hashlock.

2024

Security Audit

Lynx Finance (Trading Platform)

25/06/2024 info@hashlock.com.au Hashlock Pty Ltd

Table of Contents

CAUTION

THIS DOCUMENT IS A SECURITY AUDIT REPORT AND MAY CONTAIN CONFIDENTIAL INFORMATION. THIS INCLUDES IDENTIFIED VULNERABILITIES AND MALICIOUS CODE THAT COULD BE USED TO COMPROMISE THE PROJECT. THIS DOCUMENT SHOULD ONLY BE FOR INTERNAL USE UNTIL ISSUES ARE RESOLVED. ONCE VULNERABILITIES ARE REMEDIATED, THIS REPORT CAN BE MADE PUBLIC. THE CONTENT OF THIS REPORT IS OWNED BY HASHLOCK PTY LTD FOR THE USE OF THE CLIENT.

Executive Summary

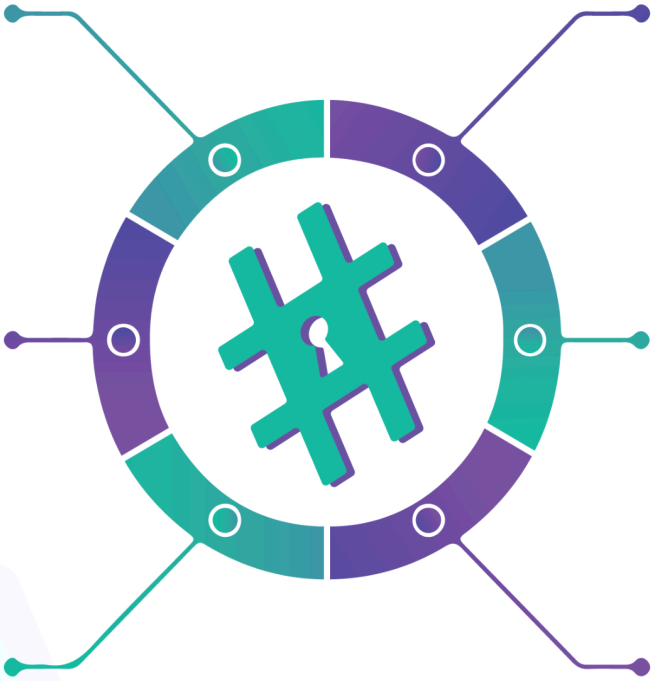
The Lynx Finance team partnered with Hashlock to conduct a security audit of their smart contracts. Hashlock manually and proactively reviewed the code to ensure the project's team and community that the deployed contracts were secure.

Project Context

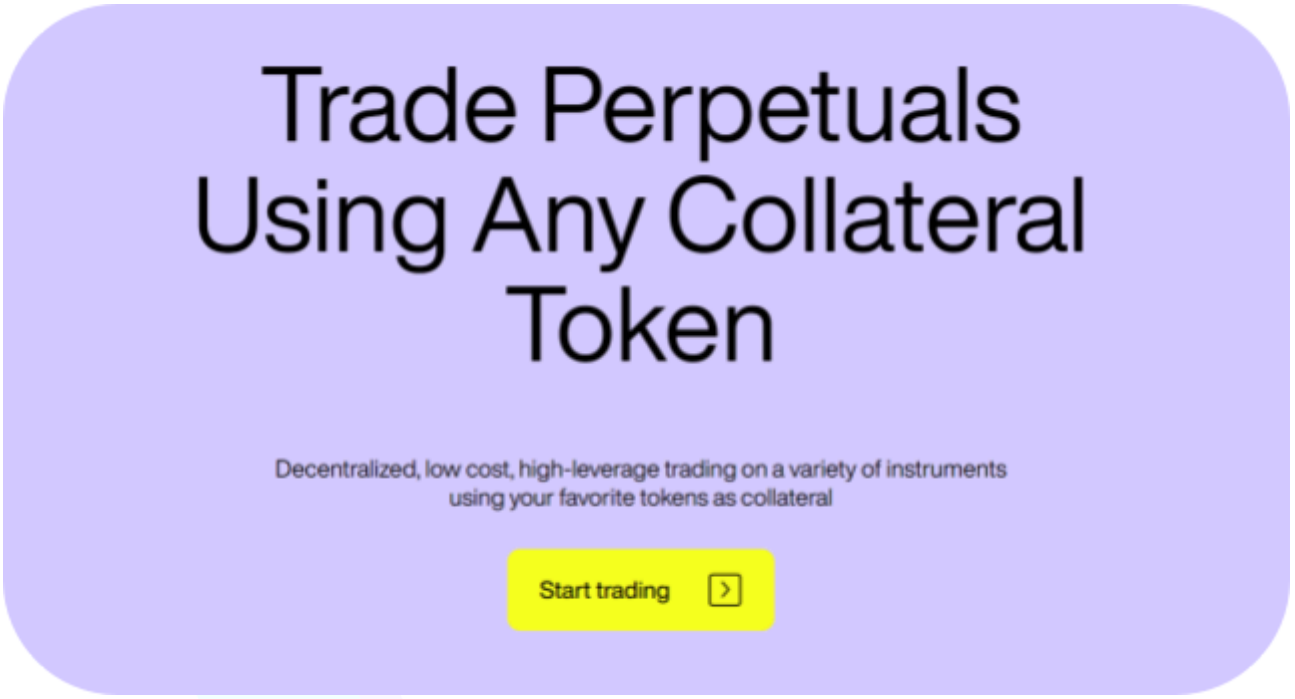
Lynx Finance is a decentralized trading platform that allows users to trade perpetuals using any token as collateral. It offers low-cost, high-leverage trading on various instruments, including crypto and forex. The platform emphasizes safety with real-time smart contract monitoring and guaranteed payouts for winning trades. It also benefits liquidity providers with single-asset deposits and no impermanent loss. Additionally, Lynx Finance allows token listing as collateral for free and provides monetization opportunities through trading fees.

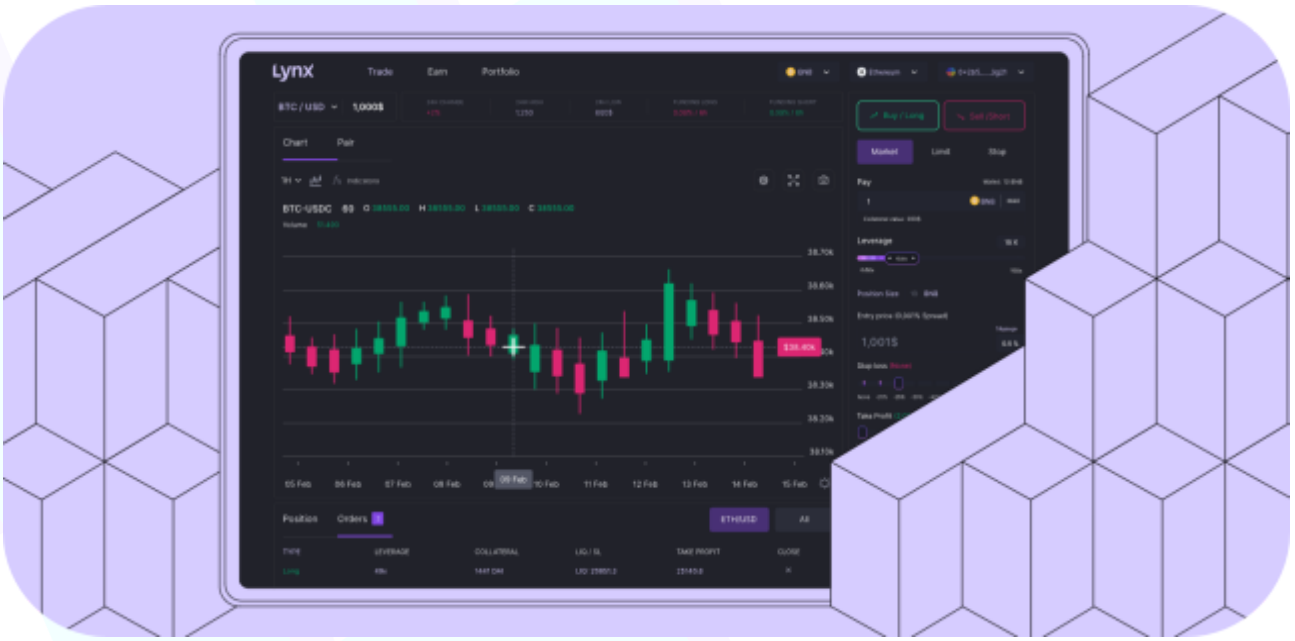
Project Name: Lynx Finance
Compiler Version: ^0.8.24
Website: https://www.lynx.finance/
Logo:

Visualised Context:



Project Visuals:





Analyse. Trade. Earn

Flexible

Trade instruments ranging from crypto to forex using any token as collateral

Safe

Real-time smart contract monitoring, guaranteed payouts for winning traders

Low cost

Optimized gas costs, competitive trading fees

Ideal for LPs

Single-asset deposits, no impermanent loss, and reduced exposure to trader PnL

Audit scope

We at Hashlock audited the solidity code within the Lynx Finance project, the scope of work included a comprehensive review of the smart contracts listed below. We tested the smart contracts to check for their security and efficiency. These tests were undertaken primarily through manual line-by-line analysis and were supported by software-assisted testing.

Description	Lynx Finance Protocol Smart Contracts
Platform	**Fantom / Solidity**
Audit Date	**June 2024**
Contract 1	AcceptableImplementationClaimableAdmin.sol
Contract 1 MD5 Hash	a59dddb36a6fdaabe6f3bef040a6cf22
Contract 2	AcceptableImplementationClaimableAdminStorage.sol
Contract 2 MD5 Hash	4e49999083871b714801367593fe432c
Contract 3	AcceptableRegistryImplementationClaimableAdmin.sol
Contract 3 MD5 Hash	f94b04b1577d26b598f792374333c2c8
Contract 4	ClaimableAdmin.sol
Contract 4 MD5 Hash	aee2b3375755e6f07be2ba8374106dca
Contract 5	EIP712Utils.sol
Contract 5 MD5 Hash	1162a60398ac2d999deea33398c5cbb5
Contract 6	LocalChip.sol
Contract 6 MD5 Hash	65d6df73a1b227a77f477fe9e5bbde64
Contract 7	OFTChip.sol
Contract 7 MD5 Hash	d31dbd7d5ba49de0ea801f6f1e7fa964
Contract 8	OFTChipAdapter.sol

Contract 8 MD5 Hash	d33f7522fdf0561ab9aa566fbdd97b84
Contract 9	BaseChip.sol
Contract 9 MD5 Hash	f89457fe9b245f84bc5bb6b3fbf46933
Contract 10	CommonScales
Contract 10 MD5 Hash	b1bb69cc40a0a0c1498c91bffa72ca20
Contract 11	Bridge.sol
Contract 11 MD5 Hash	acfa445b94c89db051e696032dc8d667
Contract 12	ChipERC20.sol
Contract 12 MD5 Hash	a3e030fbe2418f682b4fac36d93072dc
Contract 13	OriginSToken.sol
Contract 13 MD5 Hash	599edd49d7122efea8c0f701e98e2d61
Contract 14	SLxToken.sol
Contract 14 MD5 Hash	3f0d09b98d917dfd4afbafd6691d4c0f
Contract 15	ChipsIntentsVerifierV1.sol
Contract 15 MD5 Hash	a2e11bffd467cef069e320b639ee478f
Contract 16	IntentsVerifierBase.sol
Contract 16 MD5 Hash	9c1c8d0d320905cc76b89e209a5b76e4
Contract 17	LiquidityIntentsVerifierV1.sol
Contract 17 MD5 Hash	c3ab03fbc49995ae9eebd3e9066a4221
Contract 18	MultiSourceChainIntentsVerifierBase.sol
Contract 18 MD5 Hash	5bd176650328edf8775ac5e48ea3680d
Contract 19	TradeIntentsVerifierV1.sol
Contract 19 MD5 Hash	caddea474f0b40d744accc5cc3ec25d9
Contract 20	LexERC20.sol
Contract 20 MD5 Hash	d6d99f93688b7fded9ef2fc4c3e33494
Contract 21	LexPoolProxy.sol

Contract 21 MD5 Hash	c8e335f12e4bf8cfeb7283d0c8754243
Contract 22	LexPoolStorage.sol
Contract 22 MD5 Hash	194bcc3c8428fa62b87222e673d2a13d
Contract 23	LexPoolV1.sol
Contract 23 MD5 Hash	087ba033d2be47d961e92a2ac0edd62b
Contract 24	PNLR.sol
Contract 24 MD5 Hash	aadd7478fd5230e18c8d62f4a0ee4fab
Contract 25	AccountantFees.sol
Contract 25 MD5 Hash	ef77a1abfb6957519d40a71845dd0a65
Contract 26	AccountantPairGroups.sol
Contract 26 MD5 Hash	63a93cf052e195708d95d57a06651904
Contract 27	Debts.sol
Contract 27 MD5 Hash	294d1ed0b0262b5b908e7ea571024bd8
Contract 28	PoolAccountantBase.sol
Contract 28 MD5 Hash	4c880c0c7267ac71cabb5f977fcee07a
Contract 29	PoolAccountantProxy.sol
Contract 29 MD5 Hash	1855187dfc269ec62868d53ca86a2f40
Contract 30	PoolAccountantStorage.sol
Contract 30 MD5 Hash	8081bb535875c97a6a2cc93c06a2f468
Contract 31	PoolAccountantV1.sol
Contract 31 MD5 Hash	21e6b5d672ea40ab51a161a3120f5ca7
Contract 32	LexCommon.sol
Contract 32 MD5 Hash	28b06a563831b92b6bb2401d5ae0c7fd
Contract 33	SystemLocker.sol
Contract 33 MD5 Hash	e5c7d230e8532660ff29a2cb5a86ec42
Contract 34	OrderBookProxy.sol

Contract 34 MD5 Hash	a0522691250fdf48bd53e8dad76af132
Contract 35	OrderBookStorageV1.sol
Contract 35 MD5 Hash	2171a97743dddb26955cc9986aaf259c
Contract 36	OrderBookV1.sol
Contract 36 MD5 Hash	d6f2b8e527b32063007ba075913b56d3
Contract 37	RegistryProxy.sol
Contract 37 MD5 Hash	e4bc1fedc09e3ab80fdfa83d5b043acb
Contract 38	RegistryStorage.sol
Contract 38 MD5 Hash	c4cb00b046d9c37e1553bffbdccd083d
Contract 39	RegistryV1.sol
Contract 39 MD5 Hash	a6bd80be9b39894c4ce888e748bda90e
Contract 40	TradersPortalV1.sol
Contract 40 MD5 Hash	42818025f4c36c8885e8119bc9477908
Contract 41	TradingFloorProxy.sol
Contract 41 MD5 Hash	d82f3b51c33091923a811466f18c544a
Contract 42	TradingFloorV1Storage.sol
Contract 42 MD5 Hash	362f01233f56ea00a38f1ac73b46c25c
Contract 43	TradingFloorV1.sol
Contract 43 MD5 Hash	b7dbf16d0280a7a20cb9ea35cfe344a1
Contract 44	ConfidenceChecker.sol
Contract 44 MD5 Hash	7de537dc17636aca76c0937cccd54291
Contract 45	PriceAdjustmentBase.sol
Contract 45 MD5 Hash	41356cb26c756c37341dc83cfae0d43c
Contract 46	TriggersPermissionBase.sol
Contract 46 MD5 Hash	1b52529147df0cbdbecb0591e3867b4b
Contract 47	TriggersV1.sol

Contract 47 MD5 Hash	1665d39968769e21ea0d869f7578db6c
Contract 48	PythPriceValidator.sol
Contract 48 MD5 Hash	a4c9f8880d0f77d7ff59241624f6142b
Contract 49	BaseSingleKinkRateModel.sol
Contract 49 MD5 Hash	2720dda85b4a80a4410b814426c20c03
Contract 50	MutableSingleKinkRateModel.sol
Contract 50 MD5 Hash	402df1085e09f3b4be4d2fe1b621cf63
Contract 51	SteadyRateModel.sol
Contract 51 MD5 Hash	19b30736b88188618409547ae82347af
Contract 52	MutableSingleKinkFundingRateModel.sol
Contract 52 MD5 Hash	9a9751b232f5c2d81f2d692a8089186e
Contract 53	SteadyFundingRateModel.sol
Contract 53 MD5 Hash	e762113905398207a4a00716909d334b
Contract 54	MutableSingleKinkInterestRateModel.sol
Contract 54 MD5 Hash	2a565e48f1bdab2e30508f3767e09a0d
Contract 55	SteadyInterestRateModel.sol
Contract 55 MD5 Hash	cbdac1618c6aedb3e39f732be0c7deb3

Security Rating

After Hashlock's Audit, we found the smart contracts to be **"Secure"**. The contracts all follow simple logic, with correct and detailed ordering. They use a series of interfaces, and the protocol uses a list of Open Zeppelin contracts. We initially identified some significant vulnerabilities that have since been addressed.

The 'Hashlocked' rating is reserved for projects that ensure ongoing security via bug bounty programs or on-chain monitoring technology.

All issues uncovered during automated and manual analysis were meticulously reviewed and applicable vulnerabilities are presented in the Audit Findings section.

All vulnerabilities initially identified have been resolved or acknowledged.

Hashlock found:

0 High-severity vulnerabilities

6 Medium-severity vulnerabilities

10 Low-severity vulnerabilities

4 Gas Optimisations

Caution: *Hashlock's audits do not guarantee a project's success or ethics, and are not liable or responsible for security. Always conduct independent research about any project before interacting.*

Intended Smart Contract Behaviours

Claimed Behaviour	Actual Behaviour
AdministrationContracts - Manages upgradeable implementation. - Handles administrative role changes. - Delegates execution to the current implementation contract.	**Contract achieves this functionality.**
CryptographyContracts - Implements EIP-712 domain separator. - Encodes scheme for message digest.	**Contract achieves this functionality.**
Lynx - Manages core functionality, including liquidity management, maintaining the order book, and executing trading functions.	**Contract achieves this functionality.**
Chips - Allows users to trade cross-chain on Lynx - LayerZero handles cross-chain communication to mint chips on the engine chain.	**Contract achieves this functionality.**
PriceValidators - Retrieves and processes price data from Pyth. - Validates and updates prices using Pyth. - Allows admin to set price feed IDs and publication periods.	**Contract achieves this functionality.**
RateModels - Manages and calculates different funding and borrow rate models	**Contract achieves this functionality.**

Code Quality

This Audit scope involves the smart contracts of Lynx Finance, as outlined in the Audit Scope section. All contracts, libraries, and interfaces mostly follow standard best practices to help avoid unnecessary complexity that increases the likelihood of exploitation, however, some refactoring was required.

The code is very well commented on and closely follows best practice nat-spec styling. All comments are correctly aligned with code functionality.

Audit Resources

We were given the Lynx Finance projects smart contract code in the form of a GitHub repository link.

As mentioned above, code parts are well-commented. The logic is straightforward, and therefore it is easy to quickly comprehend the programming flow as well as the complex code logic. The comments help us understand the overall architecture of the protocol.

Dependencies

Per our observation, the libraries used in this smart contracts infrastructure are based on well-known industry-standard open-source projects.
Apart from libraries, its functions are used in external smart contract calls.

Severity Definitions

Significance	Description
High	High-severity vulnerabilities can result in loss of funds, asset loss, access denial, and other critical issues that will result in the direct loss of funds and control by the owners and community.
Medium	Medium-level difficulties should be solved before deployment, but won't result in loss of funds.
Low	Low-level vulnerabilities are areas that lack best practices that may cause small complications in the future.
Gas	Gas Optimisations, issues, and inefficiencies

Audit Findings

Medium

[M-01] immediateDeposit should accrue interest first

Vulnerability Details

The immediateDeposit() function in the LexPoolV1 contract allows users to deposit into the pool in a single transaction using the current exchange rate. The function transfers in the deposit amount from the user, mints the pool token to the user, and then lastly calls accrueInterest().

```
function immediateDeposit(uint256 depositAmount, bytes32 domain, bytes32 referralCode) external nonReentrant {
    require(immediateDepositAllowed, "!Allowed");

    if (depositAmount < minDepositAmount) {
        revert CapError(CapType.MIN_DEPOSIT_AMOUNT, depositAmount);
    }

    address user = msg.sender;

    takeUnderlying(user, depositAmount);

    uint256 amountToMint = underlyingAmountToOwnAmount(depositAmount);
    _mint(user, amountToMint);

    poolAccountant.accrueInterest(virtualBalanceForUtilization());

    emit ImmediateDeposit(user, depositAmount, amountToMint);
    emit LiquidityProvided(domain, referralCode, user, depositAmount, currentEpoch);
}
```

The accrueInterest() function uses the return value of the virtualBalanceForUtilization() function as its parameter. This value is then used by the accrueInterest() function to update the totalInterest, borrowIndex, and interestShare variables since the last update (last time accrueInterest was called).

Taking a closer look at the virtualBalanceForUtilization() function, we see that it takes into account currentBalanceInternal(), which is the underlying token balance of the contract.

```
function virtualBalanceForUtilization(
    uint256 extraAmount, // sum of the amounts that are held by the contract but are not part of the available ba
    int256 unrealizedFunding
) public view returns (uint256) {
    uint256 balance = currentBalanceInternal();
    uint256 subtractionFromUnrealizedFunding = unrealizedFunding < 0 ? (-unrealizedFunding).toUint256() : 0;
    uint256 pendingAmount = pendingDepositAmount + pendingWithdrawalAmount;
    if (balance < pendingAmount + extraAmount + subtractionFromUnrealizedFunding) return 0;
    return balance - pendingAmount - extraAmount - subtractionFromUnrealizedFunding;
}
```

The problem with this is that since the accrueInterest() function is called after the current deposit's underlying tokens were transferred in, they will be incorrectly counted in the accrueInterest calculations (borrowRate) for the duration from when accrueInterest was last called till the current time.

Impact

This discrepancy will result in inaccurate interest calculations, which can have several significant consequences and potentially lead to losses for users.

Recommendation

To prevent this issue, the accrueInterest() function should be called at the start of the immediateDeposit() function. This will ensure that the interest calculation only considers the balance prior to the new deposit.

Status

Resolved

[M-02] requestRedeemInternal should accrue interest first

Vulnerability Details

The requestRedeemInternal() function in the LexPoolV1 contract is called by the requestRedeem() and requestRedeemViaIntent() functions to create a redeem request. The function transfers in the redeem amount from the user, increments the pendingWithdrawalAmount, and creates a withdrawal request for the user.

```
function requestRedeemInternal(address user, uint256 amount, uint256 minAmountOut) internal {
    uint256 epoch = currentEpoch + epochsDelayRedeem;
    require(pendingDeposits[epoch][user].amount == 0, "Exists deposit");

    _transfer(user, address(this), amount);
    uint256 rate = currentExchangeRate;
    uint256 currentUnderlyingAmountOut = ownAmountToUnderlyingAmountInternal(rate, amount);

    require(minAmountOut <= currentUnderlyingAmountOut, "MinAmountOut too high");
    uint256 maxUnderlyingAmountOut =
        (currentUnderlyingAmountOut * (FRACTION_SCALE + maxExtraWithdrawalAmountF)) / FRACTION_SCALE;
    pendingWithdrawalAmount += maxUnderlyingAmountOut;
    verifyUtilizationForLPs();

    PendingRedeem storage pendingRedeem = pendingRedeems[epoch][user];
    if (pendingRedeem.amount == 0) {
        // The first time for this user on this epoch
        // So this user is not yet in the array
        pendingRedeemersArr[epoch].push(user);
    }

    pendingRedeem.amount = pendingRedeem.amount + amount;
    pendingRedeem.minAmountOut = pendingRedeem.minAmountOut + minAmountOut;
    pendingRedeem.maxAmountOut = pendingRedeem.maxAmountOut + maxUnderlyingAmountOut;

    emit RedeemRequest(user, amount, minAmountOut, epoch);
}
```

The issue is that pendingWithdrawalAmount is being modified without calling accrueInterest(). This pendingWithdrawalAmount value is used in the virtualBalanceForUtilization() function to calculate the protocol's current virtual balance, which is used by accrueInterest to calculate the borrowRate and update the totalInterest, borrowIndex, and interestShare variables since the last update.

```
function virtualBalanceForUtilization(
    uint256 extraAmount, // sum of the amounts that are held by the contract but are not part of the available ba
    int256 unrealizedFunding
) public view returns (uint256) {
    uint256 balance = currentBalanceInternal();
    uint256 subtractionFromUnrealizedFunding = unrealizedFunding < 0 ? (-unrealizedFunding).toUint256() : 0;
    uint256 pendingAmount = pendingDepositAmount + pendingWithdrawalAmount;
    if (balance < pendingAmount + extraAmount + subtractionFromUnrealizedFunding) return 0;
    return balance - pendingAmount - extraAmount - subtractionFromUnrealizedFunding;
}
```

Consequently, the missing accrueInterest() function call means the next time accrueInterest() is called, it will take into account the new virtual balance, including the pendingWithdrawalAmount, for the duration from when accrueInterest() was last called till the current time. This could be long before the redeem request was made.

Furthermore, certain checks are done in accrueInterest() to ensure the protocol stays within the set limits, such as the maximum borrow rate and other critical parameters. Skipping these checks can cause future transactions to revert, disrupting the protocol's normal operation.

Impact

This miscalculation can result in inaccurate interest distribution, which disadvantages certain users. Moreover, it can cause some rates to exceed the allowed maximum, leading to unexpected reverts in subsequent transactions.

Recommendation

To avoid this issue, the accrueInterest() function should be called at the beginning of the requestRedeemInternal() function. This guarantees that the subsequent interest calculation utilises the correct virtual balance. Additionally, limit checks can be added at the end of the function to ensure they are not exceeded with the updated balances.

Status

Resolved

[M-03] cancelRedeems should accrue interest first

Vulnerability Details

The cancelRedeems() function in the LexPoolV1 contract allows the cancellation of redeem requests whose matching epoch has passed. The function deletes the users withdrawal request, decrements the pendingWithdrawalAmount, and transfers the held amount back to the user.

```
function cancelRedeems(address[] calldata users, uint256[] calldata epochs) external nonReentrant {
    require(users.length == epochs.length, "!ArrayLengths");

    uint256 maxEpochToCancel = currentEpoch - 1;
    for (uint8 index = 0; index < users.length; index++) {
        address user = users[index];
        uint256 epoch = epochs[index];
        require(epoch <= maxEpochToCancel, "Epoch too soon");

        PendingRedeem memory pendingRedeem = pendingRedeems[epoch][user];
        delete pendingRedeems[epoch][user];

        pendingWithdrawalAmount -= pendingRedeem.maxAmountOut;
        _transfer(address(this), user, pendingRedeem.amount);

        emit CanceledRedeem(user, epoch, pendingRedeem.amount);
    }
}
```

The issue is that pendingWithdrawalAmount is being modified without calling accrueInterest(). This value is used in the virtualBalanceForUtilization function to calculate the protocol's current virtual balance, which is then used by accrueInterest to determine the borrowRate and update the totalInterest, borrowIndex, and interestShare variables since the last update.

```
function virtualBalanceForUtilization(
    uint256 extraAmount, // sum of the amounts that are held by the contract but are not part of the available l
    int256 unrealizedFunding
) public view returns (uint256) {
    uint256 balance = currentBalanceInternal();
    uint256 subtractionFromUnrealizedFunding = unrealizedFunding < 0 ? (-unrealizedFunding).toUint256() : 0;
    uint256 pendingAmount = pendingDepositAmount + pendingWithdrawalAmount;
    if (balance < pendingAmount + extraAmount + subtractionFromUnrealizedFunding) return 0;
    return balance - pendingAmount - extraAmount - subtractionFromUnrealizedFunding;
}
```

Consequently, the missing accrueInterest() function call means the next time accrueInterest is called, it will take into account the new virtual balance, including the pendingWithdrawalAmount, for the duration from when accrueInterest was last called till the current time.

Impact

This will result in inaccurate interest calculations, affecting specific rates which will put certain users at a disadvantage and cause potential losses.

Recommendation

To prevent this issue, the accrueInterest() function should be called at the start of the cancelRedeems function. This will ensure that the interest calculation uses the correct virtual balance.

Status

Resolved

[M-04] Unsafe use of ERC20 Transfer

Vulnerability Details

Some tokens do not implement the ERC20 standard properly. For example, Tether (USDT)'s transfer() function does not return a boolean as the ERC20 specification requires and instead has no return value. When these sorts of tokens are cast to IERC20/ERC20, their function signatures do not match and therefore the calls made will revert.

```
function sweepTokens(ERC20 _token, uint256 _amount) external onlyOwner {

      require(address(_token) != address(this), "CANNOT_SWEEP_SELF");



      _token.transfer(owner(), _amount); //@audit unsafe use


      emit TokensSwept(address(_token), owner(), _amount);

   }
```

Impact

The function could revert permanently

Recommendation

It is recommended to use the SafeERC20's safeTransfer() from OpenZeppelin instead.

Status

Resolved

[M-05] notContract modifier that ensures call is from EOA might not hold true in the future

Vulnerability Details

For notContract modifier, tx.origin is used to ensure that the caller is from an EOA and not a smart contract.

```
modifier notContract() {

        require(tx.origin == _msgSender()); //@audit

        _;

    }
```

However, according to EIP 3074

This EIP introduces two EVM instructions AUTH and AUTHCALL. The first sets a context variable authorized based on an ECDSA signature. The second sends a call as the authorized account. This essentially delegates control of the externally owned account (EOA) to a smart contract.

Therefore, using tx.origin to ensure msg.sender is an EOA will not hold true in the event EIP 3074 goes through.

Impact

Using modifier notContract to ensure calls are made only from EOA will not hold true in the event EIP 3074 goes through.

Recommendation

Recommend using OpenZepellin's isContract function. Note that there are edge cases like contract in constructor that can bypass this and hence caution is required when using this.

Status

Acknowledged

[M-06] reduceReserves function should call accrue interest first

Vulnerability Details

The reduceReserves() function in the LexPoolV1 contract is used to transfer out the current interestShare to a specified address and reset the interestShare.

```
function reduceReserves(address _to) external onlyAdmin {
    require(msg.sender == IRegistryV1(registry).feesManagers(address(underlying)), "!feesManager");
    uint256 interestShare = poolAccountant.readAndZeroReserves();
    uint256 reservesToSend = interestShare;

    if (reservesToSend > 0) {
        underlying.safeTransfer(_to, reservesToSend);
    }
}
```

This is done in the readAndZeroReserves() function, where the current interestShare is returned and then zeroed out.

```
function readAndZeroReserves()
    external
    returns (uint256 accumulatedInterestShare)
{
    require(msg.sender == address(lexPool), "!Pool");
    // Read
    accumulatedInterestShare = interestShare;

    // Zero
    interestShare = 0;
}
```

The problem arises because the accrueInterest() function uses the interestShare variable to update the totalInterest, borrowIndex, and interestShare since the last update (i.e., the last time accrueInterest was called). Consequently, calling the reduceReserves() function before calling accrueInterest() will lead to an inaccurate

calculation for the duration from the last time accrueInterest was called until the next call.

Additionally, the accrueInterest() function performs several checks to ensure the protocol maintains set limits, such as the maximum borrow rate and other critical parameters. If these checks are skipped, it can cause future transactions to revert, thereby disrupting the protocol's normal operation.

Impact

This miscalculation can lead to inaccurate interest distribution, potentially disadvantages certain users. Additionally, this can lead to situations where certain rates exceed the allowed maximum, causing unexpected reverts in subsequent transactions.

Recommendation

Modify the reduceReserves() function to include a call to accrueInterest() before resetting the interestShare.

Alternatively, ensure that the team manually calls accrueInterest() before executing the reduceReserves() function.

Status

Resolved

Low

[L-01] Incorrect Check in onlyTraderIntentVerifier modifier

Vulnerability Details

The onlyTraderIntentVerifier modifier in the TradersPortalV1 contract incorrectly checks if the msg.sender is equal to the fee manager when it should be checking if it equals the tradeIntentsVerifier.

```
modifier onlyTraderIntentVerifier() {
    require(
        msg.sender == IRegistryV1(tradingFloor.registry()).feesManagers(NATIVE_UNDERLYING_ADDRESS),
        "NOT_TRADER_INTENT_VERIFIER"
    );
    _;
}
```

Impact

Since the onlyTraderIntentVerifier modifier is not currently used anywhere in the contract, this issue is not a problem at the moment. However, if it is used in the future after an upgrade, it will perform an incorrect check.

Recommendation

The modifier should be updated to correctly check if the msg.sender equals the tradeIntentsVerifier instead of the fee manager. This will ensure that the correct validation is performed if the modifier is utilised in the future.

Status

Resolved

[L-02] Use Ownable 2Step

Description

By using Ownable2Step or Ownable2StepUpgradeable instead of the standard `Ownable` contract, you add an additional layer of security to your smart contracts. These contracts require the new owner to actively accept the ownership transfer, reducing the risk of mistakenly transferring the ownership to the wrong address.

Using these contracts can prevent potential security risks and provide better control over contract ownership, ultimately enhancing the overall security of your Solidity contracts.

Recommendation

Use ownable2Step

Status

Acknowledged

[L-03] LiquidityIntentsVerifierV1 contract lacks processing/canceling deposits/redeems functionality

Vulnerability Details

The LiquidityIntentsVerifierV1 contract enables users to perform gasless transactions across different chains without needing to switch chains.

However, the LiquidityIntentsVerifierV1 contract does not include the processDeposit/processRedeems nor the cancelDeposits/cancelRedeems functions, which are necessary for processing and canceling pending deposits or redeems. Consequently, if a user needs to call these functions, they must switch chains, obtain sufficient gas fees, and manually call the function. This requirement undermines the purpose of the gasless trading feature.

Impact

This issue affects the usability and efficiency of the protocol's gasless trading feature, forcing users to switch chains and pay additional gas fees, thereby defeating the purpose of gasless trading.

Recommendation

Implement the processDeposit/processRedeems and cancelDeposits/cancelRedeems functions in the LiquidityIntentsVerifierV1 contract to allow users to call these functions without switching chains.

Status

Acknowledged

[L-04] borrowRateMax can be bypassed as the check is only done for the first transaction that includes `calcAccrueFundingValues` in a block

Vulnerability Details

The protocol implements a borrowRateMax to restrict the borrow rate from exceeding a set limit. This check is enforced every time the calcAccrueInterestValues() function is called. The problem with the current implementation is that for multiple transactions calling the calcAccrueInterestValues() function within the same block, the check will only be performed on the first transaction.

```
function calcAccrueInterestValues(uint256 availableCash)
    public
    view
    returns (bool freshened, uint256 totalInterestNew, uint256 borrowIndexNew, uint256 interestShareNew)
{
    uint256 currentBlockTimestamp = block.timestamp;
    uint256 accrualBlockTimestampPrior = accrualBlockTimestamp;
    // WARNING: What happens for subsecond blocks? Is there any problem here?
    if (accrualBlockTimestampPrior == currentBlockTimestamp) {
        return (false, totalInterest, borrowIndex, interestShare);
    }

    uint256 borrowsPrior = totalBorrows;
    uint256 interestSharePrior = interestShare;
    uint256 borrowIndexPrior = borrowIndex;

    uint256 borrowRate = irm.getBorrowRate(calcUtilization(availableCash, borrowsPrior));
    if (borrowRate > borrowRateMax) {
        revert CapError(CapType.BORROW_RATE_MAX, borrowRate);
    }
    ...
}
```

As a result, any transactions after the first that causes the borrow rate to surpass the borrowRateMax will not be reverted. This can lead to reverts on subsequent transactions in the next block, even those unrelated to the borrow rate. In the worst case, it could block critical transactions like liquidations or stop losses.

Impact

This vulnerability can lead to a situation where the borrow rate exceeds the allowed maximum, potentially causing unexpected reverts in subsequent transactions. This can disrupt critical operations, including liquidations and stop losses, leading to losses.

Recommendation

Modify the calcAccrueInterestValues() function to perform the borrow rate check before the possible return, ensuring it is executed for every transaction within a block.

```
function calcAccrueInterestValues(uint256 availableCash)
    public
    view
    returns (bool freshened, uint256 totalInterestNew, uint256 borrowIndexNew, uint256 interestShareNew)
{
    uint256 currentBlockTimestamp = block.timestamp;
    uint256 accrualBlockTimestampPrior = accrualBlockTimestamp;

    // Perform the check before the possible return
    uint256 borrowsPrior = totalBorrows;
    uint256 borrowRate = irm.getBorrowRate(calcUtilization(availableCash, borrowsPrior));
    if (borrowRate > borrowRateMax) {
        revert CapError(CapType.BORROW_RATE_MAX, borrowRate);
    }
    // WARNING: What happens for subsecond blocks? Is there any problem here?
    if (accrualBlockTimestampPrior == currentBlockTimestamp) {
        return (false, totalInterest, borrowIndex, interestShare);
    }

    uint256 interestSharePrior = interestShare;
    uint256 borrowIndexPrior = borrowIndex;
    ...
}
```

Status

Acknowledged

[L-05] borrowRateMax can be bypassed as the check is only done for the first transaction that includes `calcAccrueFundingValues` in a block

Vulnerability Details

The protocol implements a borrowRateMax to restrict the borrow rate from exceeding a set limit. This check is enforced every time the calcAccrueInterestValues() function is called. The problem with the current implementation is that for multiple transactions calling the calcAccrueInterestValues() function within the same block, the check will only be performed on the first transaction.

```
function calcAccrueInterestValues(uint256 availableCash)
    public
    view
    returns (bool freshened, uint256 totalInterestNew, uint256 borrowIndexNew, uint256 interestShareNew)
{
    uint256 currentBlockTimestamp = block.timestamp;
    uint256 accrualBlockTimestampPrior = accrualBlockTimestamp;
    // WARNING: What happens for subsecond blocks? Is there any problem here?
    if (accrualBlockTimestampPrior == currentBlockTimestamp) {
        return (false, totalInterest, borrowIndex, interestShare);
    }

    uint256 borrowsPrior = totalBorrows;
    uint256 interestSharePrior = interestShare;
    uint256 borrowIndexPrior = borrowIndex;

    uint256 borrowRate = irm.getBorrowRate(calcUtilization(availableCash, borrowsPrior));
    if (borrowRate > borrowRateMax) {
        revert CapError(CapType.BORROW_RATE_MAX, borrowRate);
    }
    ...
}
```

As a result, any transactions after the first that causes the borrow rate to surpass the borrowRateMax will not be reverted. This can lead to reverts on subsequent transactions in the next block, even those unrelated to the borrow rate. In the worst case, it could block critical transactions like liquidations or stop losses.

Impact

This vulnerability can lead to a situation where the borrow rate exceeds the allowed maximum, potentially causing unexpected reverts in subsequent transactions. This can disrupt critical operations, including liquidations and stop losses, leading to losses.

Recommendation

Modify the calcAccrueInterestValues() function to perform the borrow rate check before the possible return, ensuring it is executed for every transaction within a block.

```
function calcAccrueInterestValues(uint256 availableCash)
    public
    view
    returns (bool freshened, uint256 totalInterestNew, uint256 borrowIndexNew, uint256 interestShareNew)
{
    uint256 currentBlockTimestamp = block.timestamp;
    uint256 accrualBlockTimestampPrior = accrualBlockTimestamp;

    // Perform the check before the possible return
    uint256 borrowsPrior = totalBorrows;
    uint256 borrowRate = irm.getBorrowRate(calcUtilization(availableCash, borrowsPrior));
    if (borrowRate > borrowRateMax) {
        revert CapError(CapType.BORROW_RATE_MAX, borrowRate);
    }
    // WARNING: What happens for subsecond blocks? Is there any problem here?
    if (accrualBlockTimestampPrior == currentBlockTimestamp) {
        return (false, totalInterest, borrowIndex, interestShare);
    }

    uint256 interestSharePrior = interestShare;
    uint256 borrowIndexPrior = borrowIndex;
    ...
}
```

Status

Acknowledged

[L-06] `fundingRateMax` can be bypassed as the check is only done for the first transaction that includes `calcAccrueFundingValues` in a block

Vulnerability Details

The calcAccrueFundingValues() function in the protocol enforces a maximum funding rate (fundingRateMax) to ensure the funding rate does not exceed a set limit. However, this check is currently performed only once per block, allowing subsequent transactions within the same block to bypass this restriction.

```
function calcAccrueFundingValues(uint16 pairId)
    public
    view
    returns (bool freshened, int256 valueLong, int256 valueShort)
{
    PairFunding memory f = pairFunding[pairId];
    valueLong = f.accPerOiLong;
    valueShort = f.accPerOiShort;

    uint256 timediff = block.timestamp - f.lastUpdateTimestamp;
    if (timediff == 0) {
        // Already fresh
        return (false, valueLong, valueShort);
    }

    PairOpenInterest memory openInterest = openInterestInPair[pairId];
    uint256 maxPairOpenInterest = pairs[pairId].maxOpenInterest;

    uint256 fundingRate = frm.getFundingRate(pairId, openInterest.long, openInterest.short, maxPairOpenInterest);
    if (fundingRate > fundingRateMax) {
        revert CapError(CapType.FUNDING_RATE_MAX, fundingRate);
    }
    ...
}
```

As a result, if multiple transactions occur within the same block, only the first transaction will enforce the funding rate check. Subsequent transactions can cause the funding rate to exceed fundingRateMax, potentially leading to reverts in the next block and disrupting critical transactions.

Impact

This vulnerability can lead to situations where the funding rate exceeds the allowed maximum, causing unexpected reverts in subsequent transactions.

Recommendation

Modify the calcAccrueFundingValues() function to perform the funding rate check before the possible return, ensuring it is executed for every transaction within a block.

```
function calcAccrueFundingValues(uint16 pairId)
    public
    view
    returns (bool freshened, int256 valueLong, int256 valueShort)
{
    PairFunding memory f = pairFunding[pairId];
    valueLong = f.accPerOiLong;
    valueShort = f.accPerOiShort;

    // Perform the check before the possible return
    PairOpenInterest memory openInterest = openInterestInPair[pairId];
    uint256 maxPairOpenInterest = pairs[pairId].maxOpenInterest;
    uint256 fundingRate = frm.getFundingRate(pairId, openInterest.long, openInterest.short, maxPairOpenInterest);
    if (fundingRate > fundingRateMax) {
        revert CapError(CapType.FUNDING_RATE_MAX, fundingRate);
    }

    uint256 timediff = block.timestamp - f.lastUpdateTimestamp;
    if (timediff == 0) {
        // Already fresh
        return (false, valueLong, valueShort);
    }
    ...
}
```

Status

Acknowledged

[L-07] Use .call to send ETH

Vulnerability Details

The .transfer() function intends to transfer an ETH amount with a fixed amount of 2300 gas .transfer() functions which may supply different amounts of gas in the future. Additionally, if the recipient implements a fallback function containing some sort of logic, this may inevitably revert.

```
function sweepNative(uint256 _amount) external onlyAdmin {

        payable(admin).transfer(_amount); //@audit use .call{value: _amount}("") instead
of .transfer(_amount)

    }
```

A good article about it

https://consensys.net/diligence/blog/2019/09/stop-using-soliditys-transfer-now/

Impact

The function could revert permanently

Recommendation

Consider using .call() instead with the checks-effects-interactions pattern implemented correctly. Careful consideration needs to be made to prevent reentrancy.

Status

Acknowledged

[L-08] Intents have no deadline

Vulnerability Details

The protocol enables gasless cross-chain trading by having users create a signed request on the origin chain, publishing the intent on the engine chain, and executing it via a whitelisted bot that transfers chips and opens the position.

A potential problem with the current implementation is that intents lack a validity deadline, meaning certain intents can be triggered indefinitely unless a new request for the same action is created and executed.

For example, assume a user creates an intent to redeem a specific amount of shares, and this action causes the verifyUtilizationForLPs function to fail, the intent will revert, preventing its completion at that moment.

[L-09] verifyIntent_traderRequest_openNewPosition Function Can Be Executed with a Different referralCode or domain Then What the Trader Requested

Vulnerability Details

The verifyIntent_traderRequest_openNewPosition() function in the TradeIntentsVerifierV1 contract allows users to create an open position request on another chain, which is then triggered by a whitelisted trigger bot.

The user can specify their opening position parameters, which are verified in the function to ensure they are not tampered with throughout the process. However, neither the referralCode nor the domain are verified. Additionally, the verifyIntent_traderRequest_openNewPosition() function lacks access control, meaning anyone can fulfill a user request.

```
function verifyIntent_traderRequest_openNewPosition(
    UserRequestPayload_OpenPosition calldata payload,
    uint8 v,
    bytes32 r,
    bytes32 s,
    bytes32 domain,
    bytes32 referralCode,
    bool runCapTests
) external payable notContract {
    bytes32 domainSeparator = getDomainSeparatorForAssetInternal(payload.positionRequestIdentifiers.settlementAsset);

    // Recover trader
    address trader = recoverOpenPositionPayloadSigner(payload, v, r, s, domainSeparator);

    // Sanity
    require(payload.positionRequestIdentifiers.trader == trader, "NOT_SIGNED_BY_TRADER");

    // Validate & Register Nonce
    validateNonceForActionAndIncrease(
        trader, uint8(TradeIntentsVerifierActions.REQUEST_POSITION_OPEN), payload.nonce
    );

    // Perform action
    bytes32 positionId = getTradersPortal().traderRequest_openNewPosition{value: msg.value}(
        payload.positionRequestIdentifiers,
        payload.positionRequestParams,
        payload.orderType,
        domain,
        referralCode,
        runCapTests
    );
    ...
}
```

As a result, any user can complete another user's request using their own referral code and domain. Therefore, any benefits from the referral code will be rewarded to the user who completed the transaction instead of the trader who opened the position.

Impact

The lack of verification and access control makes it easy for malicious actors to exploit this vulnerability. This can lead to misuse of referral codes, resulting in financial incentives being misdirected.

Recommendation

The referralCode and domain parameters should be included in the signature verification process to ensure that the original values are used when submitting the transaction.

Additionally, the verifyIntent_traderRequest_openNewPosition() function could be restricted to whitelist addresses to prevent unauthorised users from calling the function.

Status

Acknowledged

[L-10] verifyIntent_epochDeposit Function Can Be Executed with a Different referralCode or domain Than What the Liquidity Provider Requested

Vulnerability Details

The verifyIntent_epochDeposit() function in the LiquidityIntentsVerifierV1 contract allows users to create a deposit request on another chain, which is then triggered by a whitelisted trigger bot.

The user can specify specific parameters such as the pool, amount, and minAmount, which are then verified in the function to ensure they are not tampered with throughout the process. However, neither the referralCode nor the domain are verified. Additionally, the verifyIntent_epochDeposit() function lacks access control, meaning anyone can fulfill a user request.

```
function verifyIntent_epochDeposit(
    LiquidityProviderRequestPayload_EpochDeposit calldata payload,
    uint8 v,
    bytes32 r,
    bytes32 s,
    bytes32 domain,
    bytes32 referralCode
) external payable {
    bytes32 domainSeparator = domainSeparatorForAsset(payload.pool);

    require(recoverEpochDepositPayloadSigner(payload, v, r, s, domainSeparator) == payload.liquidityProvider, "NOT_SIGNE

    require(payload.amount > 0, "AMOUNT_ZERO");

    // Validate & Register Nonce
    validateNonceForActionAndIncrease(
        payload.liquidityProvider,
        uint8(LiquidityIntentsVerifierActions.REQUEST_EPOCH_DEPOSIT),
        payload.nonce
    );

    ILexPoolV1(payload.pool).requestDepositViaIntent(
        payload.liquidityProvider,
        payload.amount,
        payload.minAmountOut,
        domain,
        referralCode
    );
    ...
}
```

As a result, any user can complete another user's request using their own referral code and domain. Therefore, any benefits from the referral code will be rewarded to the user who completed the transaction instead of the liquidity provider who initiated the deposit.

Impact

The lack of verification and access control makes it easy for malicious actors to exploit this vulnerability. This can lead to misuse of referral codes, resulting in financial incentives being misdirected.

Recommendation

The referralCode and domain parameters should be included in the signature verification process to ensure that the original values are used when submitting the transaction.

Additionally, the verifyIntent_epochDeposit() function could be restricted to whitelist addresses to prevent unauthorised users from calling the function.

Status

Acknowledged

```
function requestRedeemInternal(address user, uint256 amount, uint256 minAmountOut) internal {
    uint256 epoch = currentEpoch + epochsDelayRedeem;
    require(pendingDeposits[epoch][user].amount == 0, "Exists deposit");

    _transfer(user, address(this), amount);
    uint256 rate = currentExchangeRate;
    uint256 currentUnderlyingAmountOut = ownAmountToUnderlyingAmountInternal(rate, amount);

    require(minAmountOut <= currentUnderlyingAmountOut, "MinAmountOut too high");
    uint256 maxUnderlyingAmountOut =
        (currentUnderlyingAmountOut * (FRACTION_SCALE + maxExtraWithdrawalAmountF)) / FRACTION_SCALE;
    pendingWithdrawalAmount += maxUnderlyingAmountOut;
    verifyUtilizationForLPs();

    PendingRedeem storage pendingRedeem = pendingRedeems[epoch][user];
    if (pendingRedeem.amount == 0) {
        // The first time for this user on this epoch
        // So this user is not yet in the array
        pendingRedeemersArr[epoch].push(user);
    }

    pendingRedeem.amount = pendingRedeem.amount + amount;
    pendingRedeem.minAmountOut = pendingRedeem.minAmountOut + minAmountOut;
    pendingRedeem.maxAmountOut = pendingRedeem.maxAmountOut + maxUnderlyingAmountOut;

    emit RedeemRequest(user, amount, minAmountOut, epoch);
}
```

The problem with this is that unless a user creates another withdrawal intent request, the original intent will remain valid indefinitely. This means any user can complete it even if the user changes their mind and no longer wishes to proceed with the transaction after some time.

The use of a deadline is common practice when using signatures as it allows users to set a duration for when the transaction should be completed.

Impact

This issue can lead to unintended transactions being executed long after the user has issued the intent, potentially against their current wishes.

Recommendation

Implement a deadline parameter in the intent creation process. This parameter should define the maximum time frame within which the intent is valid and can be executed.

Status

Acknowledged

Gas

[G-01] Nesting if-statements is cheaper than using &&

Description

Nesting if-statements avoids the stack operations of setting up and using an extra jumpdest, and saves 6 gas

There are 20 instances of this issue in the codebase

Recommendation

It is recommended to use nested if statements

Status

Acknowledged

[G-02] Cache array length outside of loop

Description

If not cached, the solidity compiler will always read the length of the array during each iteration. That is, if it is a storage array, this is an extra sload operation (100 additional extra gas for each iteration except for the first) and if it is a memory array, this is an extra mload operation (3 additional gas for each iteration except for the first).

There are 7 instances of this issue in the codebase

Recommendation

It is recommended to cache array first and then loop over it

Status

Acknowledged

[G-03] Use Custom Errors

Description

Instead of using error strings, to reduce deployment and runtime cost, you should use Custom Errors. This would save both deployment and runtime cost
Reference: Source

There are 238 instances of this issue in the codebase

Recommendation

It is recommended to use custom ERRORS

Status

Acknowledged

[G-04] Using extra variable will use more gas

Description

The function reduceReserves() in LexPoolV1.sol is using an extra variable "reservesToSend" to store the value of "interestShare" but if we see above the value of poolAccountant.readAndZeroReserves() is actually being stored in "interestShare" variable and then the value of "interestShare" is stored in "reservesToSend"

```
function reduceReserves(address _to) external onlyAdmin {
        require(
            msg.sender ==
                IRegistryV1(registry).feesManagers(address(underlying)),
            "!feesManager"
        );
        uint interestShare = poolAccountant.readAndZeroReserves();
        uint reservesToSend = interestShare; //@audit using extra variable

        if (reservesToSend > 0) {
```

```
            underlying.safeTransfer(_to, reservesToSend);
        }
    }
```

Recommendation

It is recommended not to use extra variable when there isn't any need

Status

Acknowledged

Test Coverage

The Lynx team ensures a high standard with a robust test coverage of 87.05%, indicating thorough validation of its functionalities.

87.05% Statements `1304/1498` **73.9%** Branches `1138/1540` **80.37%** Functions `438/545` **87.39%** Lines `1691/1935`

File		Statements		Branches		Functions		Lines	
AdministrationContracts/		73.33%	22/30	56.25%	9/16	83.33%	10/12	73.81%	31/42
CryptographyContracts/		85.71%	6/7	50%	1/2	100%	3/3	90%	9/10
Lynx/Chips/		100%	7/7	83.33%	5/6	100%	3/3	100%	12/12
Lynx/Chips/EngineChip/		92.59%	50/54	80.3%	53/66	87.5%	14/16	93.65%	59/63
Lynx/Chips/OFTChip/		78.57%	44/56	75.64%	59/78	88.89%	16/18	80.95%	51/63
Lynx/Common/		83.33%	5/6	50%	6/12	77.78%	7/9	80%	8/10
Lynx/IntentsVerifier/		95.36%	144/151	69%	69/100	91.53%	54/59	94.8%	164/173
Lynx/IntentsVerifier/NonceMechanisms/		66.67%	4/6	100%	4/4	50%	2/4	75%	6/8
Lynx/Lex/		5.56%	2/36	2.38%	2/84	8.33%	3/36	12.2%	5/41
Lynx/Lex/LexPool/		86.77%	164/189	67.61%	96/142	85.25%	52/61	86.74%	229/264
Lynx/Lex/PNLR/		0%	0/9	0%	0/6	16.67%	1/6	23.08%	3/13
Lynx/Lex/PoolAccountant/		88.77%	253/285	66.39%	158/238	81.98%	91/111	88.83%	342/385
Lynx/Locks/		100%	4/4	100%	0/0	100%	4/4	100%	6/6
Lynx/OrderBook/		90.32%	28/31	82.5%	33/40	91.67%	11/12	86.36%	38/44
Lynx/Registry/		90.91%	60/66	77.42%	48/62	93.94%	31/33	92.94%	79/85
Lynx/TradersPortal/		98.25%	112/114	84.29%	177/210	96.77%	30/31	97.71%	128/131
Lynx/TradingFloor/		97.69%	211/216	92.92%	223/240	100%	45/45	98.39%	305/310
Lynx/Triggers/		97.7%	170/174	92.08%	186/202	94%	47/50	97.46%	192/197
Lynx/interfaces/		100%	0/0	100%	0/0	100%	0/0	100%	0/0
Peripheral/Chips/		0%	0/17	0%	0/8	0%	0/6	0%	0/21
Peripheral/PriceValidators/		0%	0/21	0%	0/10	0%	0/11	0%	0/31
Peripheral/RateModels/BaseModels/SingleKinkRateModel/		100%	10/10	62.5%	5/8	100%	5/5	100%	15/15
Peripheral/RateModels/BaseModels/SteadyRateModel/		0%	0/1	0%	0/2	50%	1/2	33.33%	1/3
Peripheral/RateModels/FRM/		100%	6/6	100%	4/4	100%	4/4	100%	6/6
Peripheral/RateModels/IRM/		100%	2/2	100%	0/0	100%	4/4	100%	2/2

Centralisation

The Lynx Finance project values security and utility over decentralisation.

The owner executable functions within the protocol increase security and functionality but depend highly on internal team responsibility.

The Lynx Finance team does not intend to retain ownership of the contracts long-term. Currently, Lynx Finance is in a beta stage, and the ownership structure is set to change. The plan includes establishing a council to determine security parameters under a multi-signature scheme. Additionally, other parameters may be subject to community governance upon the launch of our token.

Conclusion

After Hashlocks analysis, the Lynx Finance project seems to have a sound and well-tested code base, now that our findings have been resolved or acknowledged. Overall, most of the code is correctly ordered and follows industry best practices. The code is well commented on as well. To the best of our ability, Hashlock is not able to identify any further vulnerabilities.

Our Methodology

Hashlock strives to maintain a transparent working process and to make our audits a collaborative effort. The objective of our security audits are to improve the quality of systems and upcoming projects we review and to aim for sufficient remediation to help protect users and project leaders. Below is the methodology we use in our security audit process.

Manual Code Review:

In manually analysing all of the code, we seek to find any potential issues with code logic, error handling, protocol and header parsing, cryptographic errors, and random number generators. We also watch for areas where more defensive programming could reduce the risk of future mistakes and speed up future audits. Although our primary focus is on the in-scope code, we examine dependency code and behaviour when it is relevant to a particular line of investigation.

Vulnerability Analysis:

Our methodologies include manual code analysis, user interface interaction, and whitebox penetration testing. We consider the project's website, specifications, and whitepaper (if available) to attain a high level understanding of what functionality the smart contract under review contains. We then communicate with the developers and founders to gain insight into their vision for the project. We install and deploy the relevant software, exploring the user interactions and roles. While we do this, we brainstorm threat models and attack surfaces. We read design documentation, review other audit results, search for similar projects, examine source code dependencies, skim open issue tickets, and generally investigate details other than the implementation.

Documenting Results:

We undergo a robust, transparent process for analysing potential security vulnerabilities and seeing them through to successful remediation. When a potential issue is discovered, we immediately create an issue entry for it in this document, even though we have not yet verified the feasibility and impact of the issue. This process is vast because we document our suspicions early even if they are later shown to not represent exploitable vulnerabilities. We generally follow a process of first documenting the suspicion with unresolved questions, then confirming the issue through code analysis, live experimentation, or automated tests. Code analysis is the most tentative, and we strive to provide test code, log captures, or screenshots demonstrating our confirmation. After this we analyse the feasibility of an attack in a live system.

Suggested Solutions:

We search for immediate mitigations that live deployments can take and finally we suggest the requirements for remediation engineering for future releases. The mitigation and remediation recommendations should be scrutinised by the developers and deployment engineers, and successful mitigation and remediation is an ongoing collaborative process after we deliver our report, and before the contracts details are made public.

Disclaimers

Hashlock's Disclaimer

Hashlock's team has analysed these smart contracts in accordance with the best industry practices at the date of this report, in relation to: cybersecurity vulnerabilities and issues in the smart contract source code, the details of which are disclosed in this report, (Source Code); the Source Code compilation, deployment and functionality (performing the intended functions).

Due to the fact that the total number of test cases are unlimited, the audit makes no statements or warranties on security of the code. It also cannot be considered as a sufficient assessment regarding the utility and safety of the code, bugfree status or any other statements of the contract. While we have done our best in conducting the analysis and producing this report, it is important to note that you should not rely on this report only. We also suggest conducting a bug bounty program to confirm the high level of security of this smart contract.

Hashlock is not responsible for the safety of any funds, and is not in any way liable for the security of the project.

Technical Disclaimer

Smart contracts are deployed and executed on a blockchain platform. The platform, its programming language, and other software related to the smart contract can have their own vulnerabilities that can lead to attacks. Thus, the audit can't guarantee explicit security of the audited smart contracts.

About Hashlock

Hashlock is an Australian based company aiming to help facilitate the successful widespread adoption of distributed ledger technology. Our key services all have a focus on security, as well as projects that focus on streamlined adoption in the business sector.

Hashlock is excited to continue to grow its partnerships with developers and other web3 oriented companies to collaborate on secure innovation, helping businesses and decentralised entities alike.

Website: hashlock.com.au
Contact: info@hashlock.com.au


#Hashlock.