Cyberscope

Audit Report

Liquify

November 2024

Repository	https://github.com/CatalinBalut/Liquify/tree/main
Commit	78b5eb19d7a8d3d8d49af31e7540b8fd9a71a205
Audited by	© cyberscope

Table of Contents

Risk Classification

The criticality of findings in Cyberscope's smart contract audits is determined by evaluating multiple variables. The two primary variables are:

1. **Likelihood of Exploitation**: This considers how easily an attack can be executed, including the economic feasibility for an attacker.
2. **Impact of Exploitation**: This assesses the potential consequences of an attack, particularly in terms of the loss of funds or disruption to the contract's functionality.

Based on these variables, findings are categorized into the following severity levels:

1. **Critical**: Indicates a vulnerability that is both highly likely to be exploited and can result in significant fund loss or severe disruption. Immediate action is required to address these issues.
2. **Medium**: Refers to vulnerabilities that are either less likely to be exploited or would have a moderate impact if exploited. These issues should be addressed in due course to ensure overall contract security.
3. **Minor**: Involves vulnerabilities that are unlikely to be exploited and would have a minor impact. These findings should still be considered for resolution to maintain best practices in security.
4. **Informative**: Points out potential improvements or informational notes that do not pose an immediate risk. Addressing these can enhance the overall quality and robustness of the contract.

Severity	Likelihood / Impact of Exploitation
Critical	Highly Likely / High Impact
Medium	Less Likely / High Impact or Highly Likely/ Lower Impact
Minor / Informative	Unlikely / Low to no Impact

Review

Repository	https://github.com/CatalinBalut/Liquify/tree/main
Commit	78b5eb19d7a8d3d8d49af31e7540b8fd9a71a205

Audit Updates

Initial Audit	29 Oct 2024 https://github.com/cyberscope-io/audits/blob/main/lqfy/v1/audit.pdf
Corrected Phase 2	30 Oct 2024 https://github.com/cyberscope-io/audits/blob/main/lqfy/v2/audit.pdf
Corrected Phase 3	04 Nov 2024

Source Files

Filename	SHA256
ReferralManager.sol	9585831487bc60984d4b7b6e8c972da5eeb178efc6073d1279a152e99144ac19
Liquify.sol	aec1043fe5f0ddb41cdc5121030defaecfee532ea8ce2187f10172080f5ce498
LiquidERC20.sol	375a0fcd056471727e43a79be75496818f1ce0e468689586ac0f1c806aef3142
interfaces/IReferralManager.sol	91c61af5b428ae7e6567fa6a597897c428c592a61a3ea7a09588ea05fd4de20b
interfaces/ILiquify.sol	0c923aff793e63b11b7ad52c821ab8661944dd45fb53d7c775cfde9f7b418a19

Overview

Liquify contract

The `Liquify` smart contract is a decentralized platform for liquidity provision that allows projects and investors to manage token allocations through a Liquid Vesting Mechanism. This mechanism ensures that tokens locked under vesting schedules can still provide liquidity and be traded using synthetic liquid tokens before vesting periods are completed. The contract also supports cross-chain functionality, enabling seamless token migration across blockchains. It is designed to allow project creators to initiate and manage funding rounds, token issuance, and vesting, while providing liquidity for locked tokens. The contract also incorporates a referral mechanism to reward users for referrals and ensures proper accounting of project fees.

Owner Functionalities

The contract owner has several critical functionalities. First, the owner can set up the initial whitelisted initiators via the `updateInitiators` function, which defines the Merkle root for valid project creators. The owner can also manage payment tokens using `updatePaymentTokens`, enabling or disabling specific tokens for contributions. Additionally, the owner can withdraw protocol fees accrued from payment tokens using `withdrawProtocolFees`. Finally, the owner can update or change the referral manager contract, which handles referral fees for contributors.

createProject Functionality

The `createProject` function allows whitelisted initiators (project creators) to create new projects with predefined funding and vesting parameters. This function verifies that the initiator is whitelisted and checks that all required project details are valid, such as token price, allocation, investment caps, and vesting percentages. The project is then assigned a unique ID and initialized with Liquid ERC20 tokens representing the project's vesting stages. Each stage has its own token allocation that reflects the vesting release schedule, ensuring that tokens are distributed gradually over time as per the project's parameters.

initiatorWithdraw Functionality

This function allows project creators to withdraw funds raised during a project. However, it verifies that the project's status is not set to “Refund” before allowing the withdrawal. The function calculates the available funds after accounting for already withdrawn amounts and any fees, ensuring that the project initiator can only withdraw funds within the available balance. The initiator then receives the requested funds in the payment token used for contributions.

initiatorDeposit Functionality

The `initiatorDeposit` function allows project initiators to deposit real tokens into the contract for distribution to participants as part of the vesting schedule. The function calculates the deposit required for each vesting stage based on the raised amount and the vesting percentage. These real tokens are distributed to investors in line with their synthetic token entitlements for each vesting round.

updateWhitelist Functionality

Project creators can update the whitelist for their projects using the `updateWhitelist` function. This function allows them to either update the Merkle root for address-based whitelisting or set an NFT contract that will be used for ERC-721 or ERC-1155 token-based whitelisting. The project's whitelist determines which users are eligible to contribute, and once updated, the project can be opened for contributions.

setRefundStatus Functionality

The `setRefundStatus` function enables the project initiator to set the project's status to “Refund.” This status allows contributors to request a refund if the project fails or if refunds become necessary. Once the refund status is set, the project will no longer allow new contributions or token claims, and contributors can begin withdrawing their contributions.

contribute Functionality

The `contribute` function enables users to participate in a project by contributing funds in exchange for synthetic tokens. Users can only contribute if the project is open and the funding deadline has not passed. The function also verifies the user's eligibility based on

the project's whitelist settings (if enabled) and enforces the minimum and maximum contribution limits. Contributions are subject to protocol and initiator fees, and referral fees may also apply if the user enters a referral chain.

claimSyntheticTokens Functionality

Once users have contributed to a project, they can claim their synthetic tokens for specific vesting stages using the `claimSyntheticTokens` function. This function checks the user's entitlements for the requested vesting stages and transfers the corresponding synthetic tokens. Synthetic tokens can be traded or used before the vesting period is completed, providing liquidity to participants.

claimRealTokens Functionality

The `claimRealTokens` function allows users to claim the real tokens underlying their synthetic tokens once the vesting stage has been reached. The function burns the synthetic tokens held by the user and transfers the equivalent amount of real tokens, ensuring that the synthetic tokens are converted into the actual project tokens once vesting has been unlocked.

claimRefund Functionality

Users can request a refund for their contribution if a project is in the refund status. The `claimRefund` function calculates the refund amount based on the user's synthetic token balance and any real tokens owed. It ensures that the total refund does not exceed the available refund pool and transfers the refund amount to the user in the project's payment token.

burnTokensForMigration Functionality

The contract provides a cross-chain token migration mechanism through the `burnTokensForMigration` function. Users can burn their synthetic tokens on one chain to facilitate token migration to another chain. This function ensures that the synthetic tokens are burned, and it records the burn event so that the user can receive equivalent tokens on the target chain.

ReferralManager contract

The `ReferralManager` contract is a smart contract designed to manage and facilitate a two-tier referral system for the Liquify platform. It tracks both referrer and super-referrer relationships, calculates referral fees, and allows referrers to withdraw accumulated referral balances in various ERC-20 tokens. Using basis points (BPS) for fee calculations, it ensures referral relationships are established between project initiators and referrers, supporting standard referrers and super-referrers. Only the authorized `Liquify` contract can execute specific referral functions, ensuring controlled access.

Constructor and State Variables

The `ReferralManager` constructor initializes the contract by setting the referral fee percentages for referrers and super-referrers in BPS. It assigns an owner with the authority to manage key referral parameters and sets the `Liquify` contract as the sole authorized contract to execute certain referral functions. The main state variables include referral fee percentages (`referrerFeeBPS` and `superReferrerFeeBPS`), mappings of initiators to their respective referrers, and the balance of referral fees accumulated by each referrer in different tokens. The `referrerBalances` mapping stores referral balances for each referrer across multiple ERC-20 tokens, with the fee calculations capped by `MAX_REFERRER_BPS` to maintain control over maximum referrer fees.

withdrawReferrerBalance Functionality

The `withdrawReferrerBalance` function allows referrers to withdraw their accumulated referral fees in specified ERC-20 tokens. It checks the balance for each token specified and ensures a non-zero balance exists for the caller before proceeding. The balance is then denormalized based on the token's decimals and transferred to the referrer. Each successful withdrawal emits a `ReferrerWithdrawn` event. If no balance is found for a specific token, a `NoReferralBalance` error is triggered.

processReferralFees Functionality

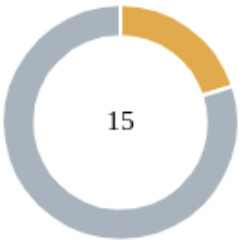
The `processReferralFees` function is central to the referral system. It calculates and allocates referral fees based on a transaction amount and set referral percentages. This function supports both regular referrers and super-referrers, adjusting balances for each as applicable. It emits a `ReferrerBalancesUpdated` event to track balance changes.

The `_processReferrer` function, called within `processReferralFees`, performs calculations for both referrer and super-referrer fees, with safeguards to prevent the cumulative fee from exceeding the defined `MAX_REFERRER_BPS` limit. This controlled fee process prevents unintended compounding and aligns with the protocol's intended limits, ensuring referrer balances reflect accurate, capped amounts.

LiquidERC20 contract

The `LiquidERC20` token contract is a specialized ERC-20 token used within the Liquify platform to represent liquid tokens that correspond to future vested tokens. It allows for the creation of tokens with a fixed total supply during the contract's initialization, which are minted to the contract deployer. The contract provides functionality for the owner to burn tokens, enabling flexible management of the token supply over time. The token plays a crucial role in the platform's liquid vesting mechanism, facilitating liquidity while tokens are still locked under vesting schedules. Additionally, it enforces access control by ensuring that only the contract owner can perform token burns.

Findings Breakdown



Critical	0
Medium	3
Minor / Informative	12

Severity	Unresolved	Acknowledged	Resolved	Other
Critical	0	0	0	0
Medium	1	2	0	0
Minor / Informative	0	12	0	0

Diagnostics

Critical Medium Minor / Informative

Severity	Code	Description	Status
Medium	ERF	Excessive Referrer Fees	Unresolved
Medium	LIL	Loop Iteration Limit	Acknowledged
Medium	MTL	Migration Token Loss	Acknowledged
Minor / Informative	CCR	Contract Centralization Risk	Acknowledged
Minor / Informative	DDP	Decimal Division Precision	Acknowledged
Minor / Informative	ITCC	Inconsistent Token Claim Conditions	Acknowledged
Minor / Informative	ITDP	Inefficient Token Deposit Process	Acknowledged
Minor / Informative	MPC	Merkle Proof Centralization	Acknowledged
Minor / Informative	MEE	Missing Events Emission	Acknowledged
Minor / Informative	MRVM	Missing RealToken Validation Mechanism	Acknowledged
Minor / Informative	PTAI	Potential Transfer Amount Inconsistency	Acknowledged
Minor / Informative	TSI	Tokens Sufficiency Insurance	Acknowledged
Minor / Informative	TRR	Trust-Based Refund Risk	Acknowledged

●	WBPC	Withdraw Before Project Completion	Acknowledged
●	L04	Conformance to Solidity Naming Conventions	Acknowledged

ERF - Excessive Referrer Fees

Criticality	Medium
Location	Liquify.sol#L455 ReferralManager.sol#L60,110,138,169
Status	Unresolved

Description

The contract lacks a comprehensive validation to ensure that cumulative referral fees remain within the intended `protocolFee` limit, not only in the `initialize` function but also in the `setReferrerFeeBps` and `setSuperReferrerFeeBps` functions. Although these setter functions check against a maximum allowed percentage (`MAX_REFERRER_BPS`), they do not account for the fact that `processReferralFees` can call `_processReferrer` twice, resulting in a compounded referral fee that may exceed `protocolFee` . This oversight can cause the combined `referrersFees` to accumulate up to 10%, doubling the expected limit and risking underflow errors during deductions within the `contribute` function.

```
   function contribute(uint256 projectId, uint256 _amount, bytes32[]
calldata merkleProof, uint256 tokenId, address userReferrer, address
superReferrer) external whenNotPaused nonReentrant
whenProjectNotPaused(projectId) {

        ...

        uint256 protocolFee = amount * PROTOCOL_FEE_BPS / MAX_BPS;
        uint256 initiatorFee = amount * project.ownerFeePercent /
MAX_BPS;

        uint256 referrersFees;
        if(userReferrer != address(0) ||
referralManager.getInitiatorReferrer(project.projectOwner) !=
address(0)){
            referrersFees =
referralManager.processReferralFees(msg.sender, project.projectOwner,
userReferrer, superReferrer, project.paymentToken, amount);
        }

        uint256 amountAfterFee = amount - protocolFee - initiatorFee;

        if (state.raisedAmount + amountAfterFee > project.allocation)
revert InvestmentExceedsAllocation();

        protocolFees[project.paymentToken] += protocolFee -
referrersFees;
```

```
    function initialize(uint16 _referrerFeeBPS, uint16
_superReferrerFeeBPS, address gnosisWallet) external initializer {
        __Ownable_init(gnosisWallet);
        __UUPSUpgradeable_init();
        __Pausable_init();

        referrerFeeBPS = _referrerFeeBPS;
        superReferrerFeeBPS = _superReferrerFeeBPS;
    }

    function setReferrerFeeBps(uint16 newReferrerFeeBps) external
onlyOwner whenNotPaused withinAllowedBps(newReferrerFeeBps){
        if (newReferrerFeeBps < superReferrerFeeBPS || newReferrerFeeBps
+ superReferrerFeeBPS > MAX_REFERRER_BPS) {
            revert InvalidReferrerFeeBPS(newReferrerFeeBps,
superReferrerFeeBPS);
        }
        referrerFeeBPS = newReferrerFeeBps;
        emit ReferrerFeeBpsUpdated(newReferrerFeeBps);
    }

    function setSuperReferrerFeeBps(uint16 newSuperReferrerFeeBps)
external onlyOwner whenNotPaused
withinAllowedBps(newSuperReferrerFeeBps) {
        if (newSuperReferrerFeeBps > referrerFeeBPS ||
newSuperReferrerFeeBps + referrerFeeBPS > MAX_REFERRER_BPS) {
            revert InvalidSuperReferrerFeeBPS(newSuperReferrerFeeBps,
referrerFeeBPS);
        }
        superReferrerFeeBPS = newSuperReferrerFeeBps;
        emit SuperReferrerFeeBpsUpdated(newSuperReferrerFeeBps);
    }
     function processReferralFees(
        address sender,
        address initiator,
        address userReferrer,
        address superReferrer,
        IERC20 paymentToken,
        uint256 amount
    ) external whenNotPaused onlyLiquify returns (uint256 totalFee) {
        Referrers memory referrers = initiatorToReferrers[initiator];

        if (referrers.referrer != address(0)) {
            totalFee += _processReferrer( referrers.referrer,
referrers.superReferrer, paymentToken, amount);
        }

        if (userReferrer != address(0)) {
            totalFee += _processReferrer( userReferrer, superReferrer,
paymentToken, amount);
```

```
        }

        emit ReferrerBalancesUpdated(
            sender,
            initiator,
            address(paymentToken),
            referrerBalances[referrers.superReferrer][paymentToken],
            referrerBalances[referrers.referrer][paymentToken],
            referrerBalances[superReferrer][paymentToken],
            referrerBalances[userReferrer][paymentToken]
        );

        return totalFee;
    }

    function _processReferrer(
        address referrer,
        address superReferrer,
        IERC20 paymentToken,
        uint256 amount
    ) internal returns (uint256) {

        uint256 superReferrerFee;
        if (superReferrer != address(0)) {
            superReferrerFee = amount * superReferrerFeeBPS / MAX_BPS;
            referrerBalances[superReferrer][paymentToken] +=
superReferrerFee;
        }

        uint256 referrerFee = amount * referrerFeeBPS / MAX_BPS -
superReferrerFee;
        referrerBalances[referrer][paymentToken] += referrerFee;

        return superReferrerFee + referrerFee;
    }
```

Recommendation

To maintain the protocol's fee structure, implement a validation mechanism in both `initialize` and setter functions (`setReferrerFeeBps` and `setSuperReferrerFeeBps`) to ensure that cumulative `referrersFees` cannot exceed `protocolFee` . A practical approach is to set a maximum cap of 2.5% for each referrer type, ensuring the total remains at or below the 5% protocol limit when called twice. Otherwise, adjust `_processReferrer` to prevent compounding of fees beyond the

protocol’s intended bounds, preserving accurate and predictable fee deductions throughout the contract’s workflow.

LIL - Loop Iteration Limit

Criticality	Medium
Location	Liquify.sol#L511
Status	Acknowledged

Description

The contract is at risk of to incomplete processing of vesting stages due to the use of a `uint8` variable as the loop counter when iterating over the `vestingReleasePercentages` array. If the `project.vestingReleasePercentages.length` exceeds 255, the loop will only execute up to 255 iterations. As a result, any vesting stages beyond the 255th element will not be processed, leading to an incorrect distribution of tokens during vesting. This can create a situation where users do not receive the full amount of their entitled tokens according to the intended vesting schedule.

```
for (uint8 i = 0; i < project.vestingReleasePercentages.length; i++) {
    ...
}
```

Recommendation

It is recommended to update the loop counter from `uint8` to `uint256` to ensure that the loop can iterate over all elements in the `vestingReleasePercentages` array, regardless of its length. This change will prevent the incomplete processing of vesting stages and ensure that all specified percentages are accounted for when calculating token entitlements. Additionally, it may be beneficial to validate the array length before processing to ensure optimal performance and gas efficiency.

Team Update

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

Given that a project having 255 vesting stages is unrealistic and outside the scope of practical usage, the current is not a vulnerability that needs to be addressed.

MTL - Migration Token Loss

Criticality	Medium
Location	ReferralManager.sol#L661
Status	Acknowledged

Description

The contract contains the `burnTokensForMigration` function, intended to facilitate cross-chain transfers by burning tokens. However, it does not provide a clear purpose for users to specify `walletAmounts` during the burn process, as users do not receive the burned amount back nor is it allocated elsewhere. This results in an irreversible loss of tokens with no value or benefit to the user, leading to potential dissatisfaction and financial harm.

```
    function burnTokensForMigration(
        uint256 projectId,
        uint256[] calldata rounds,
        uint256[] calldata walletAmounts,
        uint256[] calldata scAmounts,
        bytes calldata targetAddress
    ) external whenNotPaused nonReentrant
    validateArrayLengths(rounds.length, walletAmounts.length)
    validateArrayLengths(rounds.length, scAmounts.length){
        ProjectState storage state = projectStates[projectId];

        for (uint256 i = 0; i < rounds.length; i++) {
            uint256 round = rounds[i];

            ...

            // Burn the tokens and update the entitlements
            cloneToken.burnFrom(msg.sender, walletAmounts[i]);
            state.tokenEntitlements[msg.sender][round].entitlements -=
scAmounts[i];

state.tokenEntitlements[msg.sender][round].burnedEntitlements +=
scAmounts[i];

            ...
        }
    }
```

Recommendation

It is recommended to revise the `burnTokensForMigration` function to provide a clear rationale for burning `walletAmounts` or to ensure that users receive a corresponding benefit or allocation. If the token burn is essential, clear utility should be provided which explains the reason for the burn and its impact.

Team Update

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

This concern is mitigated through our frontend management approach, which guides users correctly. We assume that users will not interact directly with the smart contract for these operations, thus reducing the risk and maintaining a simple experience for initiators and contract owners.

CCR - Contract Centralization Risk

Criticality	Minor / Informative
Location	Liquify.sol#L157,202,333,352,431
Status	Acknowledged

Description

The contract's functionality and behavior are heavily dependent on external parameters or configurations. While external configuration can offer flexibility, it also poses several centralization risks that warrant attention. Centralization risks arising from the dependence on external configuration include Single Point of Control, Vulnerability to Attacks, Operational Delays, Trust Dependencies, and Decentralization Erosion.

The contract centralizes substantial authority in the hands of the owner, allowing them to manage key functionalities such as updating the initiators whitelist, managing payment tokens, setting the referral manager, and controlling operational states via the `pause` and `unpause` functions. Additionally, the owner can update the `LiquidERC20` token implementation using `setLiquidERC20Implementation`, ensuring control over token behavior. Whitelisted users are allowed to create projects, while project owners handle crusial tasks like managing project status and pausing or unpausing their projects. Importantly, users can only initiate refunds once the project owner has set the status to "Refund" through `setRefundStatus` and provided the necessary funds.

```
    function updateInitiators(bytes32 _initiatorsWhitelist) external
onlyOwner whenNotPaused {
        initiatorsWhitelist = _initiatorsWhitelist;
        emit InitiatorsUpdated(_initiatorsWhitelist);
    }

    function updatePaymentTokens(IERC20[] calldata tokens, bool[]
calldata states) external onlyOwner whenNotPaused
validateArrayLengths(tokens.length, states.length) {

        for (uint256 i = 0; i < tokens.length; i++) {
            paymentTokens[tokens[i]] = states[i];
        }
        emit PaymentTokensUpdated(tokens, states);
    }

    function withdrawProtocolFees(IERC20[] calldata tokens) external
onlyOwner whenNotPaused nonReentrant {
        for (uint256 i = 0; i < tokens.length; i++) {
            ...
        }
    }

    function setReferralManager(IReferralManager _referralManager)
external whenNotPaused onlyOwner {
        referralManager = _referralManager;
    }

    function pause() external onlyOwner {
        _pause();
    }

    function unpause() external onlyOwner {
        _unpause();
    }

    function setLiquidERC20Implementation(ILiquidERC20
newImplementation) external onlyOwner {
        LIQUIDERC20_IMPLEMENTATION = newImplementation;
    }

   function createProject(ILiquify.ProjectDetails calldata newProject,
bytes32[] calldata merkleProof) external whenNotPaused {
    ...
    }

    function updateWhitelist(uint256 projectId, bytes32 newRoot, address
nftContract) external whenNotPaused onlyProjectOwner(projectId) {
        ProjectDetails storage project = projectDetails[projectId];
        ProjectState storage state = projectStates[projectId];
```

```
        ...

        state.status = ProjectStatus.Open;
        emit WhitelistUpdated(projectId, newRoot, nftContract);
    }

    function setRefundStatus(uint256 projectId) external whenNotPaused
onlyProjectOwner(projectId) {
        ...
        state.status = ProjectStatus.Refund;
        emit RefundInitiated(projectId, address(this));
    }

    function pauseProject(uint256 projectId) external whenNotPaused
onlyProjectOwner(projectId){
        if (projectStates[projectId].paused) revert
ProjectAlreadyPaused();

        projectStates[projectId].paused = true;
        emit ProjectPaused(projectId);
    }

    function unpauseProject(uint256 projectId) external whenNotPaused
onlyProjectOwner(projectId){
        if (!projectStates[projectId].paused) revert ProjectNotPaused();

        projectStates[projectId].paused = false;
        emit ProjectUnpaused(projectId);
    }

    function setWaitingStatus(uint256 projectId) external whenNotPaused
onlyProjectOwner(projectId){
        projectStates[projectId].status = ProjectStatus.Waiting;
        emit ProjectWaiting(projectId);
    }
```

Recommendation

To address this finding and mitigate centralization risks, it is recommended to evaluate the feasibility of migrating critical configurations and functionality into the contract's codebase itself. This approach would reduce external dependencies and enhance the contract's self-sufficiency. It is essential to carefully weigh the trade-offs between external configuration flexibility and the risks associated with centralization.

Team Update

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

Mitigated with the use of Gnosis Multisig Wallet.

DDP - Decimal Division Precision

Criticality	Minor / Informative
Location	Liquify.sol#L255,313
Status	Acknowledged

Description

Division of decimal (fixed point) numbers can result in rounding errors due to the way that division is implemented in Solidity. Thus, it may produce issues with precise calculations with decimal numbers.

Solidity represents decimal numbers as integers, with the decimal point implied by the number of decimal places specified in the type (e.g. decimal with 18 decimal places). When a division is performed with decimal numbers, the result is also represented as an integer, with the decimal point implied by the number of decimal places in the type. This can lead to rounding errors, as the result may not be able to be accurately represented as an integer with the specified number of decimal places.

Hence, the splitted shares will not have the exact precision and some funds may not be calculated as expected.

```
LiquidERC20(clone).initialize(string.concat(newProject.name, " TGE"),
string.concat(newProject.symbol, "TGE"), totalSupply *
newProject.vestingReleasePercentages[0] / MAX_BPS);

for (uint256 i = 1; i < newProject.vestingReleasePercentages.length;
i++) {
    salt = keccak256(abi.encodePacked(globalIndex, i));
    clone = Clones.cloneDeterministic(LIQUIDERC20_IMPLEMENTATION, salt);
    LiquidERC20(clone).initialize(string.concat(newProject.name,
string.concat(" ", Strings.toString(i))),
    string.concat(newProject.symbol, Strings.toString(i)),
    totalSupply * newProject.vestingReleasePercentages[i] / MAX_BPS);
}
...
uint256 vestingPercentage =
project.vestingReleasePercentages[vestingPhaseIndex];
uint256 expectedDeposit = (projectState.raisedAmount *
vestingPercentage) / MAX_BPS;
uint256 calculatedDepositAmount = (expectedDeposit * 10**18) /
project.pricePerToken;
uint8 realTokenDecimals =
IERC20Metadata(address(sc_realToken)).decimals();
```

Recommendation

The team is advised to take into consideration the rounding results that are produced from the solidity calculations. The contract could calculate the subtraction of the divided funds in the last calculation in order to avoid the division rounding issue.

Team Update

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

Will not fix, as the minor loss is not significant.

ITCC - Inconsistent Token Claim Conditions

Criticality	Minor / Informative
Location	Liquify.sol#L533,553
Status	Acknowledged

Description

The contract requires specific project states, such as `Distributing` or `Finished`, in order to proceed with claiming real tokens via the `claimRealTokens` function. However, the `claimSyntheticTokens` function lacks a similar status check, allowing users to claim their synthetic tokens immediately after contributing. This inconsistency in the claim process can lead to confusion and create an imbalance between the timing of synthetic and real token claims, as users can access synthetic tokens without any project status restrictions, while real tokens are subject to state checks.

```
   function claimSyntheticTokens(uint256 projectId, uint256[] calldata
vestingStages) external whenNotPaused nonReentrant {
        if (vestingStages.length == 0) revert
NoVestingStagesSpecified();
        ...
            IERC20(tokenClone).safeTransfer(msg.sender, tokensToClaim);

projectState.tokenEntitlements[msg.sender][stage].entitlements = 0;
            totalClaimed += tokensToClaim;

            emit SyntheticTokensClaimed(msg.sender, projectId,
address(this), stage, tokensToClaim);
        ...
    }

    function claimRealTokens(uint256 projectId, uint256[] calldata
rounds) external whenNotPaused nonReentrant {
        ProjectState storage state = projectStates[projectId];
        if (state.status != ProjectStatus.Distributing && state.status
!= ProjectStatus.Finished)
            revert InvalidStatus2(ProjectStatus.Distributing,
ProjectStatus.Finished, state.status);

        ...
        if (totalRealTokens > 0) {
            uint8 realTokenDecimals =
IERC20Metadata(address(state.realTokensAddress)).decimals();
            uint256 denormalizedTotalRealTokens =
denormalizeTokenAmount(totalRealTokens, realTokenDecimals);

            state.realTokensAddress.transfer(msg.sender,
denormalizedTotalRealTokens);
        }
        else revert NoTokensToClaim();
    }
```

Recommendation

It is recommended to consider to implement a similar status check in the `claimSyntheticTokens` function to ensure consistency with the `claimRealTokens` process. This would align both functions and ensure that synthetic token claims are only allowed when appropriate project milestones or statuses are reached, preventing premature claims and maintaining the integrity of the token distribution process.

Team Update

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

The claimSyntheticTokens function is designed to operate without status restrictions because synthetic tokens are intended to be available and tradable immediately upon contribution. This approach allows for an open market of proof-of-investment tokens that can be freely exchanged, ensuring that users have liquidity options during the vesting period. The design is intentional and integral to the platform's functionality, providing value to users by enabling early liquidity opportunities.

ITDP - Inefficient Token Deposit Process

Criticality	Minor / Informative
Location	Liquify.sol#L298
Status	Acknowledged

Description

The contract is designed with the `initiatorDeposit` function, allowing the project owner to fund the `realTokens` for each vesting phase. However, once the deposit process starts and one phase is funded, the project owner is still required to fund the remaining phases individually, invoking the function multiple times, once for each phase, until all phases are completed. This approach is inefficient and adds unnecessary complexity, as the process remains incomplete until every vesting phase has been manually funded, increasing the operational burden on the project owner.

```
   function initiatorDeposit(uint256 projectId, IERC20 realToken)
external whenNotPaused nonReentrant onlyProjectOwner(projectId) {
        ProjectDetails storage project = projectDetails[projectId];
        ProjectState storage projectState = projectStates[projectId];
        IERC20 sc_realToken = projectState.realTokensAddress;
        uint256 vestingPhaseIndex = projectState.vestingPhaseIndex;
        ...
        if (vestingPhaseIndex ==
project.vestingReleasePercentages.length - 1) projectState.status =
ProjectStatus.Finished;
        else projectState.vestingPhaseIndex++;

        uint256 balanceBefore = sc_realToken.balanceOf(address(this));
        sc_realToken.safeTransferFrom(msg.sender, address(this),
denormalizedExpectedDeposit);
        //explanation: normalizeTokenAmount(received = balanceAfter -
balanceBefore, realTokenDecimals)
        uint256 normalizedAmountReceived =
normalizeTokenAmount(sc_realToken.balanceOf(address(this)) -
balanceBefore, realTokenDecimals);
        ...
```

Recommendation

It is recommended to optimize the `initiatorDeposit` function by allowing the project owner to deposit the total amount for all vesting phases in a single transaction. The contract can then utilize a loop to distribute the deposited tokens across all vesting phases according to the `vestingReleasePercentages`. This would streamline the process, reduce transaction overhead, and ensure that all phases are funded in one action, making the system more efficient and user-friendly.

Team Update

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

The existing initiatorDeposit function is designed based on the practical realities of how project owners receive and manage their tokens. It is not possible for project owners to deposit all real tokens for every vesting phase at once, as they depend on receiving these tokens incrementally from the project itself. Therefore, the function's current design ensures flexibility, security, and alignment with real-world project operations.

MPC - Merkle Proof Centralization

Criticality	Minor / Informative
Location	Liquify.sol#L333
Status	Acknowledged

Description

The contract uses a Merkle Proof mechanism in order to define many applicable addresses. The verification process is based on an off-chain configuration. The contract owner is responsible for updating the in-chain “Merkle Root” in order to validate correctly the provided message.

```
    function updateWhitelist(uint256 projectId, bytes32 newRoot, address
nftContract) external whenNotPaused onlyProjectOwner(projectId) {
        ProjectDetails storage project = projectDetails[projectId];
        ProjectState storage state = projectStates[projectId];

        if (msg.sender != project.projectOwner) revert
UnauthorizedAccess();
        if (state.status != ProjectStatus.Waiting) revert
InvalidStatus(ProjectStatus.Waiting, state.status);

        if (project.whitelistType == WhitelistType.AddressWhitelist) {
            state.whitelistRoot = newRoot;
        } else if (project.whitelistType ==
WhitelistType.Erc721Whitelist || project.whitelistType ==
WhitelistType.ERC1155Whitelist) {
            state.nftWhitelistContract = nftContract;
        } else if (project.whitelistType ==
WhitelistType.AddressAndErc721Whitelist || project.whitelistType ==
WhitelistType.AddressAndERC1155Whitelist) {
            state.whitelistRoot = newRoot;
            state.nftWhitelistContract = nftContract;
        }

        state.status = ProjectStatus.Open;
        emit WhitelistUpdated(projectId, newRoot, nftContract);
    }
```

Recommendation

We state that the Merkle Proof algorithm is required for proper protocol operations and gas consumption decrease. Thus, we emphasize that the Merkle proof algorithm is based on an off-chain mechanism. Any off-chain mechanism could potentially be compromised and affect the on-chain state unexpectedly. The team should carefully manage the private keys of the owner's account. We strongly recommend a powerful security mechanism that will prevent a single user from accessing the contract admin functions.

Temporary Solutions:

These measurements do not decrease the severity of the finding

- Introduce a time-locker mechanism with a reasonable delay.
- Introduce a multi-signature wallet so that many addresses will confirm the action.
- Introduce a governance model where users will vote about the actions.

Permanent Solution:

- Renouncing the ownership, which will eliminate the threats but it is non-reversible.

Team Update

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

Addressed using Gnosis Multisig. The ownership cannot be renounced.

MEE - Missing Events Emission

Criticality	Minor / Informative
Location	Liquify.sol#L182
Status	Acknowledged

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 setReferralManager(IReferralManager _referralManager)
external whenNotPaused onlyOwner {
        referralManager = _referralManager;
    }
```

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.

Team Update

The team has acknowledged that this is not a security issue.

MRVM - Missing RealToken Validation Mechanism

Criticality	Minor / Informative
Location	Liquify.sol#L162,298
Status	Acknowledged

Description

The contract contains the `updatePaymentTokens` function, which grants the owner the authority to define legitimate payment tokens for the contract. However, it lacks a similar function to set and validate the legitimate `realTokens` used during the `initiatorDeposit`. As a result, the project owner is able to specify any address as the `realToken` address during the deposit process, potentially allowing them to use incorrect or malicious token addresses, which could harm the integrity of the contract.

```
   function updatePaymentTokens(IERC20[] calldata tokens, bool[]
calldata states) external onlyOwner whenNotPaused
validateArrayLengths(tokens.length, states.length) {
        for (uint256 i = 0; i < tokens.length; i++) {
            paymentTokens[tokens[i]] = states[i];
        }
        emit PaymentTokensUpdated(tokens, states);
    }


    function initiatorDeposit(uint256 projectId, IERC20 realToken)
external whenNotPaused nonReentrant onlyProjectOwner(projectId) {
        ProjectDetails storage project = projectDetails[projectId];
        ProjectState storage projectState = projectStates[projectId];
        IERC20 sc_realToken = projectState.realTokensAddress;
        uint256 vestingPhaseIndex = projectState.vestingPhaseIndex;


        ...
        if (vestingPhaseIndex ==
project.vestingReleasePercentages.length - 1) projectState.status =
ProjectStatus.Finished;
        else projectState.vestingPhaseIndex++;
        ...
    }
```

Recommendation

It is recommended to introduce a function that allows the owner to set and validate legitimate `realTokens` in a similar manner to the `updatePaymentTokens` function. This will ensure that only pre-approved `realTokens` can be used during the `initiatorDeposit` process, providing additional security and preventing the use of unauthorized tokens in the contract.

Team Update

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

The responsibility for providing the correct realToken rests with the initiator, as established through signed legal contracts. These agreements ensure that initiators are accountable for bringing the appropriate tokens to the platform. The platform's design, therefore, aligns with standard business practices, leveraging legal assurance rather than technical validation mechanisms to maintain contract integrity.

PTAI - Potential Transfer Amount Inconsistency

Criticality	Minor / Informative
Location	Liquify.sol#L426,527
Status	Acknowledged

Description

The `safeTransferFrom` function is used to transfer a specified amount of tokens to an address. The fee or tax is an amount that is charged to the sender of an ERC20 token when tokens are transferred to another address. According to the specification, the transferred amount could potentially be less than the expected amount. This may produce inconsistency between the expected and the actual behavior.

The following example depicts the diversion between the expected and actual amount.

Tax	Amount	Expected	Actual
No Tax	100	100	100
10% Tax	100	100	90

The contract currently lacks a validation check to ensure the full expected amount is transferred, which may lead to inconsistencies if the `paymentToken` has associated fees or taxes. Specifically, when tokens with fees are transferred, the actual amount received by the contract will be less than the original specified amount.

```
paymentToken.safeTransferFrom(msg.sender, address(this),
denormalizedTotalToRefund);
...
project.paymentToken.safeTransferFrom(msg.sender, address(this), _amount
- denormalizedReferralFees);
project.paymentToken.safeTransferFrom(msg.sender,
address(referralManager), denormalizedReferralFees);
```

Recommendation

The team is advised to take into consideration the actual amount that has been transferred instead of the expected.

It is important to note that an ERC20 transfer tax is not a standard feature of the ERC20 specification, and it is not universally implemented by all ERC20 contracts. Therefore, the contract could produce the actual amount by calculating the difference between the transfer call.

```
Actual Transferred Amount = Balance After Transfer - Balance Before Transfer
```

Team Update

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

The paymentTokens will be usdc/usdt, so no tokens with fees will be used.

TSI - Tokens Sufficiency Insurance

Criticality	Minor / Informative
Location	Liquify.sol#L298
Status	Acknowledged

Description

The tokens are not held within the contract itself. Instead, the contract is designed to provide the tokens from an external administrator. While external administration can provide flexibility, it introduces a dependency on the administrator's actions, which can lead to various issues and centralization risks.

Specificallym the project owner have the authority to invoke the `initiatorDeposit` function in order to supply the `realTokensAddress` tokens to the contract.

```
   function initiatorDeposit(uint256 projectId, IERC20 realToken)
external whenNotPaused nonReentrant onlyProjectOwner(projectId) {
        ProjectDetails storage project = projectDetails[projectId];
        ProjectState storage projectState = projectStates[projectId];
        IERC20 sc_realToken = projectState.realTokensAddress;
        uint256 vestingPhaseIndex = projectState.vestingPhaseIndex;


        ...
        if (vestingPhaseIndex ==
project.vestingReleasePercentages.length - 1) projectState.status =
ProjectStatus.Finished;
        else projectState.vestingPhaseIndex++;
        ...
    }
```

Recommendation

It is recommended to consider implementing a more decentralized and automated approach for handling the contract tokens. One possible solution is to hold the tokens within the contract itself. If the contract guarantees the process it can enhance its reliability, security, and participant trust, ultimately leading to a more successful and efficient process.

Team Update

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

The responsibility for providing the correct realToken rests with the initiator, as established through signed legal contracts. These agreements ensure that initiators are accountable for bringing the appropriate tokens to the platform. The platform's design, therefore, aligns with standard business practices, leveraging legal assurance rather than technical validation mechanisms to maintain contract integrity.

TRR - Trust-Based Refund Risk

Criticality	Minor / Informative
Location	Liquify.sol#L270,352,404
Status	Acknowledged

Description

The contract contains the `refund` function, which allows the project owner to initiate refunds for tokens from the remaining phases that were not funded through the `contribute` process. These remaining tokens represent contributions made by users who funded the project in order to participate in future phases that are now unfunded. The contract relies on the project owner to first withdraw all remaining funds using `initiatorWithdraw` and then transfer those funds back to the contract for refunds. This creates a potential scenario where, if the project owner fails to return the refund funds, users will not be able to claim their rightful refunds, leading to a loss of trust and potential financial harm to participants.

```
   function initiatorWithdraw(uint256 projectId, uint256 amount)
external whenNotPaused nonReentrant onlyProjectOwner(projectId) {
        ProjectDetails storage project = projectDetails[projectId];
        ProjectState storage projectState = projectStates[projectId];
        ...
        paymentToken.safeTransfer(msg.sender, amount);

        emit FundsWithdrawn(projectId, amount);
    }

    function setRefundStatus(uint256 projectId) external whenNotPaused
onlyProjectOwner(projectId) {
        ProjectState storage state = projectStates[projectId];
        ProjectDetails storage project = projectDetails[projectId];
        ...
            amountToRefund = (state.raisedAmount + state.raisedFees) *
remainingPercentage / MAX_BPS;
        }

        uint256 deservedAmount = state.raisedAmount + state.raisedFees -
amountToRefund;

    ...
    }

    function refund(uint256 projectId, uint256 amount) external
whenNotPaused nonReentrant onlyProjectOwner(projectId){
        ...
        uint256 denormalizedTotalToRefund =
denormalizeTokenAmount(amountToRefund, paymentTokenDecimals);
        paymentToken.safeTransferFrom(msg.sender, address(this),
denormalizedTotalToRefund);

        emit RefundDeposit(projectId, amount);
    }
```

Recommendation

It is recommended to implement an automated mechanism that ensures refunds are processed directly from the contract, without relying on the project owner to transfer the funds back. This would remove the need for trust in the project owner to return the refund amounts, ensuring that users can claim their refunds regardless of the owner's actions.

Additionally, consider enforcing stricter controls on fund management to safeguard user contributions.

Team Update

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

The responsibility for providing the correct realToken rests with the initiator, as established through signed legal contracts. These agreements ensure that initiators are accountable for bringing the appropriate tokens to the platform. The platform's design, therefore, aligns with standard business practices, leveraging legal assurance rather than technical validation mechanisms to maintain contract integrity.

WBPC - Withdraw Before Project Completion

Criticality	Minor / Informative
Location	Liquify.sol#L270
Status	Acknowledged

Description

The contract allows the project owner to withdraw funds through the `initiatorWithdraw` function without verifying that the project has reached the `Finished` status. This omission could lead to premature withdrawals while the project is still ongoing, potentially disrupting the intended process and leaving insufficient funds for future project phases or user claims.

```
    function initiatorWithdraw(uint256 projectId, uint256 amount)
external whenNotPaused nonReentrant onlyProjectOwner(projectId) {
        ProjectDetails storage project = projectDetails[projectId];
        ProjectState storage projectState = projectStates[projectId];
        ...
        paymentToken.safeTransfer(msg.sender, amount);

        emit FundsWithdrawn(projectId, amount);
    }
```

Recommendation

It is recommended to include a check in the `initiatorWithdraw` function that ensures it can only be executed when the project status is set to `Finished`. This will ensure that withdrawals occur only after the project has been fully completed, safeguarding the correct process flow and maintaining sufficient funds throughout the project's lifecycle.

Team Update

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

The platform is correctly designed to allow initiators to access necessary funds before full project completion, ensuring flexibility and supporting project development.

L04 - Conformance to Solidity Naming Conventions

Criticality	Minor / Informative
Location	ReferralManager.sol#L60,91,96 Liquify.sol#L77,111,157,182,455
Status	Acknowledged

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.

```
uint16 _superReferrerFeeBPS
uint16 _referrerFeeBPS
address _liquifyContract
ILiquidERC20 public LIQUIDERC20_IMPLEMENTATION;
IReferralManager _referralManager
bytes32 _initiatorsWhitelist
IReferralManager _referralManager
uint256 _amount
```

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/stable/style-guide.html#naming-conventions.

Team Update

The team has acknowledged that this is not a security issue.

Functions Analysis

Contract	Type	Bases		
	Function Name	**Visibility**	**Mutability**	**Modifiers**
ReferralManager	Implementation	IReferralManager, OwnableUpgradeable, UUPSUpgradeable, PausableUpgradeable		
		Public	✓	-
	initialize	External	✓	initializer
	addLiquifyContract	External	✓	onlyOwner whenNotPaused
	removeLiquifyContract	External	✓	onlyOwner whenNotPaused
	setReferrers	External	✓	onlyOwner whenNotPaused
	setReferrerFeeBps	External	✓	onlyOwner whenNotPaused withinAllowedBps
	setSuperReferrerFeeBps	External	✓	onlyOwner whenNotPaused withinAllowedBps
	pause	External	✓	onlyOwner
	unpause	External	✓	onlyOwner
	processReferralFees	External	✓	whenNotPaused onlyLiquify
	_processReferrer	Internal	✓	
	withdrawReferrerBalance	External	✓	whenNotPaused
	denormalizeTokenAmount	Public		-
	getInitiatorReferrer	External		-

	getInitiatorSuperReferrer	External		-
	_authorizeUpgrade	Internal	✓	onlyOwner
Liquify	Implementation	ILiquify, OwnableUpgradeable, ReentrancyGuardUpgradeable, UUPSUpgradeable, PausableUpgradeable		
		Public	✓	-
	initialize	Public	✓	initializer
	updateInitiators	External	✓	onlyOwner whenNotPaused
	updatePaymentTokens	External	✓	onlyOwner whenNotPaused validateArrayLengths
	withdrawProtocolFees	External	✓	onlyOwner whenNotPaused nonReentrant
	setReferralManager	External	✓	whenNotPaused onlyOwner
	pause	External	✓	onlyOwner
	unpause	External	✓	onlyOwner
	setLiquidERC20Implementation	External	✓	onlyOwner
	createProject	External	✓	whenNotPaused
	initiatorWithdraw	External	✓	whenNotPaused nonReentrant onlyProjectOwner
	initiatorDeposit	External	✓	whenNotPaused nonReentrant onlyProjectOwner
	updateWhitelist	External	✓	whenNotPaused onlyProjectOwner

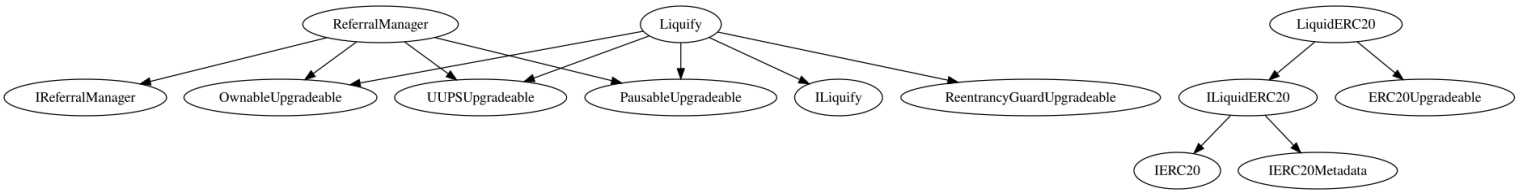
	setRefundStatus	External	✓	whenNotPaused onlyProjectOwner
	refund	External	✓	whenNotPaused nonReentrant onlyProjectOwner
	pauseProject	External	✓	whenNotPaused onlyProjectOwner
	unpauseProject	External	✓	whenNotPaused onlyProjectOwner
	setWaitingStatus	External	✓	whenNotPaused onlyProjectOwner
	contribute	External	✓	whenNotPaused nonReentrant whenProjectNotPaused
	claimSyntheticTokens	External	✓	whenNotPaused nonReentrant
	claimRealTokens	External	✓	whenNotPaused nonReentrant
	claimRefund	External	✓	whenNotPaused nonReentrant
	batchSetWaiting	External	✓	whenNotPaused
	burnTokensForMigration	External	✓	whenNotPaused nonReentrant validateArrayLengths validateArrayLengths
	batchApprove	External	✓	whenNotPaused nonReentrant validateArrayLengths
	batchReduceApproval	External	✓	whenNotPaused nonReentrant validateArrayLengths
	batchTransfer	External	✓	whenNotPaused nonReentrant validateArrayLengths validateAddress
	batchTransferFrom	External	✓	whenNotPaused nonReentrant validateArrayLengt

				hs validateAddress validateAddress
	_transferTokens	Internal	✓	
	getClone	External		-
	getProjectReleasePercentage	External		-
	getProjectReleaseLength	External		-
	getProjectVestingPercentage	External		-
	getInvestment	External		-
	getProjectDetails	External		-
	getTokenEntitlement	External		-
	getBurnedEntitlements	External		-
	getApprovedAmount	External		-
	getRaisedFees	External		-
	getRaisedAmount	External		-
	getInitiatorWithdrawableBalance	External		-
	normalizeTokenAmount	Public		-
	denormalizeTokenAmount	Public		-
	_authorizeUpgrade	Internal	✓	onlyOwner
LiquidERC20	Implementation	ILiquidERC20, ERC20Upgradeable		
		Public	✓	-
	initialize	Public	✓	initializer
	burnFrom	External	✓	-

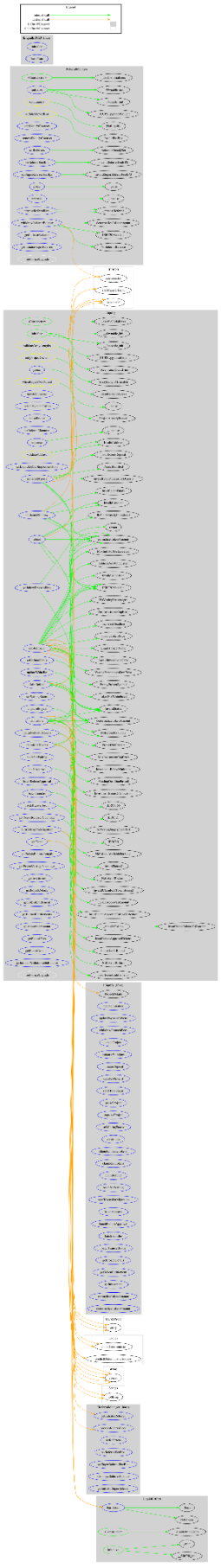
IReferralManager	Interface			
	setReferrers	External	✓	-
	setReferrerFeeBps	External	✓	-
	setSuperReferrerFeeBps	External	✓	-
	withdrawReferrerBalance	External	✓	-
	processReferralFees	External	✓	-
	getInitiatorReferrer	External		-
	getInitiatorSuperReferrer	External		-
ILiquify	Interface			
	updateInitiators	External	✓	-
	updatePaymentTokens	External	✓	-
	withdrawProtocolFees	External	✓	-
	createProject	External	✓	-
	initiatorWithdraw	External	✓	-
	initiatorDeposit	External	✓	-
	updateWhitelist	External	✓	-
	setRefundStatus	External	✓	-
	pauseProject	External	✓	-
	unpauseProject	External	✓	-
	setWaitingStatus	External	✓	-
	contribute	External	✓	-

	claimSyntheticTokens	External	✓	-
	claimRealTokens	External	✓	-
	claimRefund	External	✓	-
	batchSetWaiting	External	✓	-
	burnTokensForMigration	External	✓	-
	batchApprove	External	✓	-
	batchReduceApproval	External	✓	-
	batchTransfer	External	✓	-
	batchTransferFrom	External	✓	-
	getProjectDetails	External		-
	getTokenEntitlement	External		-
	getInvestment	External		-
	normalizeTokenAmount	External		-
	denormalizeTokenAmount	External		-
ILiquidERC20	Interface	IERC20, IERC20Metadata		
	initialize	External	✓	-
	burnFrom	External	✓	-

Inheritance Graph



Flow Graph



Summary

The Liquify contract implements a decentralized liquidity provision mechanism through its Liquid Vesting Mechanism, enabling token allocations to provide liquidity and be traded as synthetic tokens before vesting periods are complete. This audit investigates security vulnerabilities, business logic concerns, and potential improvements in the contract's functionalities, including token vesting, project and referral management, and the handling of synthetic and real token interactions to ensure robustness and efficiency.

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

cyberscope.io