Cyberscope

Audit Report

Liquify

October 2024

Repository	https://github.com/CatalinBalut/Liquify/tree/main
Commit	fcdc8bf01a2f5b45d0ae9e35f623ddd939b3a8ce
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	fcdc8bf01a2f5b45d0ae9e35f623ddd939b3a8ce

Audit Updates

Initial Audit	29 Oct 2024

Source Files

Filename	SHA256
ReferralManager.sol	897bf3029521252c5fc9a9997006a01a596855d89488b7a7ac5d105a13973bd9
Liquify.sol	2ca0b960c5cee3b9d2a9c548f8257a9c79ea545cab07243ad96c6fa271fdf77b
LiquidERC20.sol	3e9981109057ce3503e6bf84c631903be00a0299dbefa1bfb75d6983e76b62b6
utils/MockERC20.sol	72387bd778de39aa6f302a8ff51235af0aadf2fc5a8f58567f1cd6cee5eaef53
interfaces/IReferralManager.sol	f82120422deb25ed0527d5ae331f47c8947eeec9f2bac45d558e5c4dc2219161
interfaces/ILiquify.sol	24a668cfb86753eea690e98d869455fde0e8b5753457ea8cb82c0f492f8739f8

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 handles referrer and super-referrer relationships, processes referral fees, and allows referrers to withdraw their earned referral fees in various ERC-20 tokens. The contract operates using basis points (BPS) for calculating referral fees and ensures that referral relationships are established between project initiators and referrers, supporting both standard referrers and super-referrers.

Constructor and State Variables

The `ReferralManager` constructor initializes the contract by setting the referral fee percentages for referrers and super-referrers in BPS. It also assigns an owner to the contract, who has the authority to manage key aspects of the referral system. The main state variables include the referral fee percentages (`referrerFeeBPS` and `superReferrerFeeBPS`), the mapping of initiators to their referrers, and the balance of referral fees accumulated by each referrer in different tokens. These balances are tracked by the `referrerBalances` mapping, which stores the referral balances for each referrer across multiple ERC-20 tokens.

withdrawReferrerBalance Functionality

The `withdrawReferrerBalance` function allows referrers to withdraw their accumulated referral fees in specified ERC-20 tokens. It checks the balance of each specified token for the caller and ensures that the caller has a non-zero balance to withdraw. The contract transfers the appropriate amount of tokens to the referrer and emits a `ReferrerWithdrawn` event for each successful withdrawal. If a referrer has no balance in a specific token, the function throws a `NoReferralBalance` error.

processReferralFees Functionality

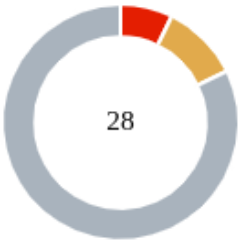
The `processReferralFees` function is central to the referral system. It calculates and distributes referral fees for a transaction initiated by a sender to a project owned by an initiator. The function checks if the project initiator has associated referrers and super-referrers, then calculates the appropriate fees based on the transaction amount and the set referral fee percentages. It allocates fees to the referrer and super-referrer and

updates their token balances in the `referrerBalances` mapping. This function emits a `ReferrerBalancesUpdated` event to track the changes in referral balances.

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	2
Medium	3
Minor / Informative	23

Severity	Unresolved	Acknowledged	Resolved	Other
Critical	2	0	0	0
Medium	3	0	0	0
Minor / Informative	23	0	0	0

Diagnostics

Critical Medium Minor / Informative

Severity	Code	Description	Status
●	IPFC	Inaccurate Protocol Fees Calculation	Unresolved
●	MAC	Missing Access Control	Unresolved
●	IFT	Inaccurate Fee Tracking	Unresolved
●	LIL	Loop Iteration Limit	Unresolved
●	MTL	Migration Token Loss	Unresolved
●	CR	Code Repetition	Unresolved
●	CCR	Contract Centralization Risk	Unresolved
●	DDP	Decimal Division Precision	Unresolved
●	IFC	Inconsistent Fee Calculations	Unresolved
●	ITCC	Inconsistent Token Claim Conditions	Unresolved
●	ITDP	Inefficient Token Deposit Process	Unresolved
●	MPC	Merkle Proof Centralization	Unresolved
●	MC	Missing Check	Unresolved

●	MEM	Missing Error Messaging	Unresolved
●	MEE	Missing Events Emission	Unresolved
●	MRVM	Missing RealToken Validation Mechanism	Unresolved
●	MWBM	Missing Withdrawable Balance Method	Unresolved
●	MU	Modifiers Usage	Unresolved
●	PBV	Percentage Boundaries Validation	Unresolved
●	PSU	Potential Subtraction Underflow	Unresolved
●	PTAI	Potential Transfer Amount Inconsistency	Unresolved
●	TSI	Tokens Sufficiency Insurance	Unresolved
●	TRR	Trust-Based Refund Risk	Unresolved
●	WBPC	Withdraw Before Project Completion	Unresolved
●	L04	Conformance to Solidity Naming Conventions	Unresolved
●	L14	Uninitialized Variables in Local Scope	Unresolved
●	L19	Stable Compiler Version	Unresolved
●	L20	Succeeded Transfer Chec	Unresolved

IPFC - Inaccurate Protocol Fees Calculation

Criticality	Critical
Location	Liquify.sol#L132,225,395,403
Status	Unresolved

Description

The contract is inconsistently normalizing and denormalizing amounts when calculating protocol fees and referrer fees. Specifically, in the `contribute` function, the `_amount` parameter is normalized based on the `paymentToken` decimals, and the `protocolFee` is correctly calculated using this normalized value. However, the `referrersFees` are calculated by denormalizing the value back to the `paymentToken` decimals, causing the referrer fees to reflect the `paymentToken` decimals instead of the normalized amount. This results in a mismatch where the `protocolFee` is incremented using a normalized amount, while the referrers' fees use a denormalized amount. Consequently, this inconsistency leads to inaccurate protocol fee values, and during the withdrawal of protocol fees, the withdrawn amount will be incorrect due to the improper calculation of referrer fees.

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

            uint256 denormalizedAmount = denormalizeTokenAmount(amount,
IERC20Metadata(address(tokens[i])).decimals());

            tokens[i].safeTransfer(owner(), denormalizedAmount);
            emit ProtocolFeesWithdrawn(tokens[i], denormalizedAmount);
        }
    }

 function contribute(uint256 projectId, uint256 _amount, bytes32[]
calldata merkleProof, uint256 tokenId, address userReferrer, address
superReferrer) external nonReentrant whenNotPaused(projectId) {
        ProjectDetails storage project = projectDetails[projectId];
        ProjectState storage state = projectStates[projectId];

        uint256 amount = normalizeTokenAmount(_amount,
IERC20Metadata(address(project.paymentToken)).decimals());
        ...

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

        uint256 referrersFees =  denormalizeTokenAmount(
            referralManager.processReferralFees(msg.sender,
project.projectOwner, userReferrer, superReferrer, project.paymentToken,
amount),
            IERC20Metadata(address(project.paymentToken)).decimals() );

        ...
        }
```

Recommendation

It is recommended to recheck the denormalization functionality of the contract, particularly when calculating the referrer fees. The contract should avoid denormalizing the referrer fees and instead use the normalized amount to increment the `protocolFees`. This will ensure consistency between the protocol and referrer fee calculations and prevent inaccuracies in the withdrawal process.

MAC - Missing Access Control

Criticality	Critical
Location	ReferralManager.sol#L132
Status	Unresolved

Description

The contract is missing access control checks for the `processReferralFees` function, which allows any user to call this function with arbitrary parameters. Without these checks, malicious users could exploit the function by passing their own addresses and get allocation for all the token funds within the contract, resuling in draining the contracts funds. This vulnerability poses a significant risk as it permits unauthorized users to manipulate the referral fee system, resulting in a complete depletion of the contract's token balance.

```
 function processReferralFees(
        address sender,
        address initiator,
        address userReferrer,
        address superReferrer,
        IERC20 paymentToken,
        uint256 amount
    ) external returns (uint256 fee) {
        Referrers memory referrers = initiatorToReferrers[initiator];
        ...
        if (referrers.referrer != address(0)) {
            if (referrers.superReferrer != address(0)) {
                superReferrerFee1 = (amount * superReferrerFeeBPS) /
MAX_BPS;
                referrerBalances[referrers.superReferrer][paymentToken]
+= superReferrerFee1;
            }
            referrerFee = (amount * referrerFeeBPS) / MAX_BPS -
superReferrerFee1;
            referrerBalances[referrers.referrer][paymentToken] +=
referrerFee;
        }

        if (userReferrer != address(0)) {
            if (superReferrer != address(0)) {
                superReferrerFee2 = amount * superReferrerFeeBPS /
MAX_BPS;
                referrerBalances[superReferrer][paymentToken] +=
superReferrerFee2;
            }
            userReferralFee = amount * referrerFeeBPS / MAX_BPS -
superReferrerFee2;
            referrerBalances[userReferrer][paymentToken] +=
userReferralFee;
        }

        ...

        return superReferrerFee1 + superReferrerFee2 + referrerFee +
userReferralFee;
    }
```

Recommendation

It is recommended to implement strict access control mechanisms to ensure that only authorized entities, such as the contract owner or designated operators, such as the Liquify contract, can invoke the `processReferralFees` function. This will prevent

unauthorized users from exploiting the contract’s referral fee logic and protect the token funds from being drained by malicious actors.

IFT - Inaccurate Fee Tracking

Criticality	Medium
Location	Liquify.sol#L218
Status	Unresolved

Description

The contract inaccurately tracks the `totalFeesWithdrawn` variable. In the `initiatorWithdraw` function, the contract adds the total `raisedFees` to the `totalFeesWithdrawn` without considering any previously withdrawn fees. As a result, the variable does not correctly reflect the remaining fees after each withdrawal, which can lead to inaccurate reporting and potential over-withdrawals.

```
   function initiatorWithdraw(uint256 projectId, uint256 amount)
external {
        ProjectDetails storage project = projectDetails[projectId];
        ProjectState storage projectState = projectStates[projectId];

        ...
        if (amount <= 0) revert InvalidAmount();
        uint256 normalizedAmount = normalizeTokenAmount(amount,
IERC20Metadata(address(project.paymentToken)).decimals());
        uint256 availableAmount = projectState.raisedAmount +
projectState.raisedFees - projectState.totalAmountWithdrawn;
        if (normalizedAmount > availableAmount) revert
InsufficientFunds();

        projectState.totalAmountWithdrawn += normalizedAmount;
        projectState.totalFeesWithdrawn += projectState.raisedFees;

        ...
    }
```

Recommendation

It is recommended to update the logic for tracking `totalFeesWithdrawn` by deducting the previously claimed amounts and only adding the portion of fees that are being withdrawn in each transaction. This will ensure accurate tracking of the remaining

withdrawable fees and prevent any discrepancies between the actual fees available and those reported by the contract.

LIL - Loop Iteration Limit

Criticality	Medium
Location	Liquify.sol#L410
Status	Unresolved

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.

```
uint256[] memory entitlements = new
uint256[](project.vestingReleasePercentages.length);
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.

MTL - Migration Token Loss

Criticality	Medium
Location	ReferralManager.sol#L560
Status	Unresolved

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 nonReentrant {
        ProjectState storage state = projectStates[projectId];

        if (rounds.length != walletAmounts.length || rounds.length !=
scAmounts.length) {
            revert ArrayLengthMismatch();
        }

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

            if (totalBurnAmount == 0) {
                revert NoTokensToBurn(round);
            }

            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.

CR - Code Repetition

Criticality	Minor / Informative
Location	Liquify.sol#L661,696 ReferralManager.sol#L132
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.

The `batchTransfer` and `batchTransferFrom` functions share similar code segments for transferring tokens between users and emitting transfer events. This duplication increases the code's complexity and introduces the potential for inconsistent logic or maintenance issues if changes need to be made in both functions separately.

Additionally, the `processReferralFees` function contains similar code segments.

```
 function batchTransfer(
        uint256 projectId,
        uint256[] calldata stages,
        address recipient,
        uint256[] calldata amounts
    ) external nonReentrant {
        ProjectState storage state = projectStates[projectId];
        ...

            emit TokensTransferred(projectId, msg.sender, recipient,
stage, amount);
        }
    }

    function batchTransferFrom(
        uint256 projectId,
        uint256[] calldata stages,
        address owner,
        address recipient,
        uint256[] calldata amounts
    ) external nonReentrant {
        ProjectState storage state = projectStates[projectId];
        if (stages.length != amounts.length) {
            revert ArrayLengthMismatch();
        }

        if (owner == address(0) || recipient == address(0)) {
            revert InvalidAddress();
        }

        for (uint256 i = 0; i < stages.length; i++) {
            ...

            emit TokensTransferred(projectId, owner, recipient, stage,
amount);
        }
    }
```

```
    function processReferralFees(
        ...
    ) external returns (uint256 fee) {
        ...
        if (referrers.referrer != address(0)) {
            if (referrers.superReferrer != address(0)) {
                superReferrerFee1 = (amount * superReferrerFeeBPS) /
MAX_BPS;
                referrerBalances[referrers.superReferrer][paymentToken]
+= superReferrerFee1;
            }
            referrerFee = (amount * referrerFeeBPS) / MAX_BPS -
superReferrerFee1;
            referrerBalances[referrers.referrer][paymentToken] +=
referrerFee;
        }

        if (userReferrer != address(0)) {
            if (superReferrer != address(0)) {
                superReferrerFee2 = amount * superReferrerFeeBPS /
MAX_BPS;
                referrerBalances[superReferrer][paymentToken] +=
superReferrerFee2;
            }
            userReferralFee = amount * referrerFeeBPS / MAX_BPS -
superReferrerFee2;
            referrerBalances[userReferrer][paymentToken] +=
userReferralFee;
        }
        ...
    }
```

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.

It is recommended to refactor the shared logic between these two functions into an internal `_transfer` function. This internal function can handle the core transfer logic, allowing both `batchTransfer` and `batchTransferFrom` to call this function. This

approach will reduce redundancy, improve code maintainability, and ensure consistency in token transfer logic across the contract.

CCR - Contract Centralization Risk

Criticality	Minor / Informative
Location	Liquify.sol#L118,123,144,266,286,333
Status	Unresolved

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 owner holds significant authority, including updating the initiators' whitelist, managing payment tokens and setting the referral manager. Whitelisted users have the ability to create projects, while project owners are responsible for handling and administering crucial functionalities such as managing the project's status and operations, and and pausing or unpausing projects. Additionally, users can only initiate a refund after the project owner has set the project's status to "Refund" using the setRefundStatus function, and provide the funds for the refund.

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

     function updatePaymentTokens(IERC20[] calldata tokens, bool[]
calldata states) external onlyOwner {
        ...
            paymentTokens[tokens[i]] = states[i];
        ...
    }

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

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

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

       ...

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

    function pauseProject(uint256 projectId) external override {
        if (msg.sender != projectDetails[projectId].projectOwner) revert
UnauthorizedAccess();
        if (projectStates[projectId].paused) revert
ProjectAlreadyPaused();

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

    function unpauseProject(uint256 projectId) external {
        if (msg.sender != projectDetails[projectId].projectOwner) revert
UnauthorizedAccess();
        if (!projectStates[projectId].paused) revert ProjectNotPaused();

        projectStates[projectId].paused = false;
```

```
        emit ProjectUnpaused(projectId, address(this));
    }

    function setWaitingStatus(uint256 projectId) external {
        if (msg.sender != projectDetails[projectId].projectOwner) revert
UnauthorizedAccess();

        projectStates[projectId].status = ProjectStatus.Waiting;
        emit ProjectWaiting(projectId, address(this));
    }

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

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.

DDP - Decimal Division Precision

Criticality	Minor / Informative
Location	Liquify.sol#L211,244
Status	Unresolved

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[projectState.vestingPhaseIndex];
```

```
uint256 expectedDeposit = (projectState.raisedAmount *
vestingPercentage) / MAX_BPS;
```

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.

IFC - Inconsistent Fee Calculations

Criticality	Minor / Informative
Location	ReferralManager.sol#L83,93
Status	Unresolved

Description

The contract calculates both the `superReferrerFee` and the `referrerFee` based on predefined percentages. However, the contract deducts the `superReferrerFee` from the `referrerFee`, implying that the `referrerFee` must always be greater than or equal to the `superReferrerFee`. Despite this dependency, the contract lacks proper checks in the `setReferrerFeeBps` and `setSuperReferrerFeeBps` functions to enforce this relationship when setting these values. As a result, it is possible to set values where the `superReferrerFee` exceeds the `referrerFee`, leading to incorrect fee calculations and potential imbalances in payment distributions.

```
   function setReferrerFeeBps(uint16 newReferrerFeeBps) external
onlyOwner {
        referrerFeeBPS = newReferrerFeeBps;
        emit ReferrerFeeBpsUpdated(newReferrerFeeBps);
    }


    function setSuperReferrerFeeBps(uint16 newSuperReferrerFeeBps)
external onlyOwner {
        superReferrerFeeBPS = newSuperReferrerFeeBps;
        emit ReferrerFeeBpsUpdated(newSuperReferrerFeeBps);
    }

 function processReferralFees(
        address sender,
        address initiator,
        address userReferrer,
        address superReferrer,
        IERC20 paymentToken,
        uint256 amount
    ) external returns (uint256 fee) {
        ...
        if (referrers.referrer != address(0)) {
            if (referrers.superReferrer != address(0)) {
                superReferrerFee1 = (amount * superReferrerFeeBPS) /
MAX_BPS;
                referrerBalances[referrers.superReferrer][paymentToken]
+= superReferrerFee1;
            }
            referrerFee = (amount * referrerFeeBPS) / MAX_BPS -
superReferrerFee1;
            referrerBalances[referrers.referrer][paymentToken] +=
referrerFee;
        }

        if (userReferrer != address(0)) {
            if (superReferrer != address(0)) {
                superReferrerFee2 = amount * superReferrerFeeBPS /
MAX_BPS;
                referrerBalances[superReferrer][paymentToken] +=
superReferrerFee2;
            }
            userReferralFee = amount * referrerFeeBPS / MAX_BPS -
superReferrerFee2;
            referrerBalances[userReferrer][paymentToken] +=
userReferralFee;
        }
        ...
        }
```

Recommendation

It is recommended to introduce validation checks within the `setReferrerFeeBps` and `setSuperReferrerFeeBps` functions to ensure that the `referrerFee` is always greater than or equal to the `superReferrerFee`. These checks will prevent any misconfiguration and ensure the correctness of fee calculations, preserving the intended relationship between the two fees.

ITCC - Inconsistent Token Claim Conditions

Criticality	Minor / Informative
Location	Liquify.sol#L431,451
Status	Unresolved

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 {
        if (vestingStages.length == 0) revert
NoVestingStagesSpecified();

        ...
            IERC20(tokenClone).transfer(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 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.

ITDP - Inefficient Token Deposit Process

Criticality	Minor / Informative
Location	Liquify.sol#L238
Status	Unresolved

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 {
        ProjectDetails storage project = projectDetails[projectId];
        ProjectState storage projectState = projectStates[projectId];

        if (msg.sender != project.projectOwner) revert
UnauthorizedAccess();

        uint256 vestingPercentage =
project.vestingReleasePercentages[projectState.vestingPhaseIndex];
        ...

        projectState.vestingPhaseIndex++;
        projectState.realTokensAddress.safeTransferFrom(msg.sender,
address(this), denormalizedExpectedDeposit);
        emit RealTokensDeposited(projectId, denormalizedExpectedDeposit,
address(this));
    }
```

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.

MPC - Merkle Proof Centralization

Criticality	Minor / Informative
Location	Liquify.sol#L266
Status	Unresolved

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 {
        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.

MC - Missing Check

Criticality	Minor / Informative
Location	ReferralManager.sol#L152
Status	Unresolved

Description

The contract is processing variables that have not been properly sanitized and checked that they form the proper shape. These variables may produce vulnerability issues.

Specifically, the following checks and validations are missing:

- The contract does not verify that `projectOwner` is not the zero address.
- The contract does not verify that `minInvestmentCap` is less than `maxInvestmentCap`.

These omissions could lead to unintended behavior or security vulnerabilities if improper or malicious data is passed to the function.

```
   function createProject(ILiquify.ProjectDetails calldata newProject,
bytes32[] calldata merkleProof) external {
        if (!MerkleProof.verify(merkleProof, initiatorsWhitelist,
keccak256(abi.encodePacked(msg.sender)))) revert
AddressNotWhitelisted();
        if (bytes(newProject.name).length == 0) revert
EmptyProjectName();
        if (bytes(newProject.symbol).length == 0) revert
EmptyProjectSymbol();
        if (newProject.fundingDeadline <= block.timestamp) revert
IncorrectDeadline();
        if (newProject.maxInvestmentCap == 0) revert
ZeroInvestmentCapError();
        if (newProject.vestingReleasePercentages.length == 0) revert
NoVestingPercentages();
        if (newProject.pricePerToken == 0) revert InvalidTokenPrice();
        if (newProject.allocation == 0) revert InvalidAllocation();
        if (!paymentTokens[newProject.paymentToken]) revert
TokenNotWhitelisted();
        if (newProject.ownerFeePercent > MAX_BPS - PROTOCOL_FEE_BPS)
revert InvalidFeePercent();
        ...
        }
```

Recommendation

The team is advised to properly check the variables according to the required specifications.

MEM - Missing Error Messaging

Criticality	Minor / Informative
Location	Liquify.sol#L560
Status	Unresolved

Description

The contract lacks a proper check that provides a clear and informative error message when the `scAmount` exceeds the user's entitlements. As a result, when `scAmount` is greater than the available balance, the transaction fails due to an underflow, but the failure does not offer a helpful explanation to the user. This can lead to confusion, as users may not understand why their transaction reverted, negatively affecting the user experience.

```
function burnTokensForMigration(
        uint256 projectId,
        uint256[] calldata rounds,
        uint256[] calldata walletAmounts,
        uint256[] calldata scAmounts,
        bytes calldata targetAddress
    ) external nonReentrant {
        ...

            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 implement a validation check that ensures the `scAmount` does not exceed the user's entitlements before proceeding. If the `scAmount` is too high, a meaningful and clear error message should be returned, explaining that the user's entitlements are insufficient for the requested operation. This will improve user understanding and avoid unnecessary transaction reverts.

MEE - Missing Events Emission

Criticality	Minor / Informative
Location	Liquify.sol#L144
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 setReferralManager(IReferralManager _referralManager)
external 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.

MRVM - Missing RealToken Validation Mechanism

Criticality	Minor / Informative
Location	Liquify.sol#L123,238
Status	Unresolved

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 {
        if (tokens.length != states.length) revert
ArrayLengthMismatch();

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

    function initiatorDeposit(uint256 projectId, IERC20 realToken)
external {
        ProjectDetails storage project = projectDetails[projectId];
        ProjectState storage projectState = projectStates[projectId];

        if (msg.sender != project.projectOwner) revert
UnauthorizedAccess();

        ...
        projectState.vestingPhaseIndex++;
        projectState.realTokensAddress.safeTransferFrom(msg.sender,
address(this), denormalizedExpectedDeposit);
        emit RealTokensDeposited(projectId, denormalizedExpectedDeposit,
address(this));
    }
```

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.

MWBM - Missing Withdrawable Balance Method

Criticality	Minor / Informative
Location	Liquify.sol#L218
Status	Unresolved

Description

The contract is missing a method to return the total amount that the project owner is eligible to withdraw. While the `initiatorWithdraw` function allows the project owner to withdraw a specified amount, there is no available function for querying the total amount the project owner can currently withdraw. This could lead to confusion for the project owner and requires them to perform additional calculations off-chain to determine the withdrawable balance.

```
    function initiatorWithdraw(uint256 projectId, uint256 amount)
external {
        ProjectDetails storage project = projectDetails[projectId];
        ProjectState storage projectState = projectStates[projectId];

        ...
        if (normalizedAmount > availableAmount) revert
InsufficientFunds();

        projectState.totalAmountWithdrawn += normalizedAmount;
        projectState.totalFeesWithdrawn += projectState.raisedFees;

        IERC20 paymentToken = project.paymentToken;
        paymentToken.safeTransfer(msg.sender, amount);

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

Recommendation

It is recommended to implement a function that returns the total withdrawable balance for the project owner, taking into account the raised amount, fees, and previously withdrawn amounts. This would provide clarity to the project owner and make it easier to manage

withdrawals directly through the contract. Additionally, this would reduce the risk of incorrect withdrawal attempts due to miscalculations.

MU - Modifiers Usage

Criticality	Minor / Informative
Location	Liquify.sol#L223,242,270,290,301,672,708
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.

```
if (msg.sender != project.projectOwner
...
if (recipient == address(0)) {
    revert InvalidAddress();
}
...
if (owner == address(0) || recipient == address(0)) {
    revert InvalidAddress();
}
```

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.

PBV - Percentage Boundaries Validation

Criticality	Minor / Informative
Location	ReferralManager.sol#L83,93
Status	Unresolved

Description

The contract utilizes variables for percentage-based calculations that are required for its operations. These variables are involved in multiplication and division operations to determine proportions related to the contract's logic. If such variables are set to values beyond their logical or intended maximum limits, it could result in incorrect calculations. This misconfiguration has the potential to cause unintended behavior or financial discrepancies, affecting the contract's integrity and the accuracy of its calculations.

```
   function setReferrerFeeBps(uint16 newReferrerFeeBps) external
onlyOwner {
        referrerFeeBPS = newReferrerFeeBps;
        emit ReferrerFeeBpsUpdated(newReferrerFeeBps);
    }

    function setSuperReferrerFeeBps(uint16 newSuperReferrerFeeBps)
external onlyOwner {
        superReferrerFeeBPS = newSuperReferrerFeeBps;
        emit ReferrerFeeBpsUpdated(newSuperReferrerFeeBps);
    }
```

Recommendation

To mitigate risks associated with boundary violations, it is important to implement validation checks for variables used in percentage-based calculations. Ensure that these variables do not exceed their maximum logical values. This can be accomplished by incorporating `require` statements or similar validation mechanisms whenever such variables are assigned or modified. These safeguards will enforce correct operational boundaries, preserving the contract's intended functionality and preventing computational errors.

PSU - Potential Subtraction Underflow

Criticality	Minor / Informative
Location	Liquify.sol#L493
Status	Unresolved

Description

The contract subtracts two values, the second value may be greater than the first value if the contract owner misuses the configuration. As a result, the subtraction may underflow and cause the execution to revert.

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

Recommendation

The team is advised to properly handle the code to avoid underflow subtractions and ensure the reliability and safety of the contract. The contract should ensure that the first value is always greater than the second value. It should add a sanity check in the setters of the variable or not allow executing the corresponding section if the condition is violated.

PTAI - Potential Transfer Amount Inconsistency

Criticality	Minor / Informative
Location	Liquify.sol#L262,425
Status	Unresolved

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

```
projectState.realTokensAddress.safeTransferFrom(msg.sender,
address(this), denormalizedExpectedDeposit);
...
project.paymentToken.safeTransferFrom(msg.sender, address(this), _amount
- referrersFees); //not normalized
project.paymentToken.safeTransferFrom(msg.sender,
address(referralManager), referrersFees); //not normalized
```

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
```

TSI - Tokens Sufficiency Insurance

Criticality	Minor / Informative
Location	Liquify.sol#L238
Status	Unresolved

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
{
        ProjectDetails storage project = projectDetails[projectId];
        ProjectState storage projectState = projectStates[projectId];

        if (msg.sender != project.projectOwner) revert
UnauthorizedAccess();

        ...

        projectState.vestingPhaseIndex++;
        projectState.realTokensAddress.safeTransferFrom(msg.sender,
address(this), denormalizedExpectedDeposit);
        emit RealTokensDeposited(projectId, denormalizedExpectedDeposit,
address(this));
    }
```

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.

TRR - Trust-Based Refund Risk

Criticality	Minor / Informative
Location	Liquify.sol#L218,297
Status	Unresolved

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 {
        ProjectDetails storage project = projectDetails[projectId];
        ProjectState storage projectState = projectStates[projectId];

        if (projectState.status == ProjectStatus.Refund) revert
        ...
        paymentToken.safeTransfer(msg.sender, amount);

        emit FundsWithdrawn(projectId, amount, address(this));
    }

    function refund(uint256 projectId, uint256 amount) external {
        ProjectDetails storage project = projectDetails[projectId];
        ProjectState storage state = projectStates[projectId];

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

        // Calculate the remaining percentage and total refundable
amount if not already set
        if (state.totalRefundable == 0) {
            uint256 remainingPercentage = 0;
            for (uint256 i = state.vestingPhaseIndex; i <
project.vestingReleasePercentages.length; i++) {
                remainingPercentage +=
project.vestingReleasePercentages[i];
            }
            state.totalRefundable = (state.raisedAmount +
state.raisedFees) * remainingPercentage / MAX_BPS;
        }

       ...
            project.paymentToken.safeTransferFrom(msg.sender,
address(this), denormalizedTotalToRefund);

            if (state.totalRefundable == 0) {
                state.refundCompleted = true;
            }
        }

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

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.

WBPC - Withdraw Before Project Completion

Criticality	Minor / Informative
Location	Liquify.sol#L238,451
Status	Unresolved

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 {
        ProjectDetails storage project = projectDetails[projectId];
        ProjectState storage projectState = projectStates[projectId];

        if (projectState.status == ProjectStatus.Refund) revert
InvalidStatus(ProjectStatus.Refund, projectState.status);
        if (msg.sender != project.projectOwner) revert
UnauthorizedAccess();
        if (amount <= 0) revert InvalidAmount();
        uint256 normalizedAmount = normalizeTokenAmount(amount,
IERC20Metadata(address(project.paymentToken)).decimals());
        uint256 availableAmount = projectState.raisedAmount +
projectState.raisedFees - projectState.totalAmountWithdrawn;
        if (normalizedAmount > availableAmount) revert
InsufficientFunds();

        projectState.totalAmountWithdrawn += normalizedAmount;
        projectState.totalFeesWithdrawn += projectState.raisedFees;

        IERC20 paymentToken = project.paymentToken;
        paymentToken.safeTransfer(msg.sender, amount);

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

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.

L04 - Conformance to Solidity Naming Conventions

Criticality	Minor / Informative
Location	Liquify.sol#L69,118,144,361
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.

```
address public immutable LIQUIDERC20_IMPLEMENTATION;
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.

L14 - Uninitialized Variables in Local Scope

Criticality	Minor / Informative
Location	ReferralManager.sol#L142,143,144,145
Status	Unresolved

Description

Using an uninitialized local variable can lead to unpredictable behavior and potentially cause errors in the contract. It's important to always initialize local variables with appropriate values before using them.

```
uint256 superReferrerFee1
uint256 superReferrerFee2
uint256 referrerFee
uint256 userReferralFee
```

Recommendation

By initializing local variables before using them, the contract ensures that the functions behave as expected and avoid potential issues.

L19 - Stable Compiler Version

Criticality	Minor / Informative
Location	ReferralManager.sol#L2 Liquify.sol#L2 LiquidERC20.sol#L2 interfaces/IReferralManager.sol#L2 interfaces/ILiquify.sol#L2
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 ^0.8.25;
```

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.

L20 - Succeeded Transfer Check

Criticality	Minor / Informative
Location	Liquify.sol#L443,488
Status	Unresolved

Description

According to the ERC20 specification, the transfer methods should be checked if the result is successful. Otherwise, the contract may wrongly assume that the transfer has been established.

```
IERC20(tokenClone).transfer(msg.sender, tokensToClaim);
state.realTokensAddress.transfer(msg.sender,
denormalizedTotalRealTokens);
```

Recommendation

The contract should check if the result of the transfer methods is successful. The team is advised to check the SafeERC20 library from the Openzeppelin library.

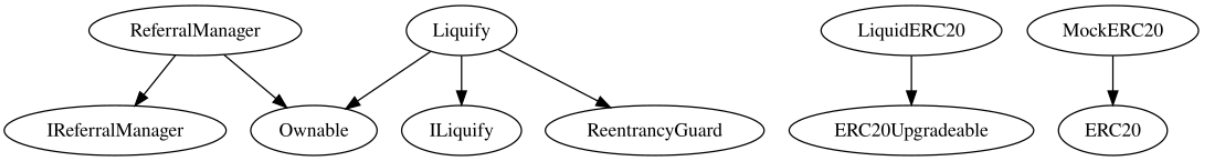
Functions Analysis

Contract	Type	Bases		
	Function Name	**Visibility**	**Mutability**	**Modifiers**
ReferralManager	Implementation	IReferralManager, Ownable		
		Public	✓	Ownable
	setReferrers	External	✓	onlyOwner
	setReferrerFeeBps	External	✓	onlyOwner
	setSuperReferrerFeeBps	External	✓	onlyOwner
	withdrawReferrerBalance	External	✓	-
	processReferralFees	External	✓	-
	denormalizeTokenAmount	Public		-
Liquify	Implementation	ILiquify, Ownable, ReentrancyGuard		
		Public	✓	Ownable
	updateInitiators	External	✓	onlyOwner
	updatePaymentTokens	External	✓	onlyOwner
	withdrawProtocolFees	External	✓	onlyOwner nonReentrant
	setReferralManager	External	✓	onlyOwner
	createProject	External	✓	-
	initiatorWithdraw	External	✓	-
	initiatorDeposit	External	✓	-

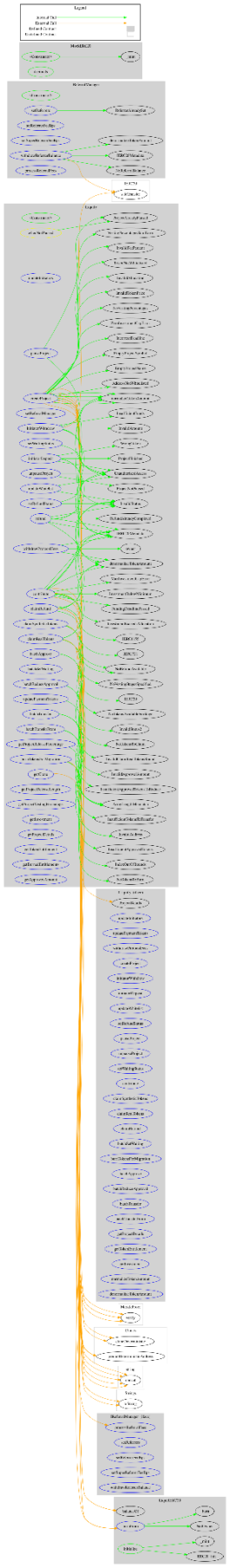
	updateWhitelist	External	✓	-
	setRefundStatus	External	✓	-
	refund	External	✓	-
	pauseProject	External	✓	-
	unpauseProject	External	✓	-
	setWaitingStatus	External	✓	-
	contribute	External	✓	nonReentrant whenNotPaused
	claimSyntheticTokens	External	✓	-
	claimRealTokens	External	✓	nonReentrant
	claimRefund	External	✓	nonReentrant
	batchSetWaiting	External	✓	-
	burnTokensForMigration	External	✓	nonReentrant
	batchApprove	External	✓	nonReentrant
	batchReduceApproval	External	✓	nonReentrant
	batchTransfer	External	✓	nonReentrant
	batchTransferFrom	External	✓	nonReentrant
	getClone	External		-
	getProjectReleasePercentage	External		-
	getProjectReleaseLength	External		-
	getProjectVestingPercentage	External		-
	getInvestment	External		-
	getProjectDetails	External		-
	getTokenEntitlement	External		-
	getBurnedEntitlements	External		-

	getApprovedAmount	External		-
	normalizeTokenAmount	Public		-
	denormalizeTokenAmount	Public		-
LiquidERC20	Implementation	ERC20Upgradeable		
	initialize	Public	✓	initializer
	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