Cyberscope

Audit Report

ØxLiquidity

May 2024

SHA256 8c37b02401ab9199f8d24f9d4481ce69b9063472ddabbbad998e73bbc12d8e1f

Audited by © cyberscope

Table of Contents

Review

Testing Deploy	https://testnet.bscscan.com/address/0x60df2c3789e34e65f19303f103b497ae27981700

Audit Updates

Initial Audit	23 May 2024

Source Files

Filename	SHA256
contracts/LiquidityMarketplace.sol	8c37b02401ab9199f8d24f9d4481ce69b9063472ddabbbad998e73bbc12d8e1f

Contract Readability Comment

The audit scope is to identify security vulnerabilities, validate the business logic, and suggest potential optimizations. The contract currently lacks critical checks necessary to secure its transactions effectively. It also fails to adhere to fundamental principles of Solidity smart contract development concerning gas consumption, code readability, and appropriate use of data structures. These shortcomings indicate that the contract is not in a production-ready state. Due to these critical findings, the development team is strongly advised to re-evaluate the contract's business logic and its alignment with Solidity best practices. It is imperative to review and optimize gas usage to minimize costs and enhance efficiency. Additionally, improving code readability by simplifying function definitions and employing descriptive variable names is crucial for better auditability and maintenance. The contract code requires thorough evaluation and extensive testing to address these issues adequately and ensure it is ready for production deployment.

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.

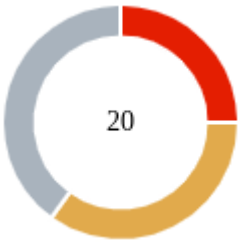
claimAuctionReward Functionality

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



Severity	Unresolved	Acknowledged	Resolved	Other
Critical	5	0	0	0
Medium	7	0	0	0
Minor / Informative	8	0	0	0

Diagnostics

Critical Medium Minor / Informative

Severity	Code	Description	Status
Critical	DEV	Duplicate Entries Vulnerability	Unresolved
Critical	IWG	Inadequate Withdrawal Guard	Unresolved
Critical	LRV	Lender Revert Vulnerability	Unresolved
Critical	MLV	Missing Lender Verification	Unresolved
Critical	MWR	Multiple Withdrawal Risk	Unresolved
Medium	CCC	Contradictory Claim Conditions	Unresolved
Medium	IRM	Interest Rate Manipulation	Unresolved
Medium	MLT	Missing Lock Transfer	Unresolved
Medium	OLLE	Overlooked Liquidity Lock Expiry	Unresolved
Medium	RFI	Repeated Function Invocation	Unresolved
Medium	UBA	Unchecked Bid Amount	Unresolved
Medium	USP	Unused StartPrice Parameter	Unresolved
Minor / Informative	ADS	Arbitrary Duration Setting	Unresolved
Minor / Informative	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

DEV - Duplicate Entries Vulnerability

Criticality	Critical
Location	contracts/LiquidityMarketplace.sol#L289,436
Status	Unresolved

Description

The contract is designed to initialize deals and auctions for managing liquidity tokens. However, it currently lacks safeguards against the creation of identical deals or auctions. This could result in multiple entries with the same parameters being stored in the contract. Such duplication not only wastes storage and complicates transaction histories but also poses functional risks, such as in the event of operations like transferring liquidity tokens, the contract could mistakenly identify duplicate entries as valid and unique, leading to erroneous or unintended transfers. This mismanagement of token ownership can significantly undermine the integrity and intended functionality of the contract.

```
    function initializeDeal(
        address lpToken,
        uint256 lockIndex,
        uint256 dealAmount,
        uint256 interestRate,
        uint256 loanDuration
    ) public {
        require(loanDuration > 0, "Loan duration must be greater than
0");
        require(
            checkLiquidityOwner(msg.sender, lpToken, lockIndex),
            "User does not owner of this lock"
        );
        require(interestRate > 0, "interestRate must be greater than 0");
        require(dealAmount > 0, "dealAmount must be greater than 0");

        deals[nextDealId] = Deal({
            borrower: msg.sender,
            lpToken: lpToken,
            lockIndex: lockIndex,
            dealAmount: dealAmount,
            interestRate: interestRate,
            loanDuration: loanDuration,
            startTime: 0,
            lender: address(0),
            isRepaid: false,
            isActive: false
        });

        emit DealInitialized(
            nextDealId,
            msg.sender,
            lpToken,
            lockIndex,
            dealAmount,
            interestRate,
            loanDuration
        );

        userDeals[msg.sender].push(nextDealId);

        nextDealId++;
    }
    ...
    function startAuction(
        address lpToken,
        uint256 lockIndex,
        uint256 startPrice,
        uint256 imeddiatelySellPrice,
        uint256 bidStep,
```

```
        uint256 duration,
        bool immediatelySell
    ) public {
        require(duration > 0, "Duration must be greater than 0");
        require(
            checkLiquidityOwner(msg.sender, lpToken, lockIndex),
            "User does not owner of this lock"
        );
        require(
            imeddiatelySellPrice > 0,
            "imeddiatelySellPrice must be positive number"
        );

        auctions[nextAuctionId] = Auction({
            owner: msg.sender,
            highestBidOwner: address(0),
            lpToken: lpToken,
            lockIndex: lockIndex,
            startPrice: startPrice,
            imeddiatelySellPrice: imeddiatelySellPrice,
            bidStep: bidStep,
            duration: duration,
            startTime: block.timestamp,
            isActive: false,
            isFinishedImmediately: false,
            immediatelySell: immediatelySell
        });
        emit AuctionStarted(
            nextAuctionId,
            msg.sender,
            lpToken,
            lockIndex,
            startPrice,
            imeddiatelySellPrice,
            bidStep,
            duration,
            immediatelySell
        );

        userAuction[msg.sender].push(nextAuctionId);

        nextAuctionId++;
    }
```

Recommendation

It is recommended to introduce checks that verify the uniqueness of each `Deal` or `Auction` created. Implementing validation mechanisms to detect and prevent the storage of identical entries before finalizing any new deal or auction will ensure that each entry is

unique. This approach will prevent the contract from erroneously processing transactions based on duplicated data, thereby maintaining accurate tracking and handling of liquidity tokens. Such precautions are crucial for ensuring the contract operates efficiently and remains reliable for users engaging in financial transactions.

IWG - Inadequate Withdrawal Guard

Criticality	Critical
Location	contracts/LiquidityMarketplace.sol#L564
Status	Unresolved

Description

The contract includes a `withdrawAuctionLiquidity` function that allows the auction owner to reclaim the liquidity locked in an auction. However, the current guarding conditions within the `require` statement do not adequately protect against premature withdrawals. The existing logic permits the auction owner to withdraw the liquidity if no immediately buy action has occurred (`isFinishedImmediately` is false), regardless of whether bids have been placed. This means that as long as the immediate purchase option has not been executed, the auction owner can reclaim the liquidity even when valid bids exist, which could undermine the integrity of the auction process and the bidder's expectations.

```
    function withdrawAuctionLiquidity(uint256 auctionId) public {
        Auction storage auction = auctions[auctionId];
        require(
            auction.startTime + auction.duration < block.timestamp,
            "Auction is active yet"
        );
        require(
            auction.highestBidOwner == address(0) ||
                !auction.isFinishedImmediately,
            "Not claimable"
        );
        require(msg.sender == auction.owner, "Caller is not auction
owner");

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

Recommendation

It is recommended to amend the condition in the `require` statement to ensure that the auction owner can only withdraw liquidity if no bids have been made (highestBidOwner is address(0)) and the immediately buy function has not been invoked (isFinishedImmediately is false). Changing the condition to an `AND` logic would close the loophole, ensuring that liquidity can only be reclaimed if there are truly no outstanding claims on it—either through bids or an immediate purchase. This modification will strengthen the contract's security and fairness, ensuring that the auction's rules are enforced consistently.

LRV - Lender Revert Vulnerability

Criticality	Critical
Location	contracts/LiquidityMarketplace.sol#L410
Status	Unresolved

Description

The contract sends funds to a `lender` as part of the transfer flow. This address can either be a wallet address or a contract. If the address belongs to a contract then it may revert from incoming payment. As a result, the error will propagate to the token's contract and revert the transfer.

Specifically, the contract is vulnerable to a scenario where the lender's address is a smart contract that can intentionally revert the transaction. If the lender's smart contract is designed to reject the repayment, this would prevent the `repayLoan` function from executing successfully. As a result, borrowers would be unable to repay their loans, leading to potential financial disputes and contract dysfunction.

```
    function repayLoan(uint256 dealId) public payable {
        ...
        (bool sent, ) = payable(deal.lender).call{value:
msg.value}("");
        require(sent, "Repay failed");
```

Recommendation

The contract should tolerate the potential revert from the underlying contracts when the interaction is part of the main transfer flow. This could be achieved by not allowing set contract addresses or by sending the funds in a non-revertable way. Consider implementing mechanisms to ensure that repayments cannot be blocked by the lender. This could include validating the lender address or providing alternative repayment handling methods to ensure the `repayLoan` function remains executable and the contract operates reliably.

MLV - Missing Lender Verification

Criticality	Critical
Location	contracts/LiquidityMarketplace.sol#L386
Status	Unresolved

Description

The contract is intended to manage lending transactions through its functions, but it lacks a crucial verification step in the `repayLoan` function to ensure that a legitimate lender exists before allowing a loan repayment. This missing check allows for a scenario where a borrower could activate a deal after transferring the liquidity pool to the contract, repay the loan (where they would effectively pay nothing since no actual loan was credited due to the absence of a lender), and then potentially engage in further transactions that exploit this oversight. This flaw can lead to manipulation of the loan process, enabling the borrower to retrieve the liquidity pool without any real loan transaction taking place, and after that claim the funds when a makeDeal functionality is called, thereby undermining the integrity of the lending system.

```
    function repayLoan(uint256 dealId) public payable {
        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(msg.value >= repayAmount, "Insuffitient payable
amount");
        require(
            deal.startTime + deal.loanDuration >
block.timestamp,
            "Loan duration exceed"
        );
        deal.isRepaid = true;
        (, , , , uint256 lockId, ) =
liquidityLocker.getUserLockForTokenAtIndex(
            address(this),
            deal.lpToken,
            deal.lockIndex
        );
        liquidityLocker.transferLockOwnership(
            deal.lpToken,
            deal.lockIndex,
            lockId,
            payable(msg.sender)
        );
        (bool sent, ) = payable(deal.lender).call{value:
msg.value}("");
        require(sent, "Repay failed");
        emit LoanRepaid(dealId, msg.sender);
    }
```

Recommendation

It is recommended to introduce a check in the `repayLoan` function to verify that the `makeDeal` function has been executed, confirming the presence of a legitimate `lender` before allowing repayment. This would ensure that the loan repayment process is tied to an actual credited loan, preventing borrowers from exploiting the system to retrieve liquidity pools without proper loan transactions. Implementing this safeguard will strengthen the contract's security and ensure that its lending functionality operates as intended.

MWR - Multiple Withdrawal Risk

Criticality	Critical
Location	contracts/LiquidityMarketplace.sol#L616
Status	Unresolved

Description

The contract's `claimAuctionReward` function allows the auction owner to claim the highest bid amount after an auction concludes. However, the current implementation does not include safeguards to prevent the owner from claiming the bid amount multiple times. This lack of restriction can lead to a scenario where the auction owner repeatedly withdraws the same bid amount, draining funds improperly and potentially exploiting the auction's financial mechanisms. This oversight compromises the security of the auction funds and undermines the fairness and integrity of the auction process.

```
   function claimAuctionReward(uint256 auctionId) external {
        Auction memory auction = auctions[auctionId];
        require(
            !auction.isFinishedImmediately &&
                auction.startTime + auction.duration <
block.timestamp,
            "Auction active yet"
        );
        require(msg.sender == auction.owner, "Not eligible for
claim");
        uint256 bidAmount =
bids[auctionId][auction.highestBidOwner];

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

        (bool sent, ) = payable(msg.sender).call{value:
bidAmount - feeAmount}(
            ""
        );
        require(sent, "Withdraw failed");
        emit AuctionRewardClaimed(
            auctionId,
            msg.sender,
            bidAmount,
            block.timestamp
        );
    }
```

Recommendation

It is recommended to add an additional check within the `claimAuctionReward` function to ensure that the function can only be invoked once per auction. This could be achieved by maintaining a state variable that tracks whether the reward for a particular auction has already been claimed. Once the reward is claimed, this variable should prevent any further attempts to claim the reward again. Implementing this safeguard will secure the auction funds against unauthorized multiple withdrawals and preserve the trust and reliability of the auction system.

CCC - Contradictory Claim Conditions

Criticality	Medium
Location	contracts/LiquidityMarketplace.sol#L590
Status	Unresolved

Description

The contract's `claimAuction` function is intended to enable the highest bid owner to claim the locked liquidity after an auction. However, the function's logic contains contradictory requirements that can lead to confusion and incorrect behavior. The first require statement allows for a claim if the auction is finished immediately (`isFinishedImmediately` is true) or if the auction duration has passed. Yet, the second require checks that the auction has not finished immediately (`isFinishedImmediately` is false) for the claim to proceed. This inconsistency means that while one condition permits claims on immediate finishes, the other explicitly denies them under the same circumstances, potentially preventing rightful claims or enabling inappropriate ones.

```
    function claimAuction(uint256 auctionId) public {
        Auction memory auction = auctions[auctionId];
        require(
            auction.isFinishedImmediately ||
                auction.startTime + auction.duration <
block.timestamp,
            "Auction is active yet"
        );
        require(
            msg.sender == auction.highestBidOwner &&
                !auction.isFinishedImmediately,
            "Not eligible for claim"
        );
        (, , , , uint256 lockId, ) =
liquidityLocker.getUserLockForTokenAtIndex(
            address(this),
            auction.lpToken,
            auction.lockIndex
        );
        liquidityLocker.transferLockOwnership(
            auction.lpToken,
            auction.lockIndex,
            lockId,
            payable(msg.sender)
        );
        emit AuctionWon(auctionId, msg.sender);
    }
```

Recommendation

It is recommended to modify the conditions within the `claimAuction` function to ensure coherence and prevent contradictory checks. The logic should consistently allow the highest bid owner to claim the locked liquidity only when the auction duration has elapsed and the auction has not finished immediately. Removing the initial check for `auction.isFinishedImmediately` from the condition will align the function's operations with the intended auction mechanics, ensuring that claims are only made under appropriate and clear circumstances. This adjustment will enhance the integrity and predictability of the auction outcomes.

IRM - Interest Rate Manipulation

Criticality	Medium
Location	contracts/LiquidityMarketplace.sol#L
Status	Unresolved

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 {
        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(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.

MLT - Missing Lock Transfer

Criticality	Medium
Location	contracts/LiquidityMarketplace.sol#L298,436
Status	Unresolved

Description

The contract is designed to initialize deals and start auctions using locked liquidity tokens. However, it currently does not include the transfer of lock ownership during these initialization phases. This omission can significantly disrupt the flow of transactions, as the actual liquidity lock does not move to secure the contract's control or to the intended other party. Without transferring lock ownership at the time of deal or auction initialization, the integrity and enforceability of the terms could be compromised, potentially halting subsequent operations like bidding, loan repayment, or collateral claims.

```
    function initializeDeal(
        address lpToken,
        uint256 lockIndex,
        uint256 dealAmount,
        uint256 interestRate,
        uint256 loanDuration
    ) public {
        ...
        userDeals[msg.sender].push(nextDealId);

        nextDealId++;
    }
    ...
    function startAuction(
        address lpToken,
        uint256 lockIndex,
        uint256 startPrice,
        uint256 imeddiatelySellPrice,
        uint256 bidStep,
        uint256 duration,
        bool immediatelySell
    ) public {
        ...
        userAuction[msg.sender].push(nextAuctionId);

        nextAuctionId++;
    }
```

Recommendation

It is recommended to integrate the transfer of lock ownership directly within the initialization phases of deals and auctions. This change would ensure that the contract has the appropriate control over the locked assets right from the start, aligning the actual token lock status with the contract's state and the intentions of the users. By securing ownership transfer at the outset, the contract can more reliably manage the locked assets throughout the lifecycle of loans and auctions, thereby preventing discrepancies and enhancing transactional security.

OLLE - Overlooked Liquidity Lock Expiry

Criticality	Medium
Location	contracts/LiquidityMarketplace.sol#L246,372,399,422,522,577,602
Status	Unresolved

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.

RFI - Repeated Function Invocation

Criticality	Medium
Location	contracts/LiquidityMarketplace.sol#L415,590
Status	Unresolved

Description

The contract currently does not implement safeguards to prevent multiple invocations of critical functions such as `claimCollateral` and `claimAuction`. This omission allows users to potentially invoke these functions repeatedly under the same conditions, leading to multiple transfers of lock ownership and repeated event emissions. Since many decentralized applications (dApps) rely on these event emissions to trigger subsequent processes, this can result in unintended behaviors or manipulations within these applications, complicating state management and potentially leading to errors or security vulnerabilities.

```
    function claimCollateral(uint256 dealId) public {
        Deal storage deal = deals[dealId];
        require(deal.lender == msg.sender, "Caller is not
lender");
        require(
            deal.startTime + deal.loanDuration <
block.timestamp,
            "Deal is active yet"
        );
        (, , , , uint256 lockId, ) =
liquidityLocker.getUserLockForTokenAtIndex(
            address(this),
            deal.lpToken,
            deal.lockIndex
        );
        liquidityLocker.transferLockOwnership(
            deal.lpToken,
            deal.lockIndex,
            lockId,
            payable(msg.sender)
        );
        emit CollateralClaimed(dealId, msg.sender);
    }

    function claimAuction(uint256 auctionId) public {
        Auction memory auction = auctions[auctionId];
        require(
            auction.isFinishedImmediately ||
                auction.startTime + auction.duration <
block.timestamp,
            "Auction is active yet"
        );
        require(
            msg.sender == auction.highestBidOwner &&
                !auction.isFinishedImmediately,
            "Not eligible for claim"
        );
        (, , , , uint256 lockId, ) =
liquidityLocker.getUserLockForTokenAtIndex(
            address(this),
            auction.lpToken,
            auction.lockIndex
        );
        liquidityLocker.transferLockOwnership(
            auction.lpToken,
            auction.lockIndex,
            lockId,
            payable(msg.sender)
        );
        emit AuctionWon(auctionId, msg.sender);
```

```
    }
```

Recommendation

It is recommended to restrict the execution of these functions to only necessary instances and to enforce that they can be successfully executed only once when conditions are met. Implementing checks that mark these transactions as completed and prevent re-invocation under the same conditions would ensure that lock ownership is transferred correctly and that event emissions do not occur multiple times for a single logical transaction. This change will help maintain the contract's integrity and reliability, supporting stable and predictable interactions within associated dApps.

UBA - Unchecked Bid Amount

Criticality	Medium
Location	contracts/LiquidityMarketplace.sol#L616
Status	Unresolved

Description

The contract's `claimAuctionReward` function is responsible for allowing auction owners to withdraw the highest bid amount once an auction has concluded. However, it currently lacks a crucial verification to ensure that the bid amount is greater than zero before proceeding with the withdrawal. This omission leaves the function vulnerable to attempts to claim a reward where no valid bids have been placed, or where bid entries might be erroneously zero due to other contract interactions or errors. This gap can lead to unnecessary event emition, wasted gas fees, and potential disruptions in the auction process.

```
   function claimAuctionReward(uint256 auctionId) external {
        Auction memory auction = auctions[auctionId];
        require(
            !auction.isFinishedImmediately &&
                auction.startTime + auction.duration <
block.timestamp,
            "Auction active yet"
        );
        require(msg.sender == auction.owner, "Not eligible for
claim");
        uint256 bidAmount =
bids[auctionId][auction.highestBidOwner];

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

        (bool sent, ) = payable(msg.sender).call{value:
bidAmount - feeAmount}(
            ""
        );
        require(sent, "Withdraw failed");
        emit AuctionRewardClaimed(
            auctionId,
            msg.sender,
            bidAmount,
            block.timestamp
        );
    }
```

Recommendation

It is recommended to add a condition within the `claimAuctionReward` function to check that the bid amount associated with the highest bidder is greater than zero before allowing any fund transfers. This verification will effectively ensure that there is a legitimate, positive bid to claim, which aligns with the intent to only distribute rewards for valid and successfully completed auctions. This additional check will enhance the robustness and reliability of the auction system by preventing futile or erroneous reward claims.

USP - Unused StartPrice Parameter

Criticality	Medium
Location	contracts/LiquidityMarketplace.sol#L460
Status	Unresolved

Description

The contract is designed to facilitate auctions through its `startAuction` function, which includes a `startPrice` variable intended to set the minimum starting bid for each auction. However, despite being declared and set by users, is not actually utilized in any auction processes. This oversight leads to a functionality gap, as the naming and purpose of the `startPrice` suggest it should play a crucial role in setting the auction's minimum starting price. Consequently, users setting this value are misled, as it has no effect on the auction outcome, potentially impacting user trust and the overall utility of the auction mechanism.

```
    function startAuction(
        address lpToken,
        uint256 lockIndex,
        uint256 startPrice,
        uint256 imeddiatelySellPrice,
        uint256 bidStep,
        uint256 duration,
        bool immediatelySell
    ) public {
        ...
        auctions[nextAuctionId] = Auction({
            owner: msg.sender,
            highestBidOwner: address(0),
            lpToken: lpToken,
            lockIndex: lockIndex,
            startPrice: startPrice,
        ...
    }
```

Recommendation

It is recommended to reconsider the usage of the `startPrice` variable within the contract's auction functionality. If the variable is intended to influence auction dynamics, such as setting a minimum starting bid price, it should be integrated appropriately into the auction's bid handling processes. Conversely, if the `startPrice` is deemed unnecessary for the contract's intended operations, it may be prudent to remove this parameter altogether to streamline the contract interface and avoid confusion among users. This adjustment will ensure clarity and functionality alignment within the contract's design.

ADS - Arbitrary Duration Setting

Criticality	Minor / Informative
Location	contracts/LiquidityMarketplace.sol#L289,436
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 {
        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 {
        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#L251
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#L334,346,370,392,553,509
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#L503
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
{
        Auction storage auction = auctions[auctionId];
        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#L251,256,278,284
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.

```
_ownerFee) exter
 _feeReceiver) exter

 _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#L7,33,120
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#L7,33,120
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

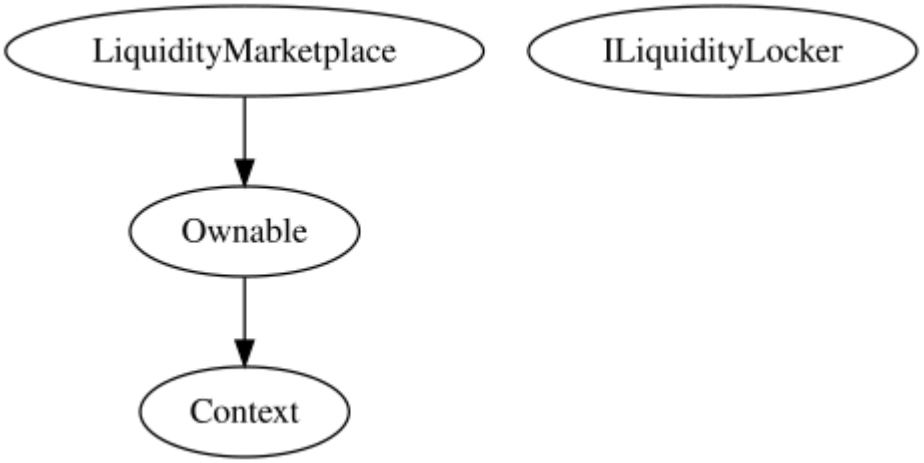
The team is advised to lock the pragma to ensure the stability of the codebase. The locked pragma version ensures that the contract will not be deployed with an unexpected version. An unexpected version may produce vulnerabilities and undiscovered bugs. The compiler should be configured to the lowest 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

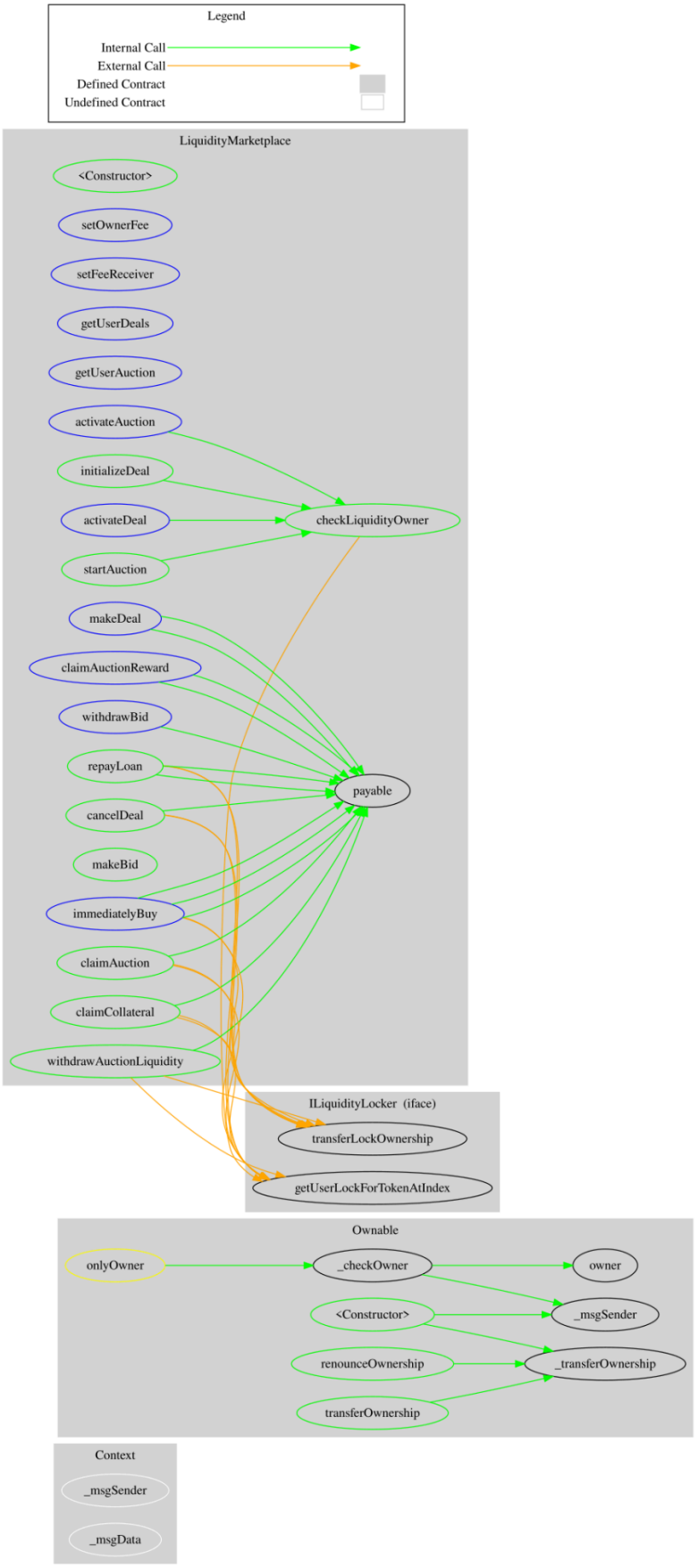
Contract	Type	Bases		
	Function Name	**Visibility**	**Mutability**	**Modifiers**
ILiquidityLocker	Interface			
	transferLockOwnership	External	✓	-
	getUserLockForTokenAtIndex	External		-
LiquidityMarketplace	Implementation	Ownable		
		Public	✓	-
	setOwnerFee	External	✓	onlyOwner
	setFeeReceiver	External	✓	onlyOwner
	checkLiquidityOwner	Public		-
	getUserDeals	External		-
	getUserAuction	External		-
	initializeDeal	Public	✓	-
	activateDeal	External	✓	-
	makeDeal	External	Payable	-
	cancelDeal	Public	✓	-
	repayLoan	Public	Payable	-
	claimCollateral	Public	✓	-
	startAuction	Public	✓	-
	activateAuction	External	✓	-

	immediatelyBuy	External	Payable	-
	makeBid	Public	Payable	-
	withdrawAuctionLiquidity	Public	✓	-
	claimAuction	Public	✓	-
	claimAuctionReward	External	✓	-
	withdrawBid	External	✓	-

Inheritance Graph



Flow Graph


Legend
Internal Call
External Call
Defined Contract
Undefined Contract
LiquidityMarketplace
<Constructor>
setOwnerFee
setFeeReceiver
getUserDeals
getUserAuction
activateAuction
initializeDeal
activateDeal
startAuction
makeDeal
claimAuctionReward
withdrawBid
repayLoan
cancelDeal
makeBid
immediatelyBuy
claimAuction
claimCollateral
withdrawAuctionLiquidity
checkLiquidityOwner
payable
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

The information provided in this report does not constitute investment, financial or trading advice and you should not treat any of the document's content as such. This report may not be transmitted, disclosed, referred to or relied upon by any person for any purposes nor may copies be delivered to any other person other than the Company without Cyberscope’s prior written consent. This report is not nor should be considered an “endorsement” or “disapproval” of any particular project or team. This report is not nor should be regarded as an indication of the economics or value of any “product” or “asset” created by any team or project that contracts Cyberscope to perform a security assessment. This document does not provide any warranty or guarantee regarding the absolute bug-free nature of the technology analyzed, nor do they provide any indication of the technologies proprietors' business, business model or legal compliance. This report should not be used in any way to make decisions around investment or involvement with any particular project. This report represents an extensive assessment process intending to help our customers increase the quality of their code while reducing the high level of risk presented by cryptographic tokens and blockchain technology.

Blockchain technology and cryptographic assets present a high level of ongoing risk Cyberscope’s position is that each company and individual are responsible for their own due diligence and continuous security Cyberscope’s goal is to help reduce the attack vectors and the high level of variance associated with utilizing new and consistently changing technologies and in no way claims any guarantee of security or functionality of the technology we agree to analyze. The assessment services provided by Cyberscope are subject to dependencies and are under continuing development. You agree that your access and/or use including but not limited to any services reports and materials will be at your sole risk on an as-is where-is and as-available basis Cryptographic tokens are emergent technologies and carry with them high levels of technical risk and uncertainty. The assessment reports could include false positives false negatives and other unpredictable results. The services may access and depend upon multiple layers of third parties.

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