Cyberscope

Audit Report

ØxLiquidity

June 2024

SHA256 2e0a02dc1246b8c6c8d8a544ee8ff666213bbe62f9ce4c15bd7ce7a397d99cef

Audited by © cyberscope

Table of Contents

Review

Testing Deploy	https://testnet.bscscan.com/address/0xe635bb2b7a36ee8d803f21523bc29febe4e60447

Audit Updates

Initial Audit	23 May 2024 https://github.com/cyberscope-io/audits/blob/main/0xlp/v1/audit.pdf
Corrected Phase 2	3 Jun 2024

Source Files

Filename	SHA256
contracts/LiquidityMarketplace.sol	2e0a02dc1246b8c6c8d8a544ee8ff666213bbe62f9ce4c15bd7ce7a397d99cef

Overview

The LiquidityMarketplace contract facilitates secure and structured financial interactions involving locked liquidity tokens through two primary mechanisms, deals and auctions. Users can initiate, manage, and engage in loan deals by setting terms such as loan amount, interest rate, and duration, ensuring that only eligible borrowers and lenders participate. Concurrently, the contract allows users to auction their liquidity tokens by setting parameters like starting price and duration, providing options for immediate purchase or traditional bidding. This dual functionality leverages ownership verification and event-driven updates to ensure transactional integrity and user control over both loan and auction processes.

Audit Scope

The contract code interacts with an external `liquidityLocker` contract, utilizing it to verify the ownership of liquidity and to facilitate the transfer of liquidity locks. However, it is important to note that the `liquidityLocker` address and its associated functionalities are outside the scope of this current audit. Any interactions with or modifications to this external contract should be approached with caution, as changes to the `liquidityLocker` could impact the integrity and security of the primary contract's operations. Special attention should be given to the manner in which the `liquidityLocker` manages and verifies ownership, as well as how it handles the transfer of locks, to ensure that these actions remain secure and function as intended within the broader system.

Deals Functionality

initializeDeal Functionality

The contract allows users to initialize loan deals with specific terms. A user can create a new deal by specifying the loaned amount, the interest rate, and the duration of the loan. This is done using a liquidity token and an associated lock index that verifies the user's ownership of the locked liquidity. The deal becomes part of the contract’s managed records and can be activated and engaged by other parties. It ensures that only the rightful owner can initialize a deal and enforces the minimum conditions for the interest rate, deal amount, and loan duration.

activateDeal Functionality

The users have the ability to activate a deal that has been initialized. This functionality verifies that the deal is owned by the contract and makes it active, enabling lenders to fund the deal. It checks the ownership of the locked liquidity and emits a notification event when a deal is successfully activated. This step is crucial for moving the deal from an initialized state to one where it is open for engagement from potential lenders.

makeDeal Functionality

The contract provides a pathway for lenders to fund active deals. When a lender decides to fund a deal, they must send an amount equal to or greater than the deal amount. The functionality ensures that a borrower cannot lend to their own deal, secures the transfer of funds minus a fee to the borrower, and establishes the lender’s role in the deal. It effectively moves the deal into an ongoing state and locks the terms until repayment or cancellation.

cancelDeal Functionality

Borrowers are given the control to cancel a deal, provided it has not yet been funded by a lender. This functionality ensures that only the borrower who initialized the deal can cancel it, and it also reverts the ownership of the associated lock in the liquidity locker to the borrower. This serves as a safeguard allowing borrowers to retrieve control over their assets if they decide not to proceed with the loan.

repayLoan Functionality

The contract facilitates borrowers to repay their loans including the agreed-upon interest. This is possible only if the loan has not already been repaid, the correct amount is paid, and the loan duration has not been exceeded. Upon successful repayment, the borrower reclaims ownership of the locked liquidity, and the lender receives the repayment. This mechanism ensures the closure of the loan agreement upon fulfillment of the repayment terms.

claimCollateral Functionality

Lenders have the ability to claim the collateral if the borrower fails to repay the loan within the agreed duration. This functionality confirms that only the registered lender for a deal can initiate a claim and that the loan period has indeed expired. The ownership of the locked liquidity is transferred to the lender, securing the lender's investment by compensating them with the collateral in case of default by the borrower.

Auction Functionality

startAuction Functionality

The contract empowers users to initiate auctions for their locked liquidity tokens. This function allows a user to set the starting price, the immediate sell price, the minimum bid increment (bid step), and the auction's duration. It includes a feature to enable or disable immediate selling at a fixed price. The auction is validated to ensure the user owns the locked liquidity and that all prices and duration parameters are positive and greater than zero. Each auction is recorded in the contract with its unique ID and associated with the user, who becomes the auction owner. The event AuctionStarted is emitted to notify about the auction setup.

activateAuction Functionality

Users can activate an auction that has been previously initialized. This feature checks that the auction is not already active and confirms the contract's ownership of the underlying liquidity. Upon activation, the auction's status is set to active, making it available for bids. The timestamp marks the start, and the event AuctionActivated is emitted, signaling that the auction is officially open for bidding.

immediatelyBuy Functionality

This function enables users to buy the auctioned item immediately if the auction allows for it. It checks that the auction is still active, not already sold immediately, and that the buyer is not the auction owner. The transaction requires the buyer to pay at least the immediate sell price. Upon a successful immediate buy, the ownership of the locked liquidity is transferred to the buyer, the auction is marked as finished, and related fees are processed. The ImmediatelyBought event is triggered to record the transaction details.

makeBid Functionality

Users are provided the ability to place bids on active auctions that are not set for immediate selling. Bids must exceed the current highest bid by at least the minimum bid increment. The auction must still be within its active duration. Each bid is added to the user's total for that auction, updating the highest bidder information and emitting the BidMade event, which logs the bid details.

withdrawAuctionLiquidity Functionality

This functionality allows the auction owner to withdraw the locked liquidity if no bids have been placed that meet the auction conditions by the end of the auction period. This ensures that the auction owner can reclaim their asset if the auction does not result in a satisfactory bid, providing a safeguard against unwanted lock-in.

claimAuction Functionality

Winning bidders can claim the auctioned item once the auction concludes and if they are the highest bidder. This function transfers the ownership of the locked liquidity to the highest bidder, assuming the auction didn't end through an immediate buy. This process is validated to ensure that the auction has indeed ended, and the AuctionWon event is emitted following a successful claim.

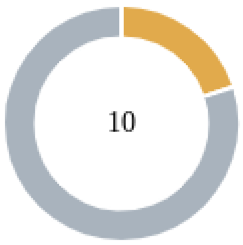
claimAuctionReward Functionality

Auction owners can claim the highest bid amount at the end of the auction if the auction was not immediately bought. This function handles the transfer of the highest bid amount minus the owner fee to the auction owner, ensuring that the rewards of the auction are distributed correctly. The transaction details are recorded through the AuctionRewardClaimed event.

withdrawBid Functionality

Bidders who have not won the auction have the option to withdraw their bids after the auction ends. This function checks that the caller has an eligible bid to withdraw, ensuring that the highest bidder cannot withdraw their winning bid. This mechanism protects the integrity of the auction process while allowing non-winning bidders to reclaim their funds, documented by the BidWithdrawn event.

Findings Breakdown



●	Critical	0
●	Medium	2
●	Minor / Informative	8

Severity		Unresolved	Acknowledged	Resolved	Other
●	Critical	0	0	0	0
●	Medium	0	2	0	0
●	Minor / Informative	8	0	0	0

Diagnostics

● Critical ● Medium ● Minor / Informative

Severity	Code	Description	Status
●	IRM	Interest Rate Manipulation	Acknowledged
●	OLLE	Overlooked Liquidity Lock Expiry	Acknowledged
●	ADS	Arbitrary Duration Setting	Unresolved
●	CR	Code Repetition	Unresolved
●	MEE	Missing Events Emission	Unresolved
●	MU	Modifiers Usage	Unresolved
●	RIL	Redundant ImmediatelySell Logic	Unresolved
●	L04	Conformance to Solidity Naming Conventions	Unresolved
●	L18	Multiple Pragma Directives	Unresolved
●	L19	Stable Compiler Version	Unresolved

IRM - Interest Rate Manipulation

Criticality	Medium
Location	contracts/LiquidityMarketplace.sol#L399
Status	Acknowledged

Description

The contract is designed in such a way that the borrower can set the `interestRate`. However, there is no incentive for the borrower to set a high value for the `interestRate` since the higher the interest rate, the more the borrower will have to pay back. This creates a scenario where the borrower can manipulate the `interestRate` to minimize their repayment, potentially undermining the intended financial structure of the contract.

```
function repayLoan(uint256 dealId) public payable nonReentrant
{
    Deal storage deal = deals[dealId];
    uint256 repayAmount = deal.dealAmount +
        (deal.dealAmount * deal.interestRate) /
        10000;
    require(!deal.isRepaid, "Deal already repaid");
    require(msg.sender == deal.borrower, "Sender is not a
borrower");
    require(deal.lender != address(0), "Nothing to repay");
    require(msg.value >= repayAmount, "Insuffitient payable
amount");
...
```

Recommendation

It is recommended to introduce a specific interest rate or modify the logic so that the `interestRate` is not set by the borrower. Instead, consider having the interestRate determined by a predefined formula, set by the lender, or defined by the contract logic to ensure fairness and maintain the integrity of the financial structure.

Team Update

The team has acknowledged that this is not a security issue and states:

It is our main business logic. Users can set their own interest rate on initializing deal. Then we moderate the deal and approve it to show it in the U

OLLE - Overlooked Liquidity Lock Expiry

Criticality	Medium
Location	contracts/LiquidityMarketplace.sol#L254,276,384,413,438,542,599,625
Status	Acknowledged

Description

The contract uses an external contract in order to determine the transaction's flow. The external contract is untrusted. As a result, it may produce security issues and harm the transactions.

Specifically, the contract is structured to facilitate the transfer of liquidity lock ownership based on user-initiated actions, such as during the repayment of loans or the conclusion of auctions. However, it currently does not account for the actual expiration time of these liquidity locks when performing transfers. This oversight can lead to a scenario where the contract's actions imply that liquidity is securely locked and transferred according to the terms of the deal or auction, while in reality, the lock period may have already expired, potentially allowing for the premature withdrawal of the tokens. This discrepancy undermines the security and reliability of the contract, affecting the integrity of transactional outcomes.

```
liquidityLocker = _liquidityLocker;
...
(, , , , uint256 lockId, ) =
liquidityLocker.getUserLockForTokenAtIndex(
    address(this),
    deal.lpToken,
    deal.lockIndex
);
liquidityLocker.transferLockOwnership(
    deal.lpToken,
    deal.lockIndex,
    lockId,
    payable(msg.sender)
);
```

Recommendation

It is recommended to incorporate checks that validate the lock status and expiration timing before executing any transfer of lock ownership. This could involve verifying the total amount that remains locked and any other conditions related to the timing of the lock. By integrating these verifications, the contract can ensure that all transfers of lock ownership are executed accurately and only when the underlying assets are securely locked as intended. This approach will mitigate risks associated with unlocked and withdrawn liquidity, thus preserving the intended safeguards and transactional fidelity of the contract operations.

Team Update

The team has acknowledged that this is not a security issue and states:

It's resolved during the moderation process. We don't approve deals with a liquidity unlock date earlier than the deal end date.

ADS - Arbitrary Duration Setting

Criticality	Minor / Informative
Location	contracts/LiquidityMarketplace.sol#L297,454
Status	Unresolved

Description

The contract includes the variables `loanDuration` and `duration` that set the duration of various processes. However, there are no checks or validations to ensure that the user initializing these variables does not pass an arbitrary future duration. As a result, this can render some of the contract's functionality useless since the specified time may never be reached, leading to processes that cannot be completed.

```
function initializedeal(
    address lptoken,
    uint256 lockindex,
    uint256 dealamount,
    uint256 interestrate,
    uint256 loanduration
) public uniquelock(msg.sender, lptoken, lockindex) {
    require(loanduration > 0, "loan duration must be greater than
0");
...
}
...
function startauction(
    address lptoken,
    uint256 lockindex,
    uint256 startprice,
    uint256 imeddiatelysellprice,
    uint256 bidstep,
    uint256 duration,
    bool immediatelysell
) public uniquelock(msg.sender, lptoken, lockindex) {
    require(duration > 0, "duration must be greater than 0");
...
```

Recommendation

It is recommended to introduce checks for the time being set. Ensure that the duration values are within a reasonable and predefined range to prevent impractical durations that could hinder the contract's operations. This will enhance the reliability and functionality of the contract by ensuring all processes can reach completion.

CR - Code Repetition

Criticality	Minor / Informative
Location	contracts/LiquidityMarketplace.sol#L372,398,419,534,626
Status	Unresolved

Description

The contract contains repetitive code segments. There are potential issues that can arise when using code segments in Solidity. Some of them can lead to issues like gas efficiency, complexity, readability, security, and maintainability of the source code. It is generally a good idea to try to minimize code repetition where possible.

```
(, , , , uint256 lockId, ) =
liquidityLocker.getUserLockForTokenAtIndex(
    address(this),
    deal.lpToken,
    deal.lockIndex
);
liquidityLocker.transferLockOwnership(
    deal.lpToken,
    deal.lockIndex,
    lockId,
    payable(msg.sender)
);
```

```
uint256 feeAmount = (msg.value * ownerFee) / 10000;
(bool feeSent, ) = payable(feeReceiver).call{value: feeAmount}("");
require(feeSent, "Failed to send fee");

(bool sent, ) = payable(auction.owner).call{
    value: msg.value - feeAmount
}("");
require(sent, "Failed to send funds");
```

Recommendation

The team is advised to avoid repeating the same code in multiple places, which can make the contract easier to read and maintain. The authors could try to reuse code wherever possible, as this can help reduce the complexity and size of the contract. For instance, the contract could reuse the common code segments in an internal function in order to avoid repeating the same code in multiple places.

MEE - Missing Events Emission

Criticality	Minor / Informative
Location	contracts/LiquidityMarketplace.sol#L259
Status	Unresolved

Description

The contract performs actions and state mutations from external methods that do not result in the emission of events. Emitting events for significant actions is important as it allows external parties, such as wallets or dApps, to track and monitor the activity on the contract. Without these events, it may be difficult for external parties to accurately determine the current state of the contract.

```
function setOwnerFee(uint256 _ownerFee) external onlyOwner {
    require(_ownerFee < 10000, "Owner fee must be less than
10000");
    ownerFee = _ownerFee;
}
```

Recommendation

It is recommended to include events in the code that are triggered each time a significant action is taking place within the contract. These events should include relevant details such as the user's address and the nature of the action taken. By doing so, the contract will be more transparent and easily auditable by external parties. It will also help prevent potential issues or disputes that may arise in the future.

MU - Modifiers Usage

Criticality	Minor / Informative
Location	contracts/LiquidityMarketplace.sol#L354,380,386,503,566
Status	Unresolved

Description

The contract is using repetitive statements on some methods to validate some preconditions. In Solidity, the form of preconditions is usually represented by the modifiers. Modifiers allow you to define a piece of code that can be reused across multiple functions within a contract. This can be particularly useful when you have several functions that require the same checks to be performed before executing the logic within the function.

```
require(deal.borrower != address(0), "Deal is empty");
require(
    checkLiquidityOwner(address(this), deal.lpToken,
deal.lockIndex),
    "Contract does not owner of this liquidity"
);
require(deal.lender == address(0), "Deal already has a lender");
require(
    deal.borrower != msg.sender,
    "Borrower cannot make loan for himself"
);
require(deal.borrower == msg.sender, "Caller not lock owner");
require(deal.lender == address(0), "Cannot cancel processing deal");
require(auction.owner != msg.sender, "Sender is auction owner");
 require(
    auction.startTime + auction.duration < block.timestamp,
    "Auction is active yet"
);
```

Recommendation

The team is advised to use modifiers since it is a useful tool for reducing code duplication and improving the readability of smart contracts. By using modifiers to perform these checks, it reduces the amount of code that is needed to write, which can make the smart contract more efficient and easier to maintain.

RIL - Redundant ImmediatelySell Logic

Criticality	Minor / Informative
Location	contracts/LiquidityMarketplace.sol#L522
Status	Unresolved

Description

The contract is set up to handle immediate purchases in auctions through two parameters, an `immediatelySell` boolean and an `imeddiatelySellPrice`. While the `immediatelySell` boolean is designed to explicitly enable or disable the option for immediate purchase, the contract could streamline its operations by relying solely on whether the `imeddiatelySellPrice` is set, making the boolean redundant. Currently, both are used to control the functionality of the `immediatelyBuy` method. This redundancy could lead to unnecessary complexity and potential misconfigurations where the boolean and the price might contradict each other, leading to user confusion and errors in auction handling.

```
    function startAuction(
         address lpToken,
         uint256 lockIndex,
         uint256 startPrice,
         uint256 imeddiatelySellPrice,
         uint256 bidStep,
         uint256 duration,
         bool immediatelySell
    ) public {
    ...
    }

    function immediatelybuy(uint256 auctionid) external payable
nonreentrant{
         auction storage auction = auctions[auctionid];
         require(msg.sender.code.length == 0, "caller cannot be a smart
contract");
         require(
             auction.immediatelySell,
             "Immediately selling is disabled for this lottery"
         );
         require(auction.owner != msg.sender, "Sender is auction
owner");
         require(
             msg.value >= auction.imeddiatelySellPrice,
             "Insuffitient payable amount"
         );
    ...
```

Recommendation

It is recommended to simplify the immediate purchase functionality by removing the `immediatelySell` boolean and solely utilizing the `imeddiatelySellPrice` to dictate the availability of this option. If a positive `imeddiatelySellPrice` is set, the auction should allow immediate purchases, otherwise if it is not set or zero, the option should be disabled. This change would reduce contract complexity and increase clarity for users, ensuring that the presence of a non-zero immediate sell price clearly signals the availability of immediate purchase options. This approach streamlines user interactions and reduces the risk of contradictory settings within the auction setup.

L04 - Conformance to Solidity Naming Conventions

Criticality	Minor / Informative
Location	contracts/LiquidityMarketplace.sol#L259,264,286,292,684
Status	Unresolved

Description

The Solidity style guide is a set of guidelines for writing clean and consistent Solidity code. Adhering to a style guide can help improve the readability and maintainability of the Solidity code, making it easier for others to understand and work with.

The followings are a few key points from the Solidity style guide:

1. Use camelCase for function and variable names, with the first letter in lowercase (e.g., myVariable, updateCounter).
2. Use PascalCase for contract, struct, and enum names, with the first letter in uppercase (e.g., MyContract, UserStruct, ErrorEnum).
3. Use uppercase for constant variables and enums (e.g., MAX_VALUE, ERROR_CODE).
4. Use indentation to improve readability and structure.
5. Use spaces between operators and after commas.
6. Use comments to explain the purpose and behavior of the code.
7. Keep lines short (around 120 characters) to improve readability.

```
uint256 _ownerFee
address _feeReceiver
address _address
...
```

Recommendation

By following the Solidity naming convention guidelines, the codebase increased the readability, maintainability, and makes it easier to work with.

Find more information on the Solidity documentation
https://docs.soliditylang.org/en/v0.8.17/style-guide.html#naming-convention.

L18 - Multiple Pragma Directives

Criticality	Minor / Informative
Location	contracts/LiquidityMarketplace.sol#L7,33,123
Status	Unresolved

Description

If the contract includes multiple conflicting pragma directives, it may produce unexpected errors. To avoid this, it's important to include the correct pragma directive at the top of the contract and to ensure that it is the only pragma directive included in the contract.

```
pragma solidity ^0.8.0;
```

Recommendation

It is important to include only one pragma directive at the top of the contract and to ensure that it accurately reflects the version of Solidity that the contract is written in.

By including all required compiler options and flags in a single pragma directive, the potential conflicts could be avoided and ensure that the contract can be compiled correctly.

L19 - Stable Compiler Version

Criticality	Minor / Informative
Location	contracts/LiquidityMarketplace.sol#L7,33,123
Status	Unresolved

Description

The `^` symbol indicates that any version of Solidity that is compatible with the specified version (i.e., any version that is a higher minor or patch version) can be used to compile the contract. The version lock is a mechanism that allows the author to specify a minimum version of the Solidity compiler that must be used to compile the contract code. This is useful because it ensures that the contract will be compiled using a version of the compiler that is known to be compatible with the code.

```
pragma solidity ^0.8.0;
```

Recommendation

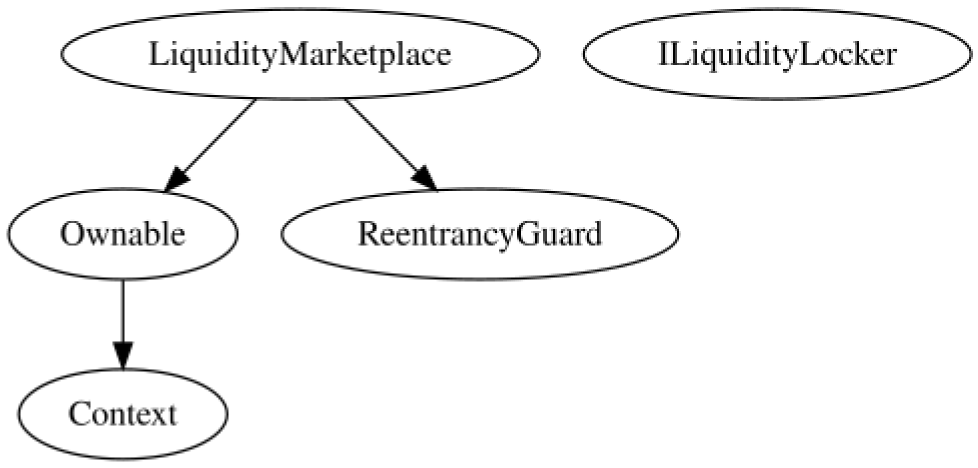
The team is advised to lock the pragma to ensure the stability of the codebase. The locked pragma version ensures that the contract will not be deployed with an unexpected version. An unexpected version may produce vulnerabilities and undiscovered bugs. The compiler should be configured to the lowest version that provides all the required functionality for the codebase. As a result, the project will be compiled in a well-tested LTS (Long Term Support) environment.

Functions Analysis

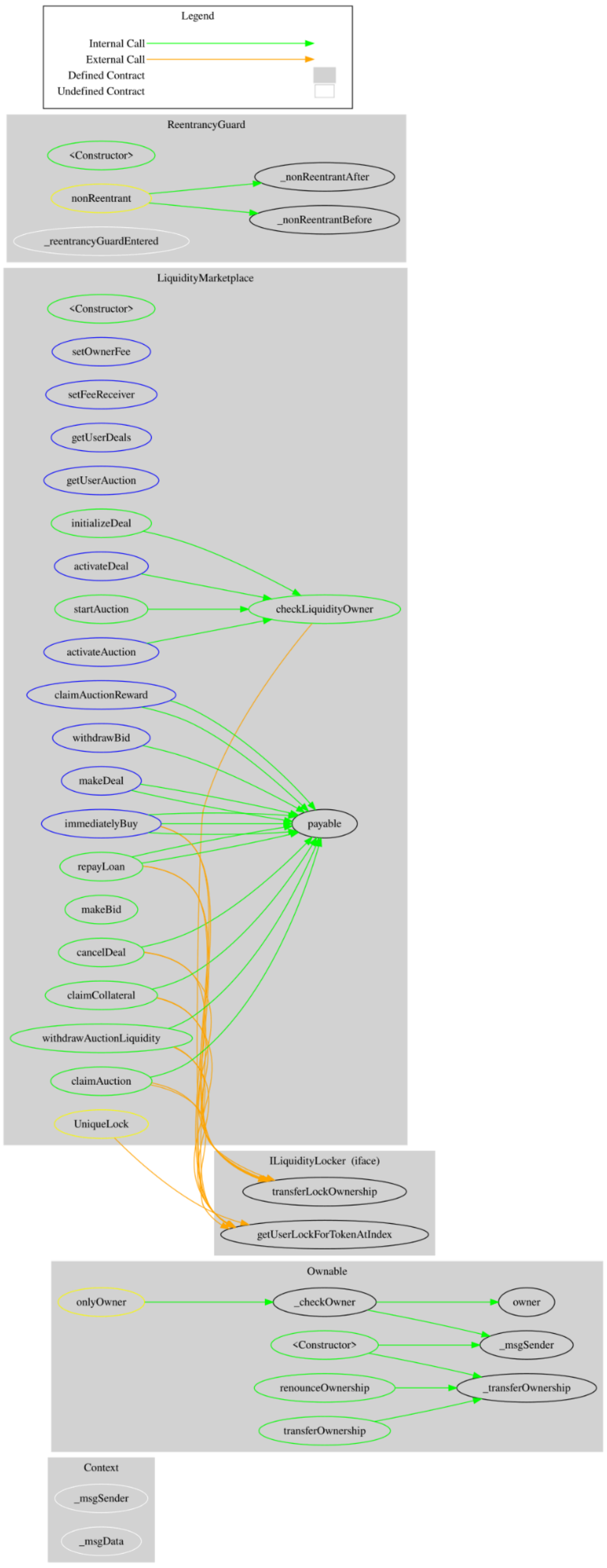
Contract	Type	Bases		
	Function Name	**Visibility**	**Mutability**	**Modifiers**
LiquidityMarketplace	Implementation	Ownable, ReentrancyGuard		
		Public	✓	-
	setOwnerFee	External	✓	onlyOwner
	setFeeReceiver	External	✓	onlyOwner
	checkLiquidityOwner	Public		-
	getUserDeals	External		-
	getUserAuction	External		-
	initializeDeal	Public	✓	UniqueLock
	activateDeal	External	✓	-
	makeDeal	External	Payable	nonReentrant
	cancelDeal	Public	✓	-
	repayLoan	Public	Payable	nonReentrant
	claimCollateral	Public	✓	nonReentrant
	startAuction	Public	✓	UniqueLock
	activateAuction	External	✓	-
	immediatelyBuy	External	Payable	nonReentrant
	makeBid	Public	Payable	-
	withdrawAuctionLiquidity	Public	✓	-

	claimAuction	Public	✓	nonReentrant
	claimAuctionReward	External	✓	nonReentrant
	withdrawBid	External	✓	nonReentrant

Inheritance Graph



Flow Graph


Legend
Internal Call
External Call
Defined Contract
Undefined Contract
ReentrancyGuard
<Constructor>
nonReentrant
_nonReentrantAfter
_nonReentrantBefore
_reentrancyGuardEntered
LiquidityMarketplace
<Constructor>
setOwnerFee
setFeeReceiver
getUserDeals
getUserAuction
initializeDeal
activateDeal
startAuction
checkLiquidityOwner
activateAuction
claimAuctionReward
withdrawBid
makeDeal
immediatelyBuy
payable
repayLoan
makeBid
cancelDeal
claimCollateral
withdrawAuctionLiquidity
claimAuction
UniqueLock
ILiquidityLocker (iface)
transferLockOwnership
getUserLockForTokenAtIndex
Ownable
onlyOwner
_checkOwner
owner
<Constructor>
_msgSender
renounceOwnership
_transferOwnership
transferOwnership
Context
_msgSender
_msgData

Summary

The ØxLiquidity contract implements a comprehensive approach to managing liquidity interactions. This audit investigates security issues, business logic concerns, and potential improvements within the contract's framework.

Disclaimer

About Cyberscope

Cyberscope is a blockchain cybersecurity company that was founded with the vision to make web3.0 a safer place for investors and developers. Since its launch, it has worked with thousands of projects and is estimated to have secured tens of millions of investors' funds.

Cyberscope is one of the leading smart contract audit firms in the crypto space and has built a high-profile network of clients and partners.

The Cyberscope team

https://www.cyberscope.io