) {**

Recommendation

It is recommended to remove these unnecessary expressions or variables.

INFORMATIONAL-94 | Blacklisted tokens will revert the whole transaction | Acknowledged

Description

In the contract **WrappedERC20Impl**, the function **redeem()** will revert the whole transaction if some of the redeemable tokens are blacklisted.

Recommendation

It is recommended to consider this case and use a **try-catch** structure to handle potential reverts caused by blacklisted tokens during the redemption process.

INFORMATIONAL-95 | Lack of zero check | Acknowledged

Description

There is no check that **_arranger** is not equal to address(0) while there is check for **_factory** even if it called as **msg.sender** in **createLoanProposal**.

Recommendation

We recommend adding zero check for _arranger.

INFORMATIONAL-96	Adding a Sweep Function to the redeem Function of Wrapped ERC20	Acknowledged

Description

There is a possibility of tokens being unintentionally sent to this "vault" due to user error or other reasons.

Recommendation

We recommend adding a **sweep()** function to the WrapERC20 implementation, similar to the one found in UniswapV2. This function will allow the contract to safely recover and transfer any mistakenly sent tokens out of the vault, ensuring that they are returned to their rightful owners or handled appropriately.

INFORMATIONAL-97	loans can be defaulted if sequencer down	Acknowledged

Description

L2 chains such as Arbitrum, Optimism, Metis have single centralize transaction sequencer. The Responsibility of sequencer is

- providing transaction confirmations and state updates
- constructing and executing L2 blocks
- submitting user transactions to L1

If sequencer down users can't send transactions via L2, as workaround users can send transactions direct to L1, but this is only available for technical specialists. Also, when sequencer down, we have timeline gaps between **sequencer down timestamp** and **sequencer up timestamp** for L2 chain. So, borrowers can't repay if sequencer will be down and loan can be expiry. Chainlink provides **UptimeSequencerFeed** contracts for L2. For example, AAVE extends the liquidation grace period positions. Details in AAVE whitepaper page 14 Defaulted loans with a high loan amount will have high impact on protocol and borrower.
You can find articles that confirm that the sequencer crashes not so rarely. For example:
https://www.thecoinrepublic.com/2023/06/09/ethereum-l2-network-arbitrum-shuts-down-for-hours-due-to-a-bug/
https://thedefiant.io/arbitrum-outage-2

Recommendation

We recommend:

- adding **UptimeSequencerFeed** contracts for L2. In case, if sequencer down **BorrowerGateway**, **LenderVaultImpl** contracts should shift expiry time or repayment period.
- adding information about additional risks in L2 chains for users if you can't to support **UptimeSequencerFeed**

INFORMATIONAL-98	Redundant re-entrance check	Acknowledged

Description

The **withdraw** function has its reentrancy guard flag **withdrawEntered** which restricts entering any of the compartment functions. It is possible to hijack the flow, but hijacking won't give any advantages to an attacker due to the absence of state changes in the **withdraw** function. The only state variable present in **withdraw** is **lockedAmounts** which is tied to an input **token**.

Recommendation

We recommend removing redundant flag and checks.

INFORMATIONAL-99	Calldata reduction	Acknowledged

Description

The **updateOnChainQuote** function asks for the full structure of **oldOnChainQuote** only to calculate its hash afterward.

Same issue:

- Function deleteOnChainQuote()

Recommendation

We recommend changing input data from the structure to its hash to save gas.

INFORMATIONAL-100	Pragma of interfaces	Acknowledged

Description

In all interfaces, there are **0.8.19** or **^0.8.19** versions of solidity in pragmas. The problem is that if someone downloads this repository as an npm package and uses a different version of solidity (even if his version is **>=0.9.0**), he won't be able to use these interfaces in his code. He must copy all required interfaces in his repository and change all pragmas in interfaces.

Recommendation

All can be fixed if use **>=0.8.0** type of pragma in all interfaces.

For example, Uniswap uses this approach: code and interface. Code has **=0.7.6** pragma, and interface has **>=0.5.0** pragma. In this case, every project with a version of solidity at least **0.5.0** can import Uniswap's interfaces and use them without modification.

INFORMATIONAL-101	Using ReentrancyGuardUpgradeable contract	Acknowledged

Description

In the contracts **WrappedERC721Impl** and **WrappedERC20Impl** the contract **ReentrancyGuard** is used. It is better to use **Upgradeable** versions of contracts when the contract is initializable. The same in the contract **FundingPoolImpl**.

Recommendation

It is better to replace **ReentrancyGuard** contract with **ReentrancyGuardUpgradeable**.

INFORMATIONAL-102	EnumerableSet.Address instead of mappings and arrays	Acknowledged

Description

In Factory some storage variables are set as a pair array and mapping, in order to make it comfortable. But this way is not practical, you can use **OpenZeppelin.EnumerableSet** instead and write some public view functions, in order to make code more clear and unite pair variables to avoid some possible problems with inconsistency.

Recommendation

We recommend replacing pair variables with **EnumerableSet** struct.

INFORMATIONAL-103	Redundant checks	Acknowledged

Description

In the function **QuoteHandler._isValidOnChainQuoteTuple** there is a check, which prohibits **quoteTuple.upfrontFeePctInBase == Constants.BASE** when other requirements for swap are not met.

```
if (quoteTuple.upfrontFeePctInBase == Constants.BASE) {
    // note: if upfrontFee=100% this corresponds to an outright swap; check other fields are consistent
    if (!isSwap) {
        return (false, isSwap);
    }
}
```

However, there are excessive conditions in **LenderVaultImpl.processQuote**:

```
if (quoteTuple.upfrontFeePctInBase == Constants.BASE) {
    // note: if upfrontFee=100% this corresponds to an outright swap; check that tenor is zero
    if (
        _loan.initCollAmount != 0 ||
        quoteTuple.tenor + generalQuoteInfo.earliestRepayTenor != 0 ||
        generalQuoteInfo.borrowerCompartmentImplementation != address(0)
    ) { // <-- redundant
        revert Errors.InvalidSwap();
    }
}
```

and in **QuoteHandler._checkTokensAndCompartmentWhitelist**:

```
if (isSwap) {
    if (compartmentImpl != address(0)) { // <-- redundant
        revert Errors.InvalidSwap();
    }
    return;
}
```

Recommendation

We recommend removing redundant checks.

7. Appendix B. Slither

High/High/arbitrary-send-erc20

BorrowerGateway._processTransfers(address,address,DataTypesPeerToPeer.BorrowTransferInstructions,DataTypesPeerToPeer.Loan,uint256) uses arbitrary from in transferFrom: IERC20Metadata(loan.collToken).safeTransferFrom(loan.borrower,collReceiver,collReceiverTransferAmount)

BorrowerGateway._processTransfers(address,address,DataTypesPeerToPeer.BorrowTransferInstructions,DataTypesPeerToPeer.Loan,uint256) uses arbitrary from in transferFrom: IERC20Metadata(loan.collToken).safeTransferFrom(loan.borrower,IAddressRegistry(addressRegistry).owner(),protocolFeeAmount)

BorrowerGateway._processTransfers(address,address,DataTypesPeerToPeer.BorrowTransferInstructions,DataTypesPeerToPeer.Loan,uint256) uses arbitrary from in transferFrom: IERC20Metadata(loan.collToken).safeTransferFrom(loan.borrower,lenderVault,upfrontFee)

BorrowerGateway._processRepayTransfers(address,DataTypesPeerToPeer.LoanRepayInstructions,DataTypesPeerToPeer.Loan,address,bytes) uses arbitrary from in transferFrom: IERC20Metadata(loan.loanToken).safeTransferFrom(loan.borrower,lenderVault,uint256(loanRepayInstructions.targetRepayAmount) + loanRepayInstructions.expectedTransferFee)

Informational/High/naming-convention

Parameter LoanProposalImpl.initialize(address,address,address,address,address,uint256,uint256,uint256,uint256)._whitelistAuthority is not in mixedCase

Parameter Ownable.proposeNewOwner(address)._newOwnerProposal is not in mixedCase

Parameter LoanProposalImpl.initialize(address,address,address,address,address,uint256,uint256,uint256,uint256)._conversionGracePeriod is not in mixedCase

Variable ChainlinkBasic.BASE_CURRENCY_UNIT is not in mixedCase

Parameter LenderVaultImpl.setMinNumOfSigners(uint256)._minNumOfSigners is not in mixedCase

Parameter Factory.createLoanProposal(address,address,address,uint256,uint256,uint256,uint256)._arrangerFee is not in mixedCase

Parameter Factory.createLoanProposal(address,address,address,uint256,uint256,uint256,uint256)._whitelistAuthority is not in mixedCase

Parameter LoanProposalImpl.initialize(address,address,address,address,address,uint256,uint256,uint256,uint256)._repaymentGracePeriod is not in mixedCase

Variable LoanProposalImpl._loanTerms is not in mixedCase

Parameter LoanProposalImpl.initialize(address,address,address,address,address,uint256,uint256,uint256,uint256)._unsubscribeGracePeriod is not in mixedCase

Parameter LoanProposalImpl.initialize(address,address,address,address,address,uint256,uint256,uint256,uint256)._arrangerFee is not in mixedCase

Parameter Factory.createLoanProposal(address,address,address,uint256,uint256,uint256,uint256)._fundingPool is not in mixedCase

Parameter LoanProposalImpl.initialize(address,address,address,address,address,uint256,uint256,uint256,uint256)._collToken is not in mixedCase

Variable ChainlinkBasic.BASE_CURRENCY is not in mixedCase

Parameter FundingPoolImpl.initialize(address,address)._factory is not in mixedCase

Variable LenderVaultImpl._loans is not in mixedCase

Parameter LenderVaultImpl.addSigners(address[])._signers is not in mixedCase

Parameter LoanProposalImpl.initialize(address,address,address,address,address,uint256,uint256,uint256,uint256)._fundingPool is not in mixedCase

Variable Factory._lenderWhitelistedUntil is not in mixedCase

Parameter LenderVaultImpl.initialize(address,address)._vaultOwner is not in mixedCase

Parameter LoanProposalImpl.initialize(address,address,address,address,address,uint256,uint256,uint256,uint256)._arranger is not in mixedCase

Function IStakingHelper.claim_rewards() is not in mixedCase

Variable FundingPoolImpl._earliestUnsubscribe is not in mixedCase

Parameter LenderVaultImpl.unlockCollateral(address,uint256[])._loanIds is not in mixedCase

Parameter AddressRegistry.initialize(address,address,address)._borrowerGateway is not in mixedCase

Parameter LoanProposalImpl.acceptLoanTerms(uint256)._loanTermsUpdateTime is not in mixedCase

Variable Ownable._owner is not in mixedCase

Variable Ownable._newOwner is not in mixedCase

Parameter LoanProposalImpl.getAbsoluteLoanTerms(DataTypesPeerToPool.LoanTerms,uint256,uint256)._tmpLoanTerms is not in mixedCase

Variable Factory._depositTokenHasFundingPool is not in mixedCase

Parameter AddressRegistry.initialize(address,address,address)._quoteHandler is not in mixedCase

Parameter FundingPoolImpl.initialize(address,address)._depositToken is not in mixedCase

Variable LoanProposalImpl._loanTokenRepaid is not in mixedCase

Parameter LenderVaultImpl.initialize(address,address)._addressRegistry is not in mixedCase

Variable LoanProposalImpl._totalSubscriptionsThatClaimedOnDefault is not in mixedCase

Variable AddressRegistry._isTokenWhitelistedForCompartment is not in mixedCase

Variable LoanProposalImpl._lenderExercisedConversion is not in mixedCase

Parameter BorrowerGateway.setProtocolFee(uint256)._newFee is not in mixedCase

Variable AddressRegistry._isInitialized is not in mixedCase

Parameter Factory.createLoanProposal(address,address,address,uint256,uint256,uint256,uint256)._unsubscribeGracePeriod is not in mixedCase

Parameter AddressRegistry.initialize(address,address,address)._lenderVaultFactory is not in mixedCase

Parameter Factory.createLoanProposal(address,address,address,uint256,uint256,uint256,uint256)._conversionGracePeriod is not in mixedCase

Parameter LenderVaultImpl.validateRepayInfo(address,DataTypesPeerToPeer.Loan,DataTypesPeerToPeer.LoanRepayInstructions)._loan is not in mixedCase

Parameter Factory.createLoanProposal(address,address,address,uint256,uint256,uint256,uint256)._collToken is not in mixedCase

Variable LoanProposalImpl._lenderClaimedRepayment is not in mixedCase

Parameter Factory.createFundingPool(address)._depositToken is not in mixedCase

Variable LoanProposalImpl._lenderClaimedCollateralOnDefault is not in mixedCase

Function IStakingHelper.reward_tokens(uint256) is not in mixedCase

Variable AddressRegistry._borrowerWhitelistedUntil is not in mixedCase

Variable AddressRegistry._registeredVaults is not in mixedCase

Function IStakingHelper.lp_token() is not in mixedCase

Parameter Factory.setArrangerFeeSplit(uint256)._newArrangerFeeSplit is not in mixedCase

Parameter Factory.createLoanProposal(address,address,address,uint256,uint256,uint256,uint256)._repaymentGracePeriod is not in mixedCase

Parameter LoanProposalImpl.initialize(address,address,address,address,address,uint256,uint256,uint256,uint256)._factory is not in mixedCase

Low/High/shadowing-local

ILenderVaultImpl.initialize(address,address).addressRegistry shadows: – ILenderVaultImpl.addressRegistry() (function)

IAddressRegistry.whitelistState(address).whitelistState shadows: – IAddressRegistry.whitelistState(address) (function)

ILenderVaultImpl.loan(uint256).loan shadows: – ILenderVaultImpl.loan(uint256) (function)

ILoanProposalImpl.lastLoanTermsUpdateTime().lastLoanTermsUpdateTime shadows: – ILoanProposalImpl.lastLoanTermsUpdateTime() (function)

ILoanProposalImpl.getAbsoluteLoanTerms(DataTypesPeerToPool.LoanTerms,uint256,uint256).loanTerms shadows: – ILoanProposalImpl.loanTerms() (function)

ILenderVaultImpl.validateRepayInfo(address,DataTypesPeerToPeer.Loan,DataTypesPeerToPeer.LoanRepayInstructions).loan shadows: – ILenderVaultImpl.loan(uint256) (function)

IAddressRegistry.setWhitelistState(address[],DataTypesPeerToPeer.WhitelistState).whitelistState shadows: – IAddressRegistry.whitelistState(address) (function)

ILenderVaultImpl.processQuote(address,DataTypesPeerToPeer.BorrowTransferInstructions,DataTypesPeerToPeer.GeneralQuoteInfo,DataTypesPeerToPeer.QuoteTuple).loan shadows: – ILenderVaultImpl.loan(uint256) (function)

Low/Medium/calls-loop

ChainlinkBasic.constructor(address[],address[],address,uint256) has external calls inside a loop: version = AggregatorV3Interface(_oracleAddrs[i]).version()

LenderVaultImpl.getTokenBalancesAndLockedAmounts(address[]) has external calls inside a loop: tokens[i] == address(0) || ! _addressRegistry.isWhitelistedERC20(tokens[i])

LenderVaultImpl.unlockCollateral(address,uint256[]) has external calls inside a loop: IBaseCompartment(_loan.collTokenCompartmentAddr).unlockCollToVault(_loan.collToken)

LenderVaultImpl.getTokenBalancesAndLockedAmounts(address[]) has external calls inside a loop: balances[i] = IERC20Metadata(tokens[i]).balanceOf(address(this))

ChainlinkBasic.constructor(address[],address[],address,uint256) has external calls inside a loop: oracleDecimals = AggregatorV3Interface(_oracleAddrs[i]).decimals()

Low/Medium/missing-zero-check

ChainlinkBasic.constructor(address[],address[],address,uint256).baseCurrency lacks a zero-check on : – BASE_CURRENCY = baseCurrency

Ownable.proposeNewOwner(address)._newOwnerProposal lacks a zero-check on : – _newOwner = _newOwnerProposal

LenderVaultImpl.initialize(address,address)._addressRegistry lacks a zero-check on : – addressRegistry = _addressRegistry

Low/Medium/reentrancy-benign

Reentrancy in AddressRegistry.createWrappedTokenForERC20s(DataTypesPeerToPeer.WrappedERC20TokenInfo[],string,string): External calls: – newERC20Addr = IERC20Wrapper(_erc20Wrapper).createWrappedToken(msg.sender,tokensToBeWrapped,name,symbol) State variables written after the call(s): – whitelistState[newERC20Addr] = DataTypesPeerToPeer.WhitelistState.ERC20_TOKEN

Reentrancy in FundingPoolImpl.subscribe(address,uint256): External calls: – IMysoTokenManager(mysoTokenManager).processP2PoolSubscribe(address(this),msg.sender,loanProposal,amount,_totalSubscriptions,loanTerms) State variables written after the call(s): – _earliestUnsubscribe[loanProposal][msg.sender] = block.timestamp + Constants.MIN_WAIT_UNTIL_EARLIEST_UNSUBSCRIBE – subscriptionAmountOf[loanProposal][msg.sender] += amount

Reentrancy in AddressRegistry.createWrappedTokenForERC721s(DataTypesPeerToPeer.WrappedERC721TokenInfo[],string,string): External calls: – newERC20Addr = IERC721Wrapper(_erc721Wrapper).createWrappedToken(msg.sender,tokensToBeWrapped,name,symbol) State variables written after the call(s): – whitelistState[newERC20Addr] = DataTypesPeerToPeer.WhitelistState.ERC20_TOKEN

Reentrancy in LenderVaultImpl.unlockCollateral(address,uint256[]): External calls: – IBaseCompartment(_loan.collTokenCompartmentAddr).unlockCollToVault(_loan.collToken) State variables written after the call(s): – lockedAmounts[collToken] -= totalUnlockableColl

Reentrancy in FundingPoolImpl.deposit(uint256,uint256): External calls: – IMysoTokenManager(mysoTokenManager).processP2PoolDeposit(address(this),msg.sender,amount,transferFee) – IERC20Metadata(depositToken).safeTransferFrom(msg.sender,address(this),amount + transferFee) State variables written after the call(s): – balanceOf[msg.sender] += amount

Low/Medium/reentrancy-events

Reentrancy in AddressRegistry.createWrappedTokenForERC721s(DataTypesPeerToPeer.WrappedERC721TokenInfo[],string,string): External calls: – newERC20Addr =

IERC721Wrapper(_erc721Wrapper).createWrappedToken(msg.sender,tokensToBeWrapped,name,symbol) Event emitted after the call(s): – CreatedWrappedTokenForERC721s(tokensToBeWrapped,name,symbol,newERC20Addr)

Reentrancy in LenderVaultImpl.withdraw(address,uint256): External calls: – IERC20Metadata(token).safeTransfer(_owner,amount) Event emitted after the call(s): – Withdrew(token,amount)

Reentrancy in LoanProposalImpl.repay(uint256): External calls: – IERC20Metadata(loanToken).safeTransferFrom(msg.sender,address(this),remainingLoanTokenDue + expectedTransferFee) – IERC20Metadata(collToken).safeTransfer(msg.sender,collSendAmount) Event emitted after the call(s): – Repaid(remainingLoanTokenDue,collSendAmount)

Reentrancy in FundingPoolImpl.withdraw(uint256): External calls: – IERC20Metadata(depositToken).safeTransfer(msg.sender,amount) Event emitted after the call(s): – Withdrawn(msg.sender,amount)

Reentrancy in FundingPoolImpl.executeLoanProposal(address): External calls: – ILoanProposalImpl(loanProposal).checkAndUpdateStatus() – IERC20Metadata(depositToken).safeTransfer(loanTerms.borrower,finalLoanAmount) – IERC20Metadata(depositToken).safeTransfer(arranger,arrangerFee – protocolFeeShare) – IERC20Metadata(depositToken).safeTransfer(IFactory(factory).owner(),protocolFeeShare) Event emitted after the call(s): – LoanProposalExecuted(loanProposal,loanTerms.borrower,finalLoanAmount,arrangerFee – protocolFeeShare,protocolFeeShare)

Reentrancy in LoanProposalImpl.claimRepayment(uint256): External calls: – IERC20Metadata(IFundingPoolImpl(fundingPool).depositToken()).safeTransfer(msg.sender,claimAmount) Event emitted after the call(s): – RepaymentClaimed(msg.sender,claimAmount)

Reentrancy in AddressRegistry.createWrappedTokenForERC20s(DataTypesPeerToPeer.WrappedERC20TokenInfo[],string,string): External calls: – newERC20Addr = IERC20Wrapper(_erc20Wrapper).createWrappedToken(msg.sender,tokensToBeWrapped,name,symbol) Event emitted after the call(s): – CreatedWrappedTokenForERC20s(tokensToBeWrapped,name,symbol,newERC20Addr)

Reentrancy in LenderVaultImpl.processQuote(address,DataTypesPeerToPeer.BorrowTransferInstructions,DataTypesPeerToPeer.GeneralQuoteInfo,DataTypesPeerToPeer.QuoteTuple): External calls: – collReceiver = _createCollCompartment(generalQuoteInfo.borrowerCompartmentImplementation,_loans.length) – IBaseCompartment(collCompartment).initialize(address(this),loanId) Event emitted after the call(s): – QuoteProcessed(borrower,_loan,loanId,collReceiver)

Reentrancy in LoanProposalImpl.finalizeLoanTermsAndTransferColl(uint256): External calls: – IMysoTokenManager(mysoTokenManager).processP2PoolLoanFinalization(address(this),fundingPool,staticData.collToken,staticData.arranger,msg.sender,_finalLoanAmount,_finalCollAmountReservedForDefault,_finalCollAmountReservedForConversions) – IERC20Metadata(collToken).safeTransferFrom(msg.sender,address(this),_finalCollAmountReservedForDefault + _finalCollAmountReservedForConversions + expectedTransferFee) Event emitted after the call(s): – LoanTermsAndTransferCollFinalized(_finalLoanAmount,_finalCollAmountReservedForDefault,_finalCollAmountReservedForConversions,_arrangerFee)

Reentrancy in Factory.createFundingPool(address): External calls: – IFundingPoolImpl(newFundingPool).initialize(address(this),_depositToken) Event emitted after the call(s): – FundingPoolCreated(newFundingPool,_depositToken)

Reentrancy in LenderVaultImpl.unlockCollateral(address,uint256[]): External calls: – IBaseCompartment(_loan.collTokenCompartmentAddr).unlockCollToVault(_loan.collToken) Event emitted after the call(s): – CollateralUnlocked(_owner,collToken,_loanIds,totalUnlockableColl)

Reentrancy in LoanProposalImpl.exerciseConversion(): External calls: – IERC20Metadata(staticData.collToken).safeTransfer(msg.sender,conversionAmount) Event emitted after the call(s): – ConversionExercised(msg.sender,repaymentIdx,conversionAmount)

Reentrancy in LoanProposalImpl.claimDefaultProceeds(): External calls: – IERC20Metadata(collToken).safeTransfer(msg.sender,totalCollTokenClaim) Event emitted after the call(s): – DefaultProceedsClaimed(msg.sender)

Low/Medium/timestamp

LoanProposalImpl.claimDefaultProceeds() uses timestamp for comparisons Dangerous comparisons: – dynamicData.status != DataTypesPeerToPool.LoanStatus.DEFAULTED

ChainlinkBasic._checkAndReturnLatestRoundData(address) uses timestamp for comparisons Dangerous comparisons: – roundId == 0 || answeredInRound < roundId || answer < 1 || updatedAt == 0 || updatedAt > block.timestamp

LoanProposalImpl.canSubscribe() uses timestamp for comparisons Dangerous comparisons: – (dynamicData.status == DataTypesPeerToPool.LoanStatus.IN_NEGOTIATION || (dynamicData.status == DataTypesPeerToPool.LoanStatus.BORROWER_ACCEPTED && block.timestamp < _lenderInOrOutCutoffTime()))

BorrowerGateway._processTransfers(address,address,DataTypesPeerToPeer.BorrowTransferInstructions,DataTypesPeerToPeer.Loan,uint256) uses timestamp for comparisons Dangerous comparisons: – borrowInstructions.collSendAmount < protocolFeeAmount + upfrontFee – protocolFeeAmount != 0

LoanProposalImpl.exerciseConversion() uses timestamp for comparisons Dangerous comparisons: – dynamicData.status != DataTypesPeerToPool.LoanStatus.LOAN_DEPLOYED – block.timestamp < _repayment.dueTimestamp || block.timestamp >= _repayment.dueTimestamp + staticData.conversionGracePeriod

FundingPoolImpl.unsubscribe(address,uint256) uses timestamp for comparisons Dangerous comparisons: – block.timestamp < earliestUnsubscribePerLender[msg.sender]

LoanProposalImpl.markAsDefaulted() uses timestamp for comparisons Dangerous comparisons: – dynamicData.status != DataTypesPeerToPool.LoanStatus.LOAN_DEPLOYED – block.timestamp < _getRepaymentCutoffTime(repaymentIdx)

LenderVaultImpl.withdraw(address,uint256) uses timestamp for comparisons Dangerous comparisons: – amount == 0 || amount > vaultBalance – lockedAmounts[token]

LenderVaultImpl.loan(uint256) uses timestamp for comparisons Dangerous comparisons: – loanLen == 0 || loanId > loanLen – 1

LoanProposalImpl.rollback() uses timestamp for comparisons Dangerous comparisons: – dynamicData.status != DataTypesPeerToPool.LoanStatus.BORROWER_ACCEPTED – (msg.sender == _loanTerms.borrower && block.timestamp < lenderInOrOutCutoffTime) || (block.timestamp >= lenderInOrOutCutoffTime && totalSubscriptions < _loanTerms.minTotalSubscriptions) || (block.timestamp >= lenderInOrOutCutoffTime + Constants.LOAN_EXECUTION_GRACE_PERIOD)

LoanProposalImpl.proposeLoanTerms(DataTypesPeerToPool.LoanTerms) uses timestamp for comparisons Dangerous comparisons: – status != DataTypesPeerToPool.LoanStatus.WITHOUT_LOAN_TERMS && status != DataTypesPeerToPool.LoanStatus.IN_NEGOTIATION – block.timestamp < lastLoanTermsUpdateTime + Constants.LOAN_TERMS_UPDATE_COOL_OFF_PERIOD

LenderVaultImpl.validateRepayInfo(address,DataTypesPeerToPeer.Loan,DataTypesPeerToPeer.LoanRepayInstructions) uses timestamp for comparisons Dangerous comparisons: – block.timestamp < _loan.earliestRepay || block.timestamp >= _loan.expiry

AddressRegistry.claimBorrowerWhitelistStatus(address,uint256,bytes,bytes32) uses timestamp for comparisons Dangerous comparisons: – whitelistedUntil < block.timestamp || whitelistedUntil <= whitelistedUntilPerBorrower[msg.sender]

Factory.isWhitelistedLender(address,address) uses timestamp for comparisons Dangerous comparisons: – _lenderWhitelistedUntil[whitelistAuthority][lender] > block.timestamp

QuoteHandler._isValidOnChainQuote(DataTypesPeerToPeer.OnChainQuote) uses timestamp for comparisons Dangerous comparisons: – onChainQuote.generalQuoteInfo.validUntil < block.timestamp

LoanProposalImpl._checkCurrRepaymentIdx(uint256) uses timestamp for comparisons Dangerous comparisons: – repaymentIdx == _loanTerms.repaymentSchedule.length

AddressRegistry.isWhitelistedBorrower(address,address) uses timestamp for comparisons Dangerous comparisons: – _borrowerWhitelistedUntil[whitelistAuthority][borrower] > block.timestamp

LoanProposalImpl.repay(uint256) uses timestamp for comparisons Dangerous comparisons: – dynamicData.status != DataTypesPeerToPool.LoanStatus.LOAN_DEPLOYED – (block.timestamp < currConversionCutoffTime) || (block.timestamp >= currRepaymentCutoffTime) – _loanTerms.repaymentSchedule.length – 1 == repaymentIdx

LoanProposalImpl.acceptLoanTerms(uint256) uses timestamp for comparisons Dangerous comparisons: – dynamicData.status != DataTypesPeerToPool.LoanStatus.IN_NEGOTIATION – block.timestamp < lastLoanTermsUpdateTime + Constants.LOAN_TERMS_UPDATE_COOL_OFF_PERIOD – _loanTermsUpdateTime != lastLoanTermsUpdateTime

Factory.claimLenderWhitelistStatus(address,uint256,bytes,bytes32) uses timestamp for comparisons Dangerous comparisons: – whitelistedUntil < block.timestamp || whitelistedUntil <= whitelistedUntilPerLender[msg.sender]

LenderVaultImpl.unlockCollateral(address,uint256[]) uses timestamp for comparisons Dangerous comparisons: – _loan.collUnlocked || block.timestamp < _loan.expiry

LoanProposalImpl.checkAndUpdateStatus() uses timestamp for comparisons Dangerous comparisons: – dynamicData.status != DataTypesPeerToPool.LoanStatus.READY_TO_EXECUTE

LoanProposalImpl.finalizeLoanTermsAndTransferColl(uint256) uses timestamp for comparisons Dangerous comparisons: – dynamicData.status != DataTypesPeerToPool.LoanStatus.BORROWER_ACCEPTED || block.timestamp < _lenderInOrOutCutoffTime() – _unfinalizedLoanTerms.repaymentSchedule[0].dueTimestamp <= block.timestamp + Constants.MIN_TIME_UNTIL_FIRST_DUE_DATE – IERC20Metadata(collToken).balanceOf(address(this)) != preBal + _finalCollAmountReservedForDefault + _finalCollAmountReservedForConversions

LoanProposalImpl._repaymentScheduleCheck(DataTypesPeerToPool.Repayment[]) uses timestamp for comparisons Dangerous comparisons: – repaymentSchedule[0].dueTimestamp < block.timestamp +

Constants.LOAN_TERMS_UPDATE_COOL_OFF_PERIOD + Constants.LOAN_EXECUTION_GRACE_PERIOD + Constants.MIN_TIME_UNTIL_FIRST_DUE_DATE

QuoteHandler._checkSenderAndGeneralQuoteInfo(address,DataTypesPeerToPeer.GeneralQuoteInfo) uses timestamp for comparisons Dangerous comparisons: – generalQuoteInfo.validUntil < block.timestamp

BorrowerGateway._checkDeadlineAndRegisteredVault(uint256,address) uses timestamp for comparisons Dangerous comparisons: – block.timestamp > deadline

LenderVaultImpl.processQuote(address,DataTypesPeerToPeer.BorrowTransferInstructions,DataTypesPeerToPeer.GeneralQuoteInfo,DataTypesPeerToPeer.QuoteTuple) uses timestamp for comparisons Dangerous comparisons: – _loan.expiry < SafeCast.toUint40(_loan.earliestRepay + Constants.MIN_TIME_BETWEEN_EARLIEST_REPAY_AND_EXPIRY)

LenderVaultImpl._getLoanAndRepayAmount(uint256,uint256,DataTypesPeerToPeer.GeneralQuoteInfo,DataTypesPeerToPeer.QuoteTuple) uses timestamp for comparisons Dangerous comparisons: – loanAmount > vaultLoanTokenBal – lockedAmounts[generalQuoteInfo.loanToken]

LoanProposalImpl.canUnsubscribe() uses timestamp for comparisons Dangerous comparisons: – canSubscribe() || dynamicData.status == DataTypesPeerToPool.LoanStatus.ROLLBACK

LoanProposalImpl.claimRepayment(uint256) uses timestamp for comparisons Dangerous comparisons: – repaymentIdx >= dynamicData.currentRepaymentIdx

Medium/High/incorrect-equality

LenderVaultImpl.loan(uint256) uses a dangerous strict equality: – loanLen == 0 || loanId > loanLen – 1

LoanProposalImpl.canSubscribe() uses a dangerous strict equality: – (dynamicData.status == DataTypesPeerToPool.LoanStatus.IN_NEGOTIATION || (dynamicData.status == DataTypesPeerToPool.LoanStatus.BORROWER_ACCEPTED && block.timestamp < _lenderInOrOutCutoffTime()))

LoanProposalImpl.repay(uint256) uses a dangerous strict equality: – _loanTerms.repaymentSchedule.length – 1 == repaymentIdx

LoanProposalImpl.canUnsubscribe() uses a dangerous strict equality: – canSubscribe() || dynamicData.status == DataTypesPeerToPool.LoanStatus.ROLLBACK

LoanProposalImpl.claimDefaultProceeds() uses a dangerous strict equality: – totalCollTokenClaim == 0

LoanProposalImpl._checkCurrRepaymentIdx(uint256) uses a dangerous strict equality: – repaymentIdx == _loanTerms.repaymentSchedule.length

Medium/Medium/divide-before-multiply

LenderVaultImpl._getLoanAndRepayAmount(uint256,uint256,DataTypesPeerToPeer.GeneralQuoteInfo,DataTypesPeerToPeer.QuoteTuple) performs a multiplication on the result of a division: – loanAmount = (loanPerCollUnit * (collSendAmount – expectedTransferFee)) / (10 ** IERC20Metadata(generalQuoteInfo.collToken).decimals()) – repayAmount = (loanAmount * interestRateFactor) / Constants.BASE

Medium/Medium/reentrancy-no-eth

Reentrancy in LenderVaultImpl.unlockCollateral(address,uint256[]): External calls: – IBaseCompartment(_loan.collTokenCompartmentAddr).unlockCollToVault(_loan.collToken) State variables written after the call(s): – _loan.collUnlocked = true LenderVaultImpl._loans can be used in cross function reentrancies: – LenderVaultImpl.loan(uint256) – LenderVaultImpl.processQuote(address,DataTypesPeerToPeer.BorrowTransferInstructions,DataTypesPeerToPeer.GeneralQuoteInfo,DataTypesPeerToPeer.QuoteTuple) – LenderVaultImpl.totalNumLoans() – LenderVaultImpl.unlockCollateral(address,uint256[]) – LenderVaultImpl.updateLoanInfo(uint128,uint256,uint256,address,address)

Reentrancy in FundingPoolImpl.subscribe(address,uint256): External calls: – IMysoTokenManager(mysoTokenManager).processP2PoolSubscribe(address(this),msg.sender,loanProposal,amount,_totalSubscriptions,loanTerms) State variables written after the call(s): – balanceOf[msg.sender] = _balanceOf – amount FundingPoolImpl.balanceOf can be used in cross function reentrancies: – FundingPoolImpl.balanceOf – FundingPoolImpl.unsubscribe(address,uint256) – FundingPoolImpl.withdraw(uint256) – totalSubscriptions[loanProposal] = _totalSubscriptions + amount FundingPoolImpl.totalSubscriptions can be used in cross function reentrancies: – FundingPoolImpl.totalSubscriptions – FundingPoolImpl.unsubscribe(address,uint256)

Reentrancy in LenderVaultImpl.withdraw(address,uint256): External calls: – IERC20Metadata(token).safeTransfer(_owner,amount) State variables written after the call(s): – withdrawEntered = false LenderVaultImpl.withdrawEntered can be used in cross function reentrancies: – LenderVaultImpl.withdraw(address,uint256) – LenderVaultImpl.withdrawEntered

Reentrancy in LenderVaultImpl.processQuote(address,DataTypesPeerToPeer.BorrowTransferInstructions,DataTypesPeerToPeer.GeneralQuoteInfo,DataTypesPeerToPeer.QuoteTuple): External calls: – collReceiver = _createCollCompartment(generalQuoteInfo.borrowerCompartmentImplementation,_loans.length) – IBaseCompartment(collCompartment).initialize(address(this),loanId) State variables written after the call(s): – _loans.push(_loan) LenderVaultImpl._loans can be used in cross function reentrancies: – LenderVaultImpl.loan(uint256) – LenderVaultImpl.processQuote(address,DataTypesPeerToPeer.BorrowTransferInstructions,DataTypesPeerToPeer.GeneralQuoteInfo,DataTypesPeerToPeer.QuoteTuple) – LenderVaultImpl.totalNumLoans() – LenderVaultImpl.unlockCollateral(address,uint256[]) – LenderVaultImpl.updateLoanInfo(uint128,uint256,uint256,address,address)

Medium/Medium/uninitialized-local

LenderVaultImpl.unlockCollateral(address,uint256[]).totalUnlockableColl is a local variable never initialized

LenderVaultImpl.removeSigner(address,uint256).signerWithSwappedPosition is a local variable never initialized

LoanProposalImpl.claimDefaultProceeds().totalCollTokenClaim is a local variable never initialized

Medium/Medium/unused-return

UniV3Looping.borrowCallback(DataTypesPeerToPeer.Loan,bytes) ignores return value by ISwapRouter(UNI_V3_SWAP_ROUTER).exactInputSingle(params)

BalancerV2Looping.repayCallback(DataTypesPeerToPeer.Loan,bytes) ignores return value by IERC20Metadata(loan.collToken).approve(BALANCER_V2_VAULT,collBalance)

UniV3Looping.borrowCallback(DataTypesPeerToPeer.Loan,bytes) ignores return value by IERC20Metadata(loan.loanToken).approve(UNI_V3_SWAP_ROUTER,0)

UniV3Looping.repayCallback(DataTypesPeerToPeer.Loan,bytes) ignores return value by ISwapRouter(UNI_V3_SWAP_ROUTER).exactInputSingle(params)

BalancerV2Looping.borrowCallback(DataTypesPeerToPeer.Loan,bytes) ignores return value by IERC20Metadata(loan.loanToken).approve(BALANCER_V2_VAULT,0)

UniV3Looping.repayCallback(DataTypesPeerToPeer.Loan,bytes) ignores return value by IERC20Metadata(loan.collToken).approve(UNI_V3_SWAP_ROUTER,collBalance)

BalancerV2Looping.repayCallback(DataTypesPeerToPeer.Loan,bytes) ignores return value by IERC20Metadata(loan.collToken).approve(BALANCER_V2_VAULT,0)

UniV3Looping.repayCallback(DataTypesPeerToPeer.Loan,bytes) ignores return value by IERC20Metadata(loan.collToken).approve(UNI_V3_SWAP_ROUTER,0)

BalancerV2Looping.borrowCallback(DataTypesPeerToPeer.Loan,bytes) ignores return value by IERC20Metadata(loan.loanToken).approve(BALANCER_V2_VAULT,0)

UniV3Looping.repayCallback(DataTypesPeerToPeer.Loan,bytes) ignores return value by IERC20Metadata(loan.collToken).approve(UNI_V3_SWAP_ROUTER,0)

BalancerV2Looping.repayCallback(DataTypesPeerToPeer.Loan,bytes) ignores return value by IBalancerVault(BALANCER_V2_VAULT).swap(singleSwap,fundManagement,minSwapReceive,deadline)

BalancerV2Looping.repayCallback(DataTypesPeerToPeer.Loan,bytes) ignores return value by IERC20Metadata(loan.collToken).approve(BALANCER_V2_VAULT,0)

UniV3Looping.borrowCallback(DataTypesPeerToPeer.Loan,bytes) ignores return value by IERC20Metadata(loan.loanToken).approve(UNI_V3_SWAP_ROUTER,0)

UniV3Looping.borrowCallback(DataTypesPeerToPeer.Loan,bytes) ignores return value by IERC20Metadata(loan.loanToken).approve(UNI_V3_SWAP_ROUTER,loan.initLoanAmount)

BalancerV2Looping.borrowCallback(DataTypesPeerToPeer.Loan,bytes) ignores return value by IERC20Metadata(loan.loanToken).approve(BALANCER_V2_VAULT,loan.initLoanAmount)

BalancerV2Looping.borrowCallback(DataTypesPeerToPeer.Loan,bytes) ignores return value by IBalancerVault(BALANCER_V2_VAULT).swap(singleSwap,fundManagement,minSwapReceive,deadline)

8. Appendix C. Tests

```
100 passing (3m)
```

```
---------------------------------------------------|----------|----------|----------|----------|------
----------|
```

File	% Stmts	% Branch	% Funcs	% Lines	Uncovered Lines
contracts/	100	100	100	100	
Constants.sol	100	100	100	100	
Errors.sol	100	100	100	100	
Ownable.sol	100	100	100	100	
contracts/interfaces/	100	100	100	100	
IMysoTokenManager.sol	100	100	100	100	
contracts/peer-to-peer/	99.31	95.61	100	97.88	
AddressRegistry.sol	100	95.71	100	96.91	75,91,110
BorrowerGateway.sol	98.08	84.09	100	94.67	245,252,253,270
DataTypesPeerToPeer.sol	100	100	100	100	
LenderVaultFactory.sol	88.89	75	100	92.31	40
LenderVaultImpl.sol	100	98.78	100	98.62	279,280
QuoteHandler.sol	100	100	100	100	
contracts/peer-to-peer/callbacks/	100	100	100	100	
BalancerV2Looping.sol	100	100	100	100	
UniV3Looping.sol	100	100	100	100	
contracts/peer-to-peer/compartments/	100	91.67	100	94.44	
BaseCompartment.sol	100	91.67	100	94.44	27
contracts/peer-to-					

peer/compartments/staking/	100	83.33	100	96.25	
AaveStakingCompartment.sol	100	100	100	100	
CurveLPStakingCompartment.sol	100	83.33	100	95.65	131,135,136
GLPStakingCompartment.sol	100	100	100	100	
contracts/peer-to-peer/compartments/voting/	100	87.5	100	94.44	
VoteCompartment.sol	100	87.5	100	94.44	37
contracts/peer-to-peer/interfaces/	100	100	100	100	
IAddressRegistry.sol	100	100	100	100	
IBorrowerGateway.sol	100	100	100	100	
ILenderVaultFactory.sol	100	100	100	100	
ILenderVaultImpl.sol	100	100	100	100	
IOracle.sol	100	100	100	100	
IQuoteHandler.sol	100	100	100	100	
IVaultCallback.sol	100	100	100	100	
contracts/peer-to-peer/interfaces/callbacks/	100	100	100	100	
BalancerDataTypes.sol	100	100	100	100	
IBalancerAsset.sol	100	100	100	100	
IBalancerVault.sol	100	100	100	100	
ISwapRouter.sol	100	100	100	100	
contracts/peer-to-peer/interfaces/compartments/	100	100	100	100	
IBaseCompartment.sol	100	100	100	100	
contracts/peer-to-peer/interfaces/compartments/staking/	100	100	100	100	
IStakingHelper.sol	100	100	100	100	
contracts/peer-to-peer/interfaces/oracles/	100	100	100	100	

IUniV2.sol	100	100	100	100	
contracts/peer-to-peer/interfaces/oracles/chainlink/	100	100	100	100	
AggregatorV3Interface.sol	100	100	100	100	
contracts/peer-to-peer/interfaces/wrappers/ERC20/	100	100	100	100	
IERC20Wrapper.sol	100	100	100	100	
IWrappedERC20Impl.sol	100	100	100	100	
contracts/peer-to-peer/interfaces/wrappers/ERC721/	100	100	100	100	
IERC721Wrapper.sol	100	100	100	100	
IWrappedERC721Impl.sol	100	100	100	100	
contracts/peer-to-peer/oracles/chainlink/	100	73.91	100	94.44	
ChainlinkBasic.sol	100	61.54	100	93.75	43,97
ChainlinkBasicWithWbtc.sol	100	75	100	87.5	43
OlympusOracle.sol	100	100	100	100	
UniV2Chainlink.sol	100	91.67	100	95.83	88
contracts/peer-to-peer/wrappers/ERC20/	100	83.33	100	97.96	
ERC20Wrapper.sol	100	87.5	100	96.43	47
WrappedERC20Impl.sol	100	75	100	100	
contracts/peer-to-peer/wrappers/ERC721/	100	90.91	100	98.28	
ERC721Wrapper.sol	100	93.75	100	100	
WrappedERC721Impl.sol	100	83.33	100	95.65	61
contracts/peer-to-pool/	97.66	88.46	100	96.61	
DataTypesPeerToPool.sol	100	100	100	100	
Factory.sol	94.74	71.88	100	91.23	... 179,195,196
FundingPoolImpl.sol	95.74	89.13	100	97.37	53,124

```
contracts/peer-to-pool/interfaces/            100        100        100        100

IFactory.sol                                  100        100        100        100

IFundingPoolImpl.sol                          100        100        100        100

ILoanProposalImpl.sol                         100        100        100        100

-------------------------------------------   --------   --------   --------   --------   -----------
-------------                                 --         --         --         --         -----

All files                                     98.65      89.97      97.63      96.98

-------------------------------------------   --------   --------   --------   --------   -----------
-------------                                 --         --         --         --         -----
```

STATE
MIND