BAIL security

BAILSEC.IO

OFFICE@BAILSEC.IO

X: @BAILSECURITY

TG: @HELLOATBAILSEC

FINAL REPORT

Liquify

January 2025

Disclaimer:

Security assessment projects are time-boxed and often reliant on information that may be provided by a client, its affiliates, or its partners. As a result, the findings documented in this report should not be considered a comprehensive list of security issues, flaws, or defects in the target system or codebase.

The content of this assessment is not an investment. The information provided in this report is for general informational purposes only and is not intended as investment, legal, financial, regulatory, or tax advice. The report is based on a limited review of the materials and documentation provided at the time of the audit, and the audit results may not be complete or identify all possible vulnerabilities or issues. The audit is provided on an "as-is," "where-is," and "as-available" basis, and the use of blockchain technology is subject to unknown risks and flaws.

The audit does not constitute an endorsement of any particular project or team, and we make no warranties, expressed or implied, regarding the accuracy, reliability, completeness, or availability of the report, its content, or any associated services or products. We disclaim all warranties, including the implied warranties of merchantability, fitness for a particular purpose, and non-infringement.

We assume no responsibility for any product or service advertised or offered by a third party through the report, any open-source or third-party software, code, libraries, materials, or information linked to, called by, referenced by, or accessible through the report, its content, and the related services and products. We will not be liable for any loss or damages incurred as a result of the use or reliance on the audit report or the smart contract.

The contract owner is responsible for making their own decisions based on the audit report and should seek additional professional advice if needed. The audit firm or individual assumes no liability for any loss or damages incurred as a result of the use or reliance on the audit report or the smart contract. The contract owner agrees to indemnify and hold harmless the audit firm or individual from any and all claims, damages, expenses, or liabilities arising from the use or reliance on the audit report or the smart contract.

By engaging in a smart contract audit, the contract owner acknowledges and agrees to the terms of this disclaimer.

This document provides an inverted chronological summary of security audit reports, starting with the 3rd Audit Report and progressing back to the 1st Audit Report. This structure was initiated by the client. Each audit presents findings, security assessments, and recommendations aimed at enhancing the security posture of the system under review.

3rd Audit

Important:

Please ensure that the deployed contract matches the source-code of the last commit hash.

Project	Liquify — 3rd Audit
Website	liquify.ventures
Language	Solidity
Methods	Manual Analysis
Github repository	https://github.com/Liquify-labs/sc/tree/a133ca5064732ea3606da349473048b8d24b2a58
Resolution 1	https://github.com/Liquify-labs/sc/tree/2ecfe16243d2fb9c44071c544d7ddc7ac0e7e86e

Detection Overview

Severity	Found	Resolved	Partially Resolved	Acknowledged (no change made)
High				
Medium	4	2		2
Low				
Informational				
Governance				
Total	4	2		2

Detection Definitions

Severity	Description
High	The problem poses a significant threat to the confidentiality of a considerable number of users' sensitive data. It also has the potential to cause severe damage to the client's reputation or result in substantial financial losses for both the client and the affected users.
Medium	While medium level vulnerabilities may not be easy to exploit, they can still have a major impact on the execution of a smart contract. For instance, they may allow public access to critical functions, which could lead to serious consequences.
Low	Poses a very low-level risk to the project or users. Nevertheless the issue should be fixed immediately
Informational	Effects are small and do not post an immediate danger to the project or users
Governance	Governance privileges which can directly result in a loss of funds or other potential undesired behavior

Detection

Issue_01	Users are not able to contribute in certain scenarios
Severity	**Medium**

Description	Liquify.sol In 507 incorrectly reverts when a user contributed slightly above the remaining amount. The code will not allow this contribution when it otherwise should.
	*uint256 lowerBound = leftover+ (leftover * 10) / 10000; // leftover * 1.001*
	*uint256 upperBound = leftover+ (leftover * 100) / 10000; // leftover * 1.01*
	if (amountAfterFee > lowerBound && amountAfterFee < upperBound) {
	we can see that if amountAfterFee is less than lowerBound, meaning we contributed slightly more than the remaining amount, then the function will revert and not allow this contribution because it will not be greater than the lowerBound. This will result in users being unable to contribute amounts that are in between the lowerBound and the leftover value.

Recommendations	If the user is contributing an amount that is below the lowerBound, we should allow this contribution as it is a valid contribution. There is no need for a lowerBound.
Comments / Resolution	Resolved by following the recommendation.

Issue_02	Erc721 whitelist does not check if user is still owner during subsequent contributions
Severity	**Medium**

Description	In liquify.sol ln 487 when the whitelist type is AddressAndErc721Whitelist or Erc721Whitelist, the code does not check that the user is still the owner of the NFT for subsequent investments.
	if(state.usedNFTs[tokenId] == address(0)){
	if (IERC721(state.nftWhitelistContract).ownerOf(tokenId) != msg.sender) revert NoErc721Token();
	state.usedNFTs[tokenId] = msg.sender;
	}
	else if (state.usedNFTs[tokenId] != msg.sender) revert NftAlreadyClaimedByAnother();
	}
	else if (state.usedNFTs[tokenId] != msg.sender) revert NftAlreadyClaimedByAnother();
	If the user owned the NFT in his first contribution and sold the NFT, he will still be able to contribute even though he is no longer the owner of said NFT.
	This is in contrast to the whitelist for address which always checks the root. This opens a scenario where a user can buy an NFT, then contribute, then sell the NFT back and be eligible for future contribution even if he is currently not the owner of the NFT during said contribution.

Recommendations	Check the NFT is owned by the user even if usedNFTs is set to the msg.sender.
Comments / Resolution	Resolved by following the recommendation.

Issue_03	When user’s amount is downscaled, they are still charged fees on the original amount
Severity	**Medium**

Description	liquify.sol In 513 Users are charged the original amount even if their investment was downscaled.
	if (amountAfterFee > lowerBound && amountAfterFee < upperBound) {
	amountAfterFee = leftover;
	In the scenario where the amount a user is investing is slightly above the projectAllocation, his value is downscaled to the leftover amount.
	However the user is still transferred out the non downscaled amount but still given the downscaled allocation. The fees are also charged on the non downscaled amount.
	project.paymentToken.safeTransferFrom(msg.sender, address(this), _amount - denormalizedReferralFees);
	project.paymentToken.safeTransferFrom(msg.sender, address(referralManager), denormalizedReferralFees);
	This will result in the user overpaying for allocation and fees.

Recommendations	Transfer out and charge the user fees on the amount that is downscaled to leftover, not the user’s original amount. If it is intended to transfer out the original amount, simply charge fees on the new amount that is downscaled in the leftover logic.
Comments / Resolution	Acknowledged.

Issue_04	Malicious project owner can withdraw excessive amounts when paymentToken.decimals>18
Severity	**Medium**

Description	A malicious project owner will be able to silently withdraw excessive amounts, which will eventually prevent the last user from calling claimRefund.
	Consider the following scenario:
	
	0. A project owner creates a project which will use YAM-V2 as a payment token, which has 24 decimals.
	
	1.The owner will set the allocation to 10e24 and Alice and Bob will have contributed 5e24 tokens each.
	
	The owner will simply call initiatorWithdraw with 999_999 as the amount. Due to the normalization, the normalizedAmount will be rounded down to 0. As a result the raisedAmountWithdrawn and the totalFeesWithdrawn will not be updated properly as they will use the normalizedAmount.
	However the paymentTokenTotalBalance will be updated will be updated with the actual amount variable:
	
	projectState.paymentTokenTotalBalance -= amount;
	
	2.After withdrawing X amount of tokens the owner calls setRefundStatus. As the total withdrawn amounts were not properly updated, the setRefundStatus will incorrectly assume that there were no withdrawn amounts directly putting the project in the following state:
	  else {
	
	state.refundFunds = amountToRefund;
	state.refundCompleted = true;
	emit RefundDeposit(projectId, amountToRefund);
	}
	state.status = ProjectStatus.Refund;

	As a result the project will assume that the owner has nothing to refund, leaving the contract in an insolvent state. Now the contract may still have enough funds to satisfy Alice's claimRefund. But Bob's request will always revert with an underflow error here: *state.paymentTokenTotalBalance -= denormalizedRefund;* as it will be less than 5e24, leaving whatever amount is left stuck in the contract. This vulnerability could also be triggered unintentionally when the paymentToken has more than 18 decimals and initiatorWithdraw is called with an amount that does not satisfy the following check: *amount % excessiveDecimals == 0* As the last user will not be able to call claimRefund due to underflow of the paymentTokenTotalBalance.
Recommendations	When using tokens with high decimal places ensure that there is no rounding in the initiatorWithdraw function.
Comments / Resolution	Acknowledged.

2nd Audit — Competition

Participants:

CryptoStaker
0xBepresent
Nisedo
Breeje

Important:
Please ensure that the deployed contract matches the source-code of the last commit hash.

Project	Liquify - Audit Competition
Website	liquify.ventures
Language	Solidity
Methods	Manual Analysis
Github repository	https://github.com/Liquify-labs/sc/commit/f75b79e7eabac5a3766d346e08c527521b380fcb
Resolution 1	https://github.com/Liquify-labs/sc/tree/cc2d37da7b04befe87e348d8513d5843585a50fc/src

Detection Overview

Severity	Found	Resolved	Partially Resolved	Acknowledged (no change made)
High	5	4		1
Medium	8			6
Low				
Informational				
Governance				
Total	13	4		7

Detection Definitions

Severity	Description
High	The problem poses a significant threat to the confidentiality of a considerable number of users' sensitive data. It also has the potential to cause severe damage to the client's reputation or result in substantial financial losses for both the client and the affected users.
Medium	While medium level vulnerabilities may not be easy to exploit, they can still have a major impact on the execution of a smart contract. For instance, they may allow public access to critical functions, which could lead to serious consequences.
Low	Poses a very low-level risk to the project or users. Nevertheless the issue should be fixed immediately
Informational	Effects are small and do not post an immediate danger to the project or users
Governance	Governance privileges which can directly result in a loss of funds or other potential undesired behavior

Detection

Liquify

For explanation and privileged functions, see first report.

Issue_01	Contract is fully drainable of realToken due to incorrect index check within claimRealTokens
Severity	**High**

Description	The claimRealTokens function allows a user to claim their rightful amount of realTokens based on the entitlement. There is a safeguard which prevents claiming tokens for a round where the project creator has not yet provided any realTokens:
	if (round > state.vestingPhaseIndex)
	revert InvalidClaimRealTokensRound(round, state.vestingPhaseIndex);
	This check is however erroneous. If the owner deposits realTokens for round0, it will automatically set state.vestingPhaseIndex to 1:
	else projectState.vestingPhaseIndex++;
	This means if the owner has only deposited tokens for round0, users can incorrectly claim realTokens for round1 as well. These tokens were however never provided, which can be cleverly abused to drain the whole protocol of that token.

Recommendations	Consider ensuring users can in fact only claim for the round where a deposit has happened. We furthermore recommend adding an internal accounting system (which has already been discussed in the previous audit and was not

	applied). If such a system is added, a normal resolution round is insufficient.
Comments / Resolution	Resolved. The claimRealTokens function only allows redeeming realTokens for valid rounds now including the edge case where vestingPhaseIndex is not increased for last round.

Issue_02	PaymentTokens can be drained due to erroneous phase updated paired with incorrect vestingPhaseIndex setting
Severity	**High**
Description	The Liquify contract contains a function setRefundStatus() that allows the project owner to set the project status to Refund. If the project status is changed from Finished to Refund and users have already claimed all their realTokens, the remaining paymentTokens could potentially become stuck in the contract. This issue arises because once all realTokens are claimed, the remaining amount is deserved by the project owner. However, since the project.status is set to refund, the initiatorWithdraw function will revert, causing the funds to become trapped in the contract. Furthermore, in the scenario where the project has been Finished and setRefund is called, users can still erroneously partially claim their refund for the last round due to the incorrect adjustment of vestingPhaseIndex: *if (vestingPhaseIndex == project.vestingReleasePercentages.length - 1) projectState.status = ProjectStatus.Finished;* *else projectState.vestingPhaseIndex++;* This allows for draining the contract of paymentTokens.

Recommendations	Consider adding a validation in the setRefundStatus() function to revert when the project.status is in the Finished status. We furthermore recommend more strict phase controls, as already recommended in the first iteration. If the vestingPhaseIndex variable is changed, this will have an impact on all parts of the code where it is used. These are: - setRefundStatus - claimRealTokens - claimRefund All these functions must be fully re-audited with the updated vestingPhaseIndex logic in mind.
Comments / Resolution	Resolved. Validation was added to ensure the faulty transition can not happen anymore.

Issue_03	Incorrect Withdrawal Amount Distribution in setRefundStatus allows for draining the whole contract
Severity	**High**
Description	setRefundStatus is incorrectly using state.raisedAmountWithdrawn instead of totalAmountWithdrawn in the payout calculation, leading to the initiator receiving more funds than entitled and users receiving less than their fair share. To understand this issue, we first need to understand how initiatorWithdraw works. First it fills the projectState.raisedAmountWithdrawn variable and once that is filled, it fills projectState.totalFeesWithdrawn. If now a creator has: a) Fully filled raisedAmountWithdrawn b) Partially/Fully filled projectState.totalFeesWithdrawn And the creator now calls setRefundStatus, the creator is entitled to receive: state.raisedAmount + state.raisedFees - amountToRefund Now we need to keep in mind that the creator has already partially withdrawn funds which means in fact the creator is not entitled to receive the full value above, rather we need to deduct totalAmountWithdrawn from this value to calculate the to received amount. However, since only state.totalFeesWithdrawn is deducted, this value will be incorrect. This allows to fully drain the contract. PoC: 1. Alice creates a project with one phase and a contribution of 100e18 and a price of 1e18

	2. Bob contributes to Alice's project, depositing 110 USDT (10 USDT is the initiatorFee) > state.raisedAmount = 100e18 > state.raisedFees = 10e18 3. Alice deposits the realToken in the contract which immediately marks the project as Finished. 4. Alice calls initiatorWithdraw and withdraws 109 USDT > projectState.raisedAmountWithdrawn = 100e18 > projectState.totalFeesWithdrawn = 9e18 TRICKY: Alice intentionally withdrew 109 USDT such that the setRefundStatus enters the following if-clause: *if (totalAmountWithdrawn < deservedAmount) {* 5. Alice calls setRefundStatus > deservedAmount = 110e18 > totalAmountWithdrawn = 109e18 > payout = deservedAmount - state.raisedAmountWithdrawn > 110e18 - 100e18 = 10e18 > Alice withdrew 10 USDT while in fact she already withdrew 109 USDT This can be repeated to drain the whole contract.
Recommendations	Consider ensuring that setRefundState cannot be called once the project is finished and consider ensuring that the correct payout amount is calculated.
Comments / Resolution	Resolved, it is now ensured that such a transition cannot occur. Furthermore, the payout is now calculated using totalAmountWithdrawn

Issue_04	Some users may not be able to claim their real token because of an incorrect check in claimRealTokens
Severity	**High**
Description	If users have synthetic tokens or entitlements, they can swap them for real tokens once the initiator calls initiatorDeposit and deposits the real tokens. The initiatorDeposit call deposits real tokens per vesting phase, so if there are 10 vesting phases, the initiator has to call initiatorDeposit 10 times. In claimRealTokens, L563 checks that the round is greater than the vestingPhaseIndex, and revert if it is so. This check is problematic because a user can claim 2 vesting phases at the expense of another user's real token. For example, let's say there are two vesting phases, 50% allocation each. The total raised amount is 1000e18 by 2 users (Alice and Bob) and pricePerToken is 2e18. The initiator calls initiatorDeposit and deposits the first batch of real tokens, which amounts to 500e18. vestingPhaseIndex was zero, and when initiatorDeposit is called, vestingPhaseIndex is now one. By right, Alice and Bob can only get 250e18 real tokens each if they call claimRealTokens. However, Alice can call claimRealTokens(projectId, [0,1]) and claim for both schedules because in the *round > state.vestingPhaseIndex check*, round is 0 and vestingPhaseIndex is one, so the check passes.

	round is 1 and vestingPhase is one, the check passes again. Bob cannot get his 250e18 real tokens, and has to wait for the depositor to call initiateDeposit again to collect his 500e18 real tokens. Also, if another project uses the same real token, users can take their real tokens first by claiming twice.
Recommendations	Consider changing round > state.vestingPhaseIndex to state >= vestingPhaseIndex, while at the same time correctly incrementing the index to ensure that users can fully claim. In the scenario where *state >= vestingPhaseIndex* is set, there is an edge-case where users cannot claim tokens in the last round, which is due to the fact how vestingPhaseIndex is increased: *if (vestingPhaseIndex == project.vestingReleasePercentages.length - 1) projectState.status = ProjectStatus.Finished;* *else projectState.vestingPhaseIndex++;* As it is not increased for the very last round and thus users cannot claim for this round. Furthermore, all spots within the code where vestingPhaseIndex is used must be re-checked with the new change in mind. These are: - setRefundStatus - claimRealTokens - claimRefund All these functions must be fully re-audited with the updated vestingPhaseIndex logic in mind.

Comments / Resolution	Resolved, the claimRealTokens function only allows redeeming realTokens for valid rounds now including the edge case where vestingPhaseIndex is not increased for last round.

Issue_05	Unfair claim refund distribution can lead to unequal refunds for contributors
Severity	**High**
Description	The function Liquify::claimRefund is designed to allow contributors to claim refunds from state.refundFunds. However, the current implementation has a flaw where if a contributor's refund entitlement exceeds the available funds in state.refundFunds, the transaction reverts due to the next condition: *if(totalRefund > state.refundFunds) revert NoRefundAvailable();* This can result in situations where a contributor who is entitled to a large refund may be unable to claim any refund if the available funds are insufficient, while others who claim earlier may deplete the funds. The flaw in this logic can lead to unfair distribution of refunds, where contributors with larger entitlements may end up receiving none if others have already claimed the limited available funds. The root-cause of this issue is that users can call claimRefund while in fact the refund is not yet fully processed. There are likely other side-effects with that implementation.
Recommendations	Consider only allowing to call claimRefund if state.refundCompleted = true.
Comments / Resolution	Acknowledged.

Issue_06	Precision Loss Due to Truncation in claimRealTokens
Severity	**Medium**
Description	When converting between tokens with different decimal places, users may suffer precision loss due to truncation, particularly when converting from higher to lower decimal places. In claimRealTokens, if the real token has 6 decimals, it will convert 18 decimals (synthetic token) to 6 decimals leading to truncation, potentially leading to loss of funds for users.
Recommendations	Consider only allowing 18 decimals realTokens.
Comments / Resolution	Acknowledged.

Issue_07	Lack of validation for referral fees exceeding protocol fee basis points
Severity	**Medium**

Description	The Liquify::setReferralManager function does not include validation to ensure that the sum of referral fees does not exceed the protocolFeeBPS.
	function setReferralManager(IReferralManager _referralManager) external whenNotPaused onlyOwner {
	referralManager = _referralManager;
	}
	This oversight can lead to an overflow in the Liquify::contribute function, specifically at Liquify#L508.
	protocolFees[project.paymentToken] += protocolFee - referrersFees;
	When the total referral fees calculated are greater than the

	protocolFee, an arithmetic overflow occurs, potentially causing the transaction to revert unexpectedly. This vulnerability can disrupt the functionality of the smart contract, affecting users' ability to contribute funds in time with funding deadlines and potentially causing loss of funds (paymentTokens or contributions) for projects.

Recommendations	Consider adding validation checks in the setReferralManager function to ensure that the total referral fees do not exceed the protocolFeeBPS.
	
	diff
	function setReferralManager(IReferralManager _referralManager) external whenNotPaused onlyOwner {
	referralManager = _referralManager;
	+ if (referralManager.getTotalReferralsFeesInBPS() > protocolFeeBPS) revert ReferralBPSExceedsProtocolBPS();
	}

Comments / Resolution	Acknowledged.

Issue_08	Blacklisting project owner prevents contributor refunds
Severity	**Medium**
Description	In the current implementation in Liquify::setRefundStatus, If the project owner is blacklisted, preventing any token transfers to their address in codeline Liquify#L388, this could disrupt the usual flow of funds necessary for processing refunds to contributors. *if (totalAmountWithdrawn < deservedAmount) {* *// Initiator hasn't withdrawn enough and deserves additional funds based on vesting* *uint256 payout = deservedAmount - state.raisedAmountWithdrawn;* *uint256 denormalizedPayout = denormalizeTokenAmount(payout, IERC20Metadata(address(project.paymentToken)).decimals());* *project.paymentToken.safeTransfer(msg.sender, denormalizedPayout);* */ Set the full amount to be refunded to users and mark refund as complete* *state.refundFunds = amountToRefund;* *state.refundCompleted = true;* *emit RefundDeposit(projectId, amountToRefund);* *} else if (totalAmountWithdrawn > deservedAmount) {* Consider the following scenario: 1. The projectOwner creates a project using USDc as the paymentToken (USDc has a blacklist functionality). The project has a vestingReleasePercentages of 50% and 50%. 2. Contributors contribute 1000 USDc. 3. The projectOwner calls Liquify::initiatorDeposit and deposits the first 50% of realTokens. 4. Something happens, and the projectOwner is blacklisted. 5. The projectOwner attempts to return the remaining 50% of the paymentTokens by calling Liquify::setRefundStatus, but the function reverts because the projectOwner is blocked by USDc.

	In the end, the 1000 USDc (50% assigned to the projectOwner and the remaining 50% that should be returned to the contributors) would end up stuck in the Liquify contract.
Recommendations	The setRefundStatus function should be executable even if the projectOwner is blocked, to avoid affecting the contributors who are not blocked. The best approach would be to add the deservedAmount to a variable so that it can later be claimed by the projectOwner. Alternatively, one can implement a separate “to” address which allows for receiving these tokens.
Comments / Resolution	Acknowledged.

Issue_09	Deadlock in contribute Function Could Prevent Projects from Reaching Full Allocation
Severity	**Medium**
Description	The root cause lies in how the user contribution amount is validated against minInvestmentCap and maxInvestmentCap in Line A and Line B below. *A-> uint256 maxRemainingInvestment = project.maxInvestmentCap - state.investments[msg.sender];* *// Check if it's a new contribution or a top-up contribution* *B-> if (amount < project.minInvestmentCap && amount != maxRemainingInvestment) {* *revert InvestmentBelowMinimum();* *}* *if (amount > maxRemainingInvestment) revert MaxInvestmentCapError(state.investments[msg.sender], project.maxInvestmentCap);* Consider the following Scenario: 1. Alice has created a Project with: • Allocation = 100_000 Tokens • maxInvestmentCap = 20_000 Tokens • minInvestmentCap = 5_000 Tokens 2. The check on line B is supposed to allow contributions below minInvestmentCap only if the user's contribution fully tops up their investment to exactly match the maxInvestmentCap. 3. Issue Scenario: • Suppose the total investment reaches 96_000 tokens. • Now, only 4_000 tokens remain to be allocated.

	• The issue arises here: o New users cannot contribute, as their contribution amount will be less than the minInvestmentCap (5_000 tokens), and their maxRemainingInvestment (20_000 tokens) will cause the check to fail at line B, resulting in a revert. o If no existing user has an investment value that matches exactly 16_000, their transaction will also revert at Line B. This creates a deadlock situation where no further contributions are possible, effectively DoSing the contribute function and preventing it from reaching full allocation.
Recommendations	Consider updating the logic to allow contributions below minInvestmentCap in two specific scenarios instead of just one: • If the contribution tops up the user's investment to reach their maxInvestmentCap. OR • If the contribution completes the project's full allocation & User's current investment is more than minInvestmentCap to maintain minInvestmentCap invariant.
Comments / Resolution	Failed resolution, we recommend reversing the change and acknowledging the issue.

Issue_10	updateWhitelist is not set when project is created, leading to frontrunning issues
Severity	**Medium**
Description	The createProject function and the updateWhitelist function are separate functions. After the project is created and before the owner updates the whitelist, a user can frontrun the update and interact with the protocol. In the scenario where a protocol is created, the corresponding root is 0x0 which can result in side-effects including these potentially tricking the verification.
Recommendations	Consider executing a check which ensures the root is set.
Comments / Resolution	Acknowledged.

Issue_11	Using Global Index as Project ID Exposes Users to Risks of Contributing to the Wrong Project During Block Reorgs
Severity	**Medium**
Description	The current implementation of the createProject function uses a globalIndex to assign project IDs sequentially. If two project owners create projects at almost the same time, they get consecutive project IDs. While this works under normal circumstances, a blockchain reorganization (reorg) can cause transactions to be processed in a different order. This reshuffling changes the assignment of project IDs, which leads to users unknowingly contributing to the wrong projects for a brief period. How This Can Happen: 1. Normal Case (Block A) • Alice creates a project using createProject and is assigned ProjectId = 1. • Bob creates another project in the same block and is assigned ProjectId = 2. • User 1 contributes 100 tokens to Alice's project using ProjectId = 1. • User 2 contributes 200 tokens to Bob's project using ProjectId = 2. 2. Reorg Happens • A reorg occurs, and the order of transactions is rearranged. In the new chain (Block B), Bob's transaction is processed before Alice's.

3. New Order (Block B)

- Bob’s project is created first, so he now gets ProjectId = 1.
- Alice’s project is created second, so she now gets ProjectId = 2.
- Contributions are now misaligned:
    - User 1’s 100 tokens, originally intended for Alice’s project, are now sent to Bob’s project.
    - User 2’s 200 tokens, originally intended for Bob’s project, are now sent to Alice’s project.

Preconditions here:

- Both projects had the same payment token.
- The users’ contribution amount meets the minimum and maximum investment caps for both projects.
- Either there is no Whitelist in both Projects or A User is whitelisted for Both the projects.

Reorgs are common in blockchain networks. For example, Polygon has experienced over 30 reorg events in the past 30 days, as can be seen here: https://polygonscan.com/blocks_forked.

In the past reorg events there have lasted as long as 1.5 Minutes. Similarly, optimistic rollups (like Optimism and Arbitrum) are also susceptible to reorgs, especially in cases where a fraud proof is submitted.

Calculation of salt also uses globalIndex which leads to a similar issue as described above, in case users claim synthetic tokens after contribution which then reorgs. Users can get synthetic tokens of different project.

```
    bytes32 salt = keccak256(abi.encodePacked(globalIndex,
uint256(0)));
```

Recommendations	Use a Key which is a combination of Global Index and initiator (msg.sender) as ProjectId rather than incremental Global Index.
	uint256 projectId = uint256(keccak256(abi.encodePacked(globalIndex, msg.sender));
	Also update calculation of salt as:
	bytes32 salt = keccak256(abi.encodePacked(projectId, uint256(0)));

Comments / Resolution	Acknowledged.

Issue_12	Raised amounts may not reach allocation amounts in certain edge case
Severity	**Medium**
Description	In the contribute function, if the raisedAmount equals the allocation amount, the status of the project will be updated to ProjectStatus.Waiting. There can be a case where the raisedAmount will never reach the allocation because the remaining amount is too small. For example, if allocation is 1000e18 and raisedAmount is 999e18, a user cannot contribute anymore if the pricePerToken is 2e18.
Recommendations	Consider allowing for overflow if the raised amount is greater than the allocation and refunding the last user, while simultaneously setting the project status to ProjectStatus.Waiting.
Comments / Resolution	Failed resolution, we recommend reverting the change and acknowledging the issue.

LiquidER20

For explanation and privileged functions, see first report.

ReferralManager

For explanation and privileged functions, see first report.

Issue_13	Inadequate validation in ReferralManager leading to fee collection discrepancy
Severity	**Medium**

Description	The ReferralManager::setReferrers function allows for the assignment of superReferrer and referrer addresses associated with an initiator. The function does not enforce a validation check to ensure that if a superReferrer is specified, a corresponding referrer must also be non-zero.
	
	function setReferrers(address initiator, address superReferrer, address referrer) external onlyOwner whenNotPaused {
	if (initiatorToReferrers[initiator].referrer != address(0)) revert ReferrerAlreadySet();
	nitiatorToReferrers[initiator] = Referrers({
	superReferrer: superReferrer,
	referrer: referrer
	});
	emit ReferrerSet(initiator, superReferrer, referrer);
	}
	
	Consequently, a scenario could arise where a superReferrer is set with a non-zero address while the referrer remains zero, the problem is that in this configuration the superReferrer fees will not be collected. In the

<table>
<tr><td></td><td>Liquify::contribute function, referral fees are processed, but the validation does not verify whether the projectOwner has a superReferrer using referralManager::getInitiatorSuperReferrer. As a result, fees that should be allocated to the superReferrer may not be collected.

uint256 referrersFees;
if(userReferrer != address(0) ||
referralManager.getInitiatorReferrer(project.projectOwner) !=
address(0)){
referrersFees = referralManager.processReferralFees(msg.sender,
project.projectOwner, userReferrer, superReferrer,
project.paymentToken, amount);
}</td></tr>
<tr><td><b>Recommendations</b></td><td>There are two possible options:

1. If that configuration is valid (superReferrer is non-zero and referrer is zero), then update the contribute function to enable the collection of superFees without requiring a referrer. Also adjust the ReferralManager::processReferralFees function to be able to process superReferrers fees.

-        if(userReferrer != address(0) ||
referralManager.getInitiatorReferrer(project.projectOwner) !=
address(0)){

+   if(userReferrer != address(0) ||
referralManager.getInitiatorReferrer(project.projectOwner) != address(0)
|| referralManager.getInitiatorSuperReferrer(project.projectOwner)){
            referrersFees =
referralManager.processReferralFees(msg.sender,
project.projectOwner, userReferrer, superReferrer,
project.paymentToken, amount);
        }</td></tr>
</table>

	2. If that configuration is incorrect (superReferrer is non-zero and referrer is zero), implement additional checks in the ReferralManager::setReferrers function to ensure that if a superReferrer is specified, a referrer must also be non-zero. This will prevent inadvertent configurations of referrer relationships that could result in fee collection discrepancies.
Comments / Resolution	Acknowledged.

1. Project Details

Important:

Please ensure that the deployed contract matches the source-code of the last commit hash.

Project	Liquify — 1st Audit
Website	liquify.ventures
Language	Solidity
Methods	Manual Analysis
Github repository	https://github.com/Liquify-labs/sc/tree/791e0fc4eaa4e546cee9df1e4b9581b23badff9d/src
Resolution 1	

2. Detection Overview

Severity	Found	Resolved	Partially Resolved	Acknowledged (no change made)
High	20			
Medium	9			
Low	9			
Informational	3			
Governance	2			
Total	43			

2.1 Detection Definitions

Severity	Description
High	The problem poses a significant threat to the confidentiality of a considerable number of users' sensitive data. It also has the potential to cause severe damage to the client's reputation or result in substantial financial losses for both the client and the affected users.
Medium	While medium level vulnerabilities may not be easy to exploit, they can still have a major impact on the execution of a smart contract. For instance, they may allow public access to critical functions, which could lead to serious consequences.
Low	Poses a very low-level risk to the project or users. Nevertheless the issue should be fixed immediately
Informational	Effects are small and do not post an immediate danger to the project or users
Governance	Governance privileges which can directly result in a loss of funds or other potential undesired behavior

3. Detection

This report excludes a resolution round, as a further auditing round takes place at the conclusion.

LiquidERC20

The LiquidERC20 token is a simple ERC20 token that exposes a burnFrom function which allows the contract owner to burn tokens from any address. It is deployed within the Liquify contract using the Clonable pattern, during the creation of new projects and serves as a synthetic token which represents entitlements.

Privileged Functions

- burnFrom

No issues found.

ReferralManager

The ReferralManager contract is designed to manage the referral system for the Liquify contract. It tracks referrers and calculates referral fees in basis points (BPS), allowing users to withdraw their accumulated referral balances. The contract supports both caller-related and project-owner-related referral fees, including regular and super referrers.

Key features of the contract include:

- **Referral Tracking**: Maintains mappings to track initiators and their associated referrers and super-referrers.
- **Fee Management**: Allows the owner to set and update referral fees for both regular and super-referrers, ensuring that combined fees do not exceed predefined maximums.
- **Authorized Contracts**: Manages a list of authorized Liquify contracts that can interact with the referral system. This should only be the Liquify contract.

- **Referral Fee Processing**: Handles the calculation and distribution of referral fees during contributions, updating referrers' balances accordingly.
- **Withdrawal Functionality**: Enables referrers to withdraw their referral earnings.

The contract furthermore uses OpenZeppelin's UUPSUpgradeable pattern for future implementation upgrades and is pausable.

Privileged Functions

- transferOwnership
- renounceOwnership
- addLiquifyContract
- removeLiquifyContract
- setReferrers
- setReferrerFees
- pause
- unpause

Issue_01	Governance Issue: Governance can steal funds
Severity	**Governance**
Description	Currently, the addLiquifyContract function trivially allows adding a new liquifyContract which then allows to increase referral fees to steal all tokens. Furthermore, the contract is meant to be behind a proxy contract, which means that the proxy admin can change the implementation.
Recommendations	Consider incorporating a Gnosis Multisignature contract as owner and ensuring that the Gnosis participants are trusted entities.
Comments / Resolution	

Issue_02	Fee calculation will result in incorrect referrerFee
Severity	High

Description	The current way how the referrer fee is calculated is as follows:
	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 referrerFee + superReferrerFee;
	Ideally, if both fees would be 250 BPS, the superReferrer and the normal referrer would receive 2.5%.
	However, the result is as follows:
	amount = 1000 USDC
	referrerFeeBPS = 250 // 2.5%
	superReferrerFeeBPS = 250
	superReferrerFee = 1000 * 250 / 10000 = 25 USDC // Correct
	2.5% referrerFee = (1000 * 250 / 10000) - 25 = 25 - 25 = 0 USDC
	Total = 25 USDC (2.5%)

	This means in the scenario where both fees are 250BPS, only the superReferrer will get 2.5%.
Recommendations	Consider removing the deduction.
Comments / Resolution	

Issue_03	Parameter check within setReferrerFees is flawed
Severity	**High**

Description	The setReferrerFees function implements the following sanity check:
	uint16 totalBps = referrerContributionFeeBPS + superReferrerContributionFeeBPS + referrerInitiatorFeeBPS + superReferrerInitiatorFeeBPS;
	if (totalBps > MAX_REFERRER_BPS) {
	revert CombinedReferralFeesExceedLimit();
	}
	This check is incorrect as it takes the current variables and not the input parameters. If now accidentally the first time fees are set too high, they can never be changed again.

Recommendations	Consider using the input parameters.
Comments / Resolution	

Liquify

The Liquify contract introduces an innovative **Liquid Vesting Mechanism**, enabling projects and investors to manage, trade, and unlock liquidity throughout the vesting period. This mechanism effectively addresses common liquidity constraints inherent in traditional vesting models by issuing **liquid tokens**. These liquid tokens represent future vested tokens and can be traded even before the vesting period is complete.

Generally, the Liquify contract allows project initiators to create and manage projects with specific parameters such as project name, symbol, funding deadlines, investment caps, token price, allocation amounts, and custom vesting schedules. Initiators can set up various types of whitelists, including address-based and NFT-based whitelists ERC721, to control participant eligibility and ensure that contributions come from verified sources.

Investors can contribute to these projects by providing funds in supported payment tokens. Upon contribution, investors may receive synthetic tokens proportional to their investment amount and the project's token price, if they decide to claim these. Optionally, investors can just wait until the owner deposits realTokens and then claim these.

These synthetic tokens represent their entitlements in future vesting stages and can be traded or transferred, offering liquidity during the vesting period. Once the vesting phases are completed, investors can claim the actual tokens by redeeming their synthetic tokens.

The Liquify contract integrates a referral system through the ReferralManager contract. Referral fees are processed during contributions, rewarding both user referrers and project initiators' referrers.

In addition to managing token vesting and contributions, the Liquify contract supports cross-chain token migration. It allows users to burn tokens on one blockchain and facilitate the issuance of corresponding tokens on another chain. This feature is currently not fully supported.

The contract includes mechanisms for handling refunds, should a project need to return funds to investors. Project initiators can set a project to refund status and manage refund deposits. Investors can claim refunds for their contributions, with the contract calculating the refundable amounts based on the vesting stages and the fees involved.

Furthermore, the Liquify contract offers batch operations for approvals and transfers. Users can approve multiple token entitlements across different vesting stages in a single transaction, as well as transfer tokens in batches.

The Liquify contract uses the UUPSUpgradeable proxy pattern, which allows the contract to be upgraded.

Appendix: Project Owner Privileges:
The project owner has the following privileges:

- initiatorWithdraw:
    - Allows the project owner to withdraw any raised paymentToken
    - Can be executed as long as the project is not in the Refund stage
- updateWhitelist:
    - Allows the owner to update the whitelist state of the project, if it is in the Waiting state
    - Sets the project to the Open state
- pauseProject
    - Allows the owner to pause the project
- unpauseProject
    - Allows the owner to unpause the project
- setWaitingStatus
    - Allows the owner to set the project into the Waiting status
    - Can be invoked anytime
- initiatorWithdraw
    - Allows the owner to withdraw up to the raised amount
    - In case of refund, owner may need to pay back
- initiatorDeposit
    - Allows the owner to deposit realTokens for each phase
    - Sets the project in Distribution phase

- Increments vestingPhaseIndex with each call

- setRefundStatus
  - Sets the project in the Refund state
  - Calculates how much funds are available to re-claim and how much are excessively withdrawn

- refund
  - Fulfills the refund if there has been an excess withdrawal amount
  - Owner needs to pay back the excess withdrawal amount

Appendix: Fee Structure:

Upon the contribute function, the provided amount is decreased by the fee and amountAfterFee is then used for contribution purposes.

The following fees will be applied:

a) protocolFee: This fee can be up to 10%

b) initiatorFee: This fee can be up to 50%

Additionally, a referrer fee is being applied which is deducted from the protocolFee.

Appendix: Referral Mechanism

Upon contribution, a referral fee is applied which is calculated via a cross-contract call to the referralManager contract. The following instances of referral can be included:

a) The referrer of the initiator
b) The superReferrer of the initiator
c) The referrer of the caller
d) The superReferrer of the caller

The maximum fee for all referrals aggregated is 5%.

Appendix: Refund Mechanism

Whenever the project owner decides to not provide the project token (realToken) via initiatorDeposit (or just partially for some phases), users have the right to re-claim their payment tokens.

The setRefundStatus and refund functions are handling the fund initiation which then allows users to re-claim their tokens.

There are several distinct scenarios how a refund is initiated

Scenario A: The project owner has not provided any realTokens and has not withdrawn any paymentTokens

> setRefundStatus is invoked and amountToRefund is the equivalent amount of all provided paymentTokens. The whole amount will be claimable by users.

Scenario B: The project owner has provided one or more phases with realTokens and has not withdrawn any paymentTokens

> setRefundStatus is invoked and the owner will receive the equivalent of his provided realTokens as paymentTokens. The leftover amount will be claimable by users.

Scenario C: The project owner has not provided any realTokens and has already withdrawn any paymentTokens

> setRefundStatus is invoked and the owner has withdrawn more than he deserved. The owner needs to call refund and provide the exact excess withdrawn amount.

Scenario D: Owner has provided realToken and has already withdrawn the deserved amount

> No further funds are transferred to the owner or needed to be paid back from the owner. The project state is marked to refunded

Appendix: Normalization/Denormalization

The contract allows for paymentToken and realToken to be all decimals. In an effort to handle arithmetic operations, all amounts are normalized to 18 decimals.

If a token's decimals are below 18 it will be increased and if a token's decimals are above 18, it will be decreased.

Normalizations are happening on the following occasions:

a) minInvestmentCap
b) maxInvestmentCap
c) allocation
d) pricePerToken
e) initiatorWithdraw: normalizes desired withdrawal amount for calculations
f) initiatorDeposit: normalizes deposited amount for invariant check
g) refund: normalizes amount to refund for calculation
h) contribute: normalizes contributed amount for calculation and state changes

Denormalizations are happening on the following occasions:

a) initiatorDeposit: denormalizes the calculated deposit amount
b) refund: denormalizes the to be transferred amount
c) contribute: denormalizes referrer fee amount
d) claimRealTokens: denormalizes claimable amount
e) claimRefund: denormalizes refundable amount

Appendix: Token Amount Conversions

Each project is created with a distinct allocation which is denominated in the paymentToken. Users can also contribute directly with the desired paymentToken amount.

However, entitlements and the amount which is needed to be deposited by the project owner is denominated in the realToken.

This following conversions are done

syntheticTokenAmount = allocation / pricePerToken * 1e18

realTokenAmount = raisedAmount * 1e18 / pricePerToken

Privileged Functions

- transferOwnership
- renounceOwnership
- updateInitiators
- updatePaymentTokens
- withdrawProtocolFees
- setLiquidERC20Implementation
- setProtocolFeeBPS
- setReferralManager
- pause
- unpause

Issue_04	Project owner has full control over all deposited funds
Severity	**Governance**
Description	Currently, the project creator has full control over all deposited funds. The creator can for example immediately withdraw all paymentTokens without ever providing the realToken, provide a fake realToken or execute any other malicious actions. Moreover, the contract is under a proxy which means that the proxy admin has full control over all states and funds.
Recommendations	We do not recommend a change, this is the protocol design.
Comments / Resolution	

Issue_05	Flaw within initiatorDeposit allows for draining the contract
Severity	**High**
Description	The callInitiatorDeposit function allows the project owner to determine and deposit the realToken. A blunder within this function allows it to be called after a refund has happened which then allows resetting the state from Refund to Distributed. This blunder can be abused to drain the protocol. PoC: 1. Alice creates a new project and whitelists her own addresses 2. Alice contributes in this project with their own addresses, deposits 100_000 USDC split over 10 addresses 3. Alice calls setRefundStatus which marks the complete amount to be refunded 4. Alice calls claimRefund with their 10 addresses. 100_000 USDC are now transferred out 5. Alice now calls initiatorDeposit (which should not be allowed). The project status is now changed back to Distribution. 6. Alice now calls initiatorWithdraw, which passes because the protocol is not in the Refund state: *if (projectState.status == ProjectStatus.Refund) revert InvalidStatus(ProjectStatus.Refund, projectState.status);* 7. Alice has now successfully drained 100_000 USDC from the contract. This PoC can be arbitrarily repeated until all paymentTokens from the contract are drained.

Recommendations	Consider not allowing to call initiatorDeposit after a refund is initiated.
Comments / Resolution	

Issue_06	Sophisticated edge-case within claimRefund allows for draining the contract
Severity	**High**
Description	The claimRefund function allows users to re-claim their tokens for the phases which do not have received the realToken. This function allows to withdraw up to state.refundFunds: *if(totalRefund > state.refundFunds) revert NoRefundAvailable();* The problem is that state.refundFunds is never decreased. A sophisticated exploit including exploiting a blunder within the refund process allows for draining all paymentTokens within the protocol. PoC: 1. Alice creates a project -> allocation = 100_000 USDC -> whitelisted for own addresses -> minCap = 1 -> maxCap = large value -> 10 phases, each phase 10% 2. Alice contributes 100_000 USDC with 10 different wallets -> contract accumulates 100_000 USDC (for the PoC, we ignore the fee)

3. Alice withdraws 20_000 USDC via initiatorWithdraw

4. Alice deposits for 1 phase
-> is equivalent to 10_000USDC

5. At this point, Alice has withdrawn 20_000 USDC but only provided the equivalent of 10_000 USDC

6. Alice invokes setRefundStatus
-> amountToRefund = 90_000
-> deservedAmount = 10_000
-> raisedAmountWithdrawn = 20_000
-> excessWithdrawn = 10_000
-> state.totalRefundable = 10_000
-> availableFunds = 80_000
-> state.refundFunds = 80_000

7. At this point, the project has 80_000 USDC left to be claimed. Usually, one would need to wait until refund is invoked such that the project owner provides the remaining 10_000 USDC. However, claimRefund can already be called.

8. claimRefund is now called, note how state.refundFunds = 80_000e6 which corresponds to the 80_000 USDC in the contract. This means at this point, users could only claim up to 80_000 USDC.

The problem: state.refundFunds is NOT decreased upon calling claimRefund which allows all 10 addresses to call it and re-claim their corresponding amount.

The first address calls claimRefund and receives 9000 USDC
The second address calls claimRefund and receives 9000 USDC
....

After the 8th address, 72_000 USDC have been claimed and it should

	not further be possible to claim. However, because state.refundFunds is not decreased, the 9th/10th addresses can now claim as well which will now steal USDC from the contract. In this example 10_000 USDC was stolen. This attack can be more efficient with different numbers and the decimals of 1e6 were used for simplification. However, it essentially allows for draining the contract.
Recommendations	There are two root-causes that need to be fixed: a) claimRefund must not be callable if the refund has not been fully initiated b) state.refundFunds must be properly decreased.
Comments / Resolution	

Issue_07	Reentrancy attack within setRefundStatus allows for draining paymentToken with hooks
Severity	**High**

Description	Within setRefundStatus, the following scenario is existent:
	if (state.raisedAmountWithdrawn < deservedAmount) {
	// Initiator hasn't withdrawn enough and deserves additional funds based on vesting
	uint256 payout = deservedAmount - state.raisedAmountWithdrawn;
	project.paymentToken.safeTransfer(msg.sender, payout);
	// Set the full amount to be refunded to users and mark refund as complete
	state.refundFunds = amountToRefund;
	state.refundCompleted = true;
	emit RefundDeposit(projectId, amountToRefund);
	}
	If a project owner has deposited at least for one phase and has not yet withdrawn the exact corresponding amount, this amount is simply transferred out to the owner during the refund initiation.
	A problem in the aforementioned snippet is that the CEI pattern is violated, which allows for a reentrancy attack to drain the token (https://bailsec.io/tpost/gxcih1xoy1-checks-effects-interactions):
	project.paymentToken.safeTransfer(msg.sender, payout);
	// Set the full amount to be refunded to users and mark refund as complete
	state.refundFunds = amountToRefund;
	state.refundCompleted = true;

	This can either be done by repetitively reentering into the setRefundStatus function or by reentering into the initiatorWithdraw function.
Recommendations	Consider simply following the CEI pattern and guarding the function with a reentrancy guard.
Comments / Resolution	

Issue_08	Contract is drainable due to usage of updateWhitelist
Severity	**High**
Description	The updateWhitelist function allows for marking a project state as Open without any limitations. This can be trivially exploited by first creating a project, contribute in the own project, refund users such that the paymentToken can be withdrawn, then re-open the project via updateWhitelist and invoke the initiatorWithdraw function. PoC: a) Alice creates a project b) Alice contributes in the own project c) Alice invokes setRefundState which now allows to re-claim paymentTokens d) Alice re-claims all provided paymentTokens e) Alice invokes setWaitingStatus to mark change the status to Waiting f) Alice invokes updateWhitelist to change the status to Open g) Alice invokes initiatorWithdraw to steal paymentTokens from the contract.
Recommendations	Consider refactoring the whitelist mechanism to set it directly during the project creation.

Comments / Resolution	

Issue_09	Contract is drainable due to usage of updateWhitelist and unallowed contribution
Severity	**High**
Description	The updateWhitelist function allows for marking a project state as Open without any limitations. This can be trivially exploited by first creating a project, contributing in the project and then calling initiatorDeposit. Afterwards, the owner can then call updateWhitelist to mark the project as Open again, deposit funds to receive entitlements and then claim these entitlements which steals tokens from the contract. PoC: a) Alice creates a project with a very low price. A small amount of paymentToken will result in a large amount of realToken b) Alice deposits in the project with own addresses to increase entitlements c) Alice invokes initiatorDeposit to provide the realToken and increase vestingPhaseIndex (there are many phases, one is kept left) d) Alice claims the realToken with her entitlements e) Alice invokes setWaitingStatus and updateWhitelist to bring the project again in the open phase f) Alice now continues with contributions to re-increase entitlements g) Alice now fulfills the allocation which moves the state to Waiting h) Alice calls initiatorDeposit again to set the project to Finished (The deposit for the last phase now happens) h) Alice invokes claimRealTokens to again claim realTokens based on the new entitlements, while not having supplied any further realTokens, these are now drained from the contract.

	There are likely several more instances to exploit this blunder. However, due to time constraints we will not further investigate this vulnerability.
Recommendations	Consider refactoring the whitelist mechanism to set it directly during the project creation.
Comments / Resolution	

Issue_10	Contract is drainable due to lack of denormalization within setRefundStatus
Severity	**High**
Description	The setRefundStatus function allows the project owner to initiate a refund for the project. There are several distinct scenarios which are explained within Appendix: Refund Mechanism One scenario occurs when the project owner has deposited a realToken at least for one phase which grants him paymentTokens but these paymentTokens have not yet been withdrawn via the initiatorWithdraw function: *if (state.raisedAmountWithdrawn < deservedAmount) {* *// Initiator hasn't withdrawn enough and deserves additional funds based on vesting* *uint256 payout = deservedAmount - state.raisedAmountWithdrawn;* *project.paymentToken.safeTransfer(msg.sender, payout);* *// Set the full amount to be refunded to users and mark refund as complete*

	state.refundFunds = amountToRefund;
	state.refundCompleted = true;
	emit RefundDeposit(projectId, amountToRefund);
	
	A problem occurs because the payout amount is not denormalized which means it will transfer out tokens as 1e18 while the paymentToken may be in fact a token with 1e6. This allows for draining the whole contract.

Recommendations	Consider denormalizing the amount before the transfer.
Comments / Resolution	

Issue_11	Potential DoS of referral claiming due edge-case in cross-contract logic
Severity	**High**
Description	An edge-case within the cross-contract logic between the Liquify and ReferralManager contract which is related due to normalization/denormalization will result in a potential DoS when users attempt to claim their rightful referral fee. **PoC:** Status Quo: paymentToken has 6 decimals a) Alice contributes a specific amount of tokens such that referrersFees = 1000000500000000000 -> referrerBalance = 1000000500000000000 -> contract receives 1000000 tokens (due to denormalization)

	b) Alice contributes a specific amount of tokens (the second time) such that referrersFees = 1000000500000000000 -> referrerBalance = 2000001000000000000 -> contract received 1000000 tokens c) Referrer wants calls withdrawReferrerBalance -> denormalizedBalance = 2000001 -> contract balance = 2000000 -> call reverts
Recommendations	Consider storing the denormalized and received amounts as referrer balance.
Comments / Resolution	

Issue_12	DoS of initiatorDeposit due to wrong order of operations
Severity	**High**

Description	The initiatorDeposit function caches the realToken as follows:
	
	IERC20 sc_realToken = projectState.realTokensAddress;
	
	At this point however, projectState.realTokensAddress is not yet set, as it is only set afterwards:
	
	if (projectState.vestingPhaseIndex == 0) {
	if (address(realToken) == address(0)) revert InvalidAddress();
	projectState.status = ProjectStatus.Distributing;
	projectState.realTokensAddress = realToken;
	}
	
	Thus, the .decimals call will result in calling .decimals on address(0).

Recommendations	Consider caching realToken only once it has been set.
Comments / Resolution	

Issue_13	Contract is drainable due to usage of setWaitingStatus
Severity	**High**
Description	The setWaitingStatus function allows the project owner to set a project to the Waiting state any time during the lifecycle. This can be exploited several ways, we will just describe one way below: a) Alice creates a project b) Alice contributes in the project c) Alice invokes setRefundState d) Alice claims the refund e) Alice invokes setWaitingState which moves the state from Refund to Waiting f) Alice invokes initiatorWithdraw, which is now possible because the project is not anymore in the refunded state: https://github.com/Liquify-labs/sc/blob/791e0fc4eaa4e546cee9df1e4b9581b23badff9d/src/Liquify.sol#L283 this will transfer the amount of paymentTokens which have already been refunded.
Recommendations	Consider removing the setWaitingStatus function as there are several exploit iterations which can be abused due to it.
Comments / Resolution	

Issue_14	Change of LIQUIDERC20_IMPLEMENTATION will DoS running projects
Severity	**High**

Description	The setLiquidERC20Implementation function allows for changing LIQUIDERC20_IMPLEMENTATION. If this variable is ever changed all running projects will point to a wrong synthetic token:
	address cloneTokenAddress = Clones.predictDeterministicAddress(address(LIQUIDERC20_IMPLEMENTATION), keccak256(abi.encodePacked(projectId, round)), address(this));
	LiquidERC20 cloneToken = LiquidERC20(cloneTokenAddress);
	This will DoS all operations related to the synthetic token.

Recommendations	Consider storing the corresponding LIQUIDERC20_IMPLEMENTATION upon project creation.
Comments / Resolution	

Issue_15	Refund will DoS claiming
Severity	**High**
Description	The contract works in such a way that it is possible for users to receive a part of their contribution as realToken and on the other hand re-claim the paymentToken for any undistributed phases. If for example a project with 2 phases is created, a contributor will receive entitlements for all two phases. If the project owner now calls initiatorDeposit only one time, the user is entitled to receive the realToken for phase 1 and claim a refund for phase 2. In that scenario, the owner will invoke the setRefundStatus function which enables refunds: *state.status = ProjectStatus.Refund;* However, at the same time this will also disable claimRealTokens: *if (state.status != ProjectStatus.Distributing && state.status != ProjectStatus.Finished)* *revert InvalidStatus2(ProjectStatus.Distributing, ProjectStatus.Finished, state.status);*
Recommendations	Consider rewriting this approach, allowing to still claim within the Refund phase will likely implement undesired side-effects.
Comments / Resolution	

Issue_16	Malicious user can permanently DoS claiming/refund via batchSetWaiting
Severity	**High**

Description	The batchSetWaiting function allows any user to move the project to the Waiting state once the deadline has passed:
	
	function batchSetWaiting(uint256[] calldata projectIds) external whenNotPaused {
	for (uint256 i = 0; i < projectIds.length; i++) {
	if (block.timestamp > projectDetails[projectIds[i]].fundingDeadline) {
	projectStates[projectIds[i]].status = ProjectStatus.Waiting;
	emit ProjectWaiting(projectIds[i]);
	}
	}
	}
	
	This function can even be called once the project has been moved to the Distribution or Refund phase which then effectively prevents users from calling claimRefund or claimRealTokens

Recommendations	Consider making this function permissioned for the project owner.
Comments / Resolution	

Issue_17	Incorrect refund amount calculation
Severity	**High**
Description	The setRefundStatus function calculates first how many tokens a projectOwner deserves and then allocates it via the deservedAmount variable as follows: *uint256 deservedAmount = state.raisedAmount + state.raisedFees - amountToRefund;* Afterwards, several different checks are being executed, all different states are being reflected within **Appendix: Refund Mechanism**. The problem is that these checks are only based on: *state.raisedAmountWithdrawn* While in fact the following amount was withdrawn: *projectState.raisedAmountWithdrawn + projectState.totalFeesWithdrawn* This has many unexpected side-effects which range from accidentally transferring out too much tokens to the projectOwner and a DoS of the last claims
Recommendations	Consider aggregating raisedAmountWithdrawn and totalFeesWithdrawn and using this aggregated variable.
Comments / Resolution	

Issue_18	walletAmount is not entitled to burnedEntitlements
Severity	**High**

Description	The burnTokensForMigration function allows a contributor to burn synthetic tokens or entitlements for migration. This is done as follows:
	// 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];
	As one can see, the burned walletAmount is not added to burnedEntitlements which means that it is nowhere reflected how much tokens a user has burned.

Recommendations	Consider implementing a separate mapping which reflects how much tokens have been burned. If however the migration is only be depending on the event, this issue can be entirely removed.
Comments / Resolution	

Issue_19	initiatorDeposit will never work in edge-cases due to truncation
Severity	**High**
Description	The initiatorDeposit function allows the project creator to deposit the necessary realToken amount for the corresponding raised investment amount. In an effort to prevent transfer tax tokens from being submitted, the following check is present: *if (normalizedAmountReceived != expectedDeposit) {* *revert InsufficientDeposit(expectedDeposit, normalizedAmountReceived);* *}* This check reverts under a specific edge-case and thus prevents a project from being fulfilled. Problem: If expectedDeposit is 1.9999...9e18, and realToken has 6 decimals, the calculation will be as follows: a) expectedDeposit = 1999999999999999999 b) calculatedDepositAmount = expectedDeposit * 1e18 / pricePerToken -> 1999999999999999999 * 1e18 / 1e18 -> calculatedDepositAmount = 1999999999999999999 c) denormalizedExpectedDeposit = 1999999 Now the contract transfers 1999999 from the owner and executes the following check: uint256 normalizedAmountReceived =

	normalizeTokenAmount(sc_realToken.balanceOf(address(this)) - balanceBefore, realTokenDecimals);
	-> normalizedTokenAmount = 1999999000000000000
	if (normalizedAmountReceived != expectedDeposit) {
	revert InsufficientDeposit(expectedDeposit, normalizedAmountReceived);
	}
	This will now always revert because it attempts to compare 1999999000000000000 to 1999999999999999999
	***THIS IS JUST AN EXAMPLE ISSUE WHICH WILL STILL BE EXISTENT WHEN THE INVARIANT CHECK IS EXECUTED IN THE CORRECT DENOMINATION**

Recommendations	Consider adjusting this check to ensure that expectedDeposit zeroes out all truncated digits.
Comments / Resolution	

Issue_20	Division before multiplication will result in truncated totalSupply
Severity	**High**
Description	The createProject function mints the necessary amount of synthetic tokens to the contract in an effort to support trading of the synthetic token. The calculation for the amount to be minted is as follows: *totalSupply = allocation / pricePerToken* *mint(totalSupply * 1e18)* This is incorrect and the classical division before multiplication issue which results in an erroneous output amount. **PoC 1:** Status Quo: allocation = 9999999999999999999 price = 1000e18 a) Calculate totalSupply: -> totalSupply = allocation / pricePerToken -> 9999999999999999999 / 1000e18 -> totalSupply = 0 b) Mint 0*1e18 **PoC 2:** Status Quo: allocation = 9999999999999999999 price = 1e18

	a) Calculate totalSupply: -> totalSupply = allocation / pricePerToken -> 9999999999999999999 / 1e18 -> totalSupply = 9 b) Mint 9*1e18 However, the contract expects to receive 9.999.999e18
Recommendations	Consider calculating totalSupply as: allocation * 1e18 / pricePerToken and removing the * 1e18 during the minting. This calculation however still inherently truncates values if the pricePerToken is higher than 1e18.
Comments / Resolution	

Issue_21	Invariant check within initiatorDeposit is based on wrong denomination
Severity	**High**

Description	The initiatorDeposit function has the following invariant check at the end of the function which ensures provided amount of realToken matches the needed amount:
	if (normalizedAmountReceived != expectedDeposit) {
	revert InsufficientDeposit(expectedDeposit, normalizedAmountReceived);
	}
	This check compares expectedDeposit, which is denominated in the paymentToken:
	*uint256 expectedDeposit = (projectState.raisedAmount * vestingPercentage) / MAX_BPS;*
	with normalizedAmountReceived, which is denominated in the realToken.

Recommendations	Consider comparing denormalizedExpectedDeposit with amountReceived (without normalizing amountReceived)
Comments / Resolution	

Issue_22	Incorrect vestingPhaseIndex check will prevent claiming from last phase
Severity	**High**
Description	Input value round Is checked when calling claimRealTokens and is expected not to be higher or equal the current state.vestingPhaseIndex *if (round >= state.vestingPhaseIndex)* *revert InvalidClaimRealTokensRound(round, state.vestingPhaseIndex);* This will prevent users from claiming realTokens for the last round as state.vestingPhaseIndex starts with index 0. **PoC:** Project has two rounds [0,1] a) Owner calls initiatorDeposit for both rounds, and state.vestingPhaseIndex is incremented twice before state is set to finished. *state.vestingPhaseIndex == 1* b) User calls claimRealTokens for both rounds [0,1] c) At the second loop, *round == state.vestingPhaseIndex* so it reverts
Recommendations	Consider changing *round >= state.vestingPhaseIndex to round > state.vestingPhaseIndex.* Eventual unexpected side-effects will be considered during the re-audit.
Comments / Resolution	

Issue_23	Usage of uint8 during contribute can result in revert
Severity	**Medium**
Description	A project creator can create projects with up to 10_000 rounds (theoretically), which is limited by the BPS value. Within contribute, uint8 is used to determine the looping index, which is up to 255. In the scenario where a project has >255 phases, this will revert.
Recommendations	Consider using a higher uint type.
Comments / Resolution	

Issue_24	ERC1155/ERC721 whitelist can be bypassed
Severity	**Medium**
Description	The contribute function implements a whitelist mechanism which can be as follows: a) Whitelist mechanism via address b) Whitelist mechanism via ERC721 or ERC1155 c) Whitelist mechanism via ERC721 or ERC1155 and via address In the scenario where the whitelist mechanism is only based on ERC721 or ERC1155, this check can trivially bypassed by simply transferring the token to a different address after participation: *require(IERC721(state.nftWhitelistContract).ownerOf(tokenId) == msg.sender, "No qualifying ERC-721 tokens owned.");* *require(!state.registeredNFTs[msg.sender][tokenId], "NFT*

	already used for investment."); *state.registeredNFTs[msg.sender][tokenId] = true;*
Recommendations	Consider marking the tokenId as used without the msg.sender linkage.
Comments / Resolution	

Issue_25	Possible underflow revert due to blunder within fee configuration
Severity	**Medium**
Description	The contribute function allocates protocolFees as follows: *protocolFees[project.paymentToken] += protocolFee - referrersFees;* This can result in a revert if referrersFees are larger than protocolFees. Notably, referrersFees are based on the amount instead of the protocolFee which can legitimately result in such an issue. Moreover, there is no explicit enforcement that protocolFees must be larger than referrersFees.
Recommendations	Consider either calculating referrersFees depending on the protocolFees or handling the scenario where referrersFees can exceed protocolFees.
Comments / Resolution	

Issue_26	Incorrect amount check within contribute will result in user allocation never being reached
Severity	**Medium**

Description	The contribute has the following amount enforcement:
	if (amount < project.minInvestmentCap && amount != maxRemainingInvestment) {
	revert InvestmentBelowMinimum();
	}
	If the amount is below the minInvestmentCap and does not match with maxRemainingInvestment amount, a contribution reverts. This special edge-case handles the scenario where there is just a small leftover amount needed to finalize the project.
	In itself, this check is correct. However, a fee is deducted from the amount and amountAfterFee is then used to increase the investment of the user:
	state.investments[msg.sender] += amountAfterFee;
	Therefore, a user will never be able to precisely reach the allowed allocation.

Recommendations	Consider rewriting this and apply the amount enforcement on amountAfterFee.
Comments / Resolution	

Issue_27	tokenAllocation calculation within contribute can result in large precision loss
Severity	**Medium**
Description	The tokenAllocation calculation within the contribute function calculates the entitlement size a user should receive based on the provided paymentToken. The calculation is done as follows: *uint256 tokenAllocation = amountAfterFee * 10 ** 18 / project.pricePerToken;* This calculation can result in a large precision loss depending on the pricePerToken, which would then result in less tokens than expected for users. The lower amountAfterFee and the higher pricePerToken, the higher will be the loss for the user.
Recommendations	Consider implementing a slippage parameter which prevents unexpected outcomes.
Comments / Resolution	

Issue_28	Amount truncation can result in DoS
Severity	**Medium**
Description	The contract has several spots where token amounts are truncated. Most notably this is present whenever amounts are normalized from higher decimals to 1e18 or denormalized from 1e18 to lower decimals. In the first scenario, a truncation from 1e21 to 1e18 can result in truncating 3 digits and in the latter scenario a truncation from 1e18 to 1e6 can result in truncating 12 digits. PoC: 9.9e18 = 9999999999999999999 denormalized back to 6 decimals: 999999 Furthermore, during all percentage calculations the truncation can be up to (divisor-1). There are several scenarios where the contract could result in a DoS due to truncation. For example whenever all users attempt to claim refunds, real tokens or synthetic tokens, the contract expects that exactly these amounts are existing. This can become problematic for the last claimer if the contract for example does not hold the exact amount due to truncation: state.realTokensAddress.safeTransfer(msg.sender, denormalizedTotalRealTokens); Such a call could result in a DoS.

Recommendations	Consider the following steps: a) Rounding up for amounts which are provided -> Minting of synthetic tokens -> Deposit of realTokens -> Repayment of paymentTokens in excess withdrawal scenario b) Incorporate fuzz testing to ensure these scenarios can never happen
Comments / Resolution	

Issue_29	Lack of state.refundFunds decrease allows for temporarily occupying funds from other projects
Severity	**Medium**
Description	On top of the issues which allow for draining funds from the contract due to the lack of state.refundFunds decrease, this blunder allows for temporarily withdrawing tokens which are allocated to other protocols (in the scenario where the project owner intends to pay back any excessively withdrawn funds).
Recommendations	Consider correctly decreasing state.refundFunds and only allow claimRefund to be called when all funds have been repaid by the project owner.
Comments / Resolution	

Issue_30	Blacklisted project owner will result in refund DoS
Severity	**Medium**

Description	The setRefundStatus function attempts to transfer any deserved amount to the project owner, which hasn't been withdrawn yet:
	if (state.raisedAmountWithdrawn < deservedAmount) {
	// Initiator hasn't withdrawn enough and deserves additional funds based on vesting
	uint256 payout = deservedAmount - state.raisedAmountWithdrawn;
	project.paymentToken.safeTransfer(msg.sender, payout);
	
	// Set the full amount to be refunded to users and mark refund as complete
	state.refundFunds = amountToRefund;
	state.refundCompleted = true;
	emit RefundDeposit(projectId, amountToRefund);
	
	If the project owner is blacklisted from the paymentToken, the transfer will never work, effectively DoS'ing the refunds.
	
	Furthermore, it is also not possible to call initiatorWithdraw due to the same issue. On top of that, if the owner withdrew too much and is now enforced to refund the excess withdrawn amount, this would also not work:
	
	uint256 denormalizedTotalToRefund = denormalizeTokenAmount(amountToRefund, paymentTokenDecimals);
	paymentToken.safeTransferFrom(msg.sender, address(this), denormalizedTotalToRefund);

Recommendations	First of all, the initiatorWithdraw function should expose a “to” parameter which serves as recipient address.

	Furthermore, it must be considered if it makes sense to expose a whitelisted addresses mapping which allows the project owner to add an address that can invoke refund on behalf of the owner.
Comments / Resolution	

Issue_31	Potentially permanently stuck funds due to inactivity of project owner
Severity	**Medium**
Description	If a project owner suddenly becomes inactive (without turning malicious), this will result in all provided paymentTokens residing permanently in the contract.
Recommendations	Consider either implementing a recover function (which can be done because the contract is under a proxy anyways which means owner has full privileges) or implementing a structure which allows permissionless refunding tokens if a project has not received any deposits after a certain time.
Comments / Resolution	

Issue_32	Normalization of pricePerToken can result in loss for project creator
Severity	**Low**
Description	During the createProject function, the pricePerToken parameter must be from the same decimals as the paymentToken. In the scenario where this is 21 decimals the last 3 digits from pricePerToken will be truncated, resulting in a loss if any of the last 3 digits is non-zero.
Recommendations	We do not recommend a change.
Comments / Resolution	

Issue_33	Missing check for zero percentages
Severity	**Low**
Description	The createProject function lacks a check which ensures that phase percentages are non-zero. This can result in unexpected side-effects.
Recommendations	Consider implementing such a check.
Comments / Resolution	

Issue_34	Normalization can result in loss for contributor
Severity	**Low**
Description	Within the contribute function, the _amount is normalized to 18 decimals. This will result in a dust loss for the contributor if the token has 21 decimals and the last 3 digits of the provided amount are non-zero.
Recommendations	We do not recommend a change
Comments / Resolution	

Issue_35	state.investments is erroneously reset
Severity	**Low**
Description	Within claimRefund, it is possible to claim a refund for all phases which have not yet experienced a deposit. This means at the same time, for some phases users can have valid entitlements which allows them to claim tokens. However, despite this fact, the investment mapping is still reset: state.investments[msg.sender] = 0; While this does not have a negative impact, it is still incorrect to do.
Recommendations	Consider if it makes sense to decrease investments only by the amount entitlements which are reclaimed. However, that may result in truncation.
Comments / Resolution	

Issue_36	Users can be prevented from consuming allocation during edge-case
Severity	**Low**

Description	Within the contribute function, the following allocation check is existent:
	`if (amount < project.minInvestmentCap && amount != maxRemainingInvestment) {`
	`revert InvestmentBelowMinimum();`
	`}`
	`if (amount > maxRemainingInvestment) revert MaxInvestmentCapError(state.investments[msg.sender], project.maxInvestmentCap);`
	A check is being executed which allows a user to deposit only below the minInvestmentCap if the deposit amount is equal to the remaining deposit.
	In the scenario where the remaining global allocation is smaller than maxRemainingInvestment, the user can never consume the leftover allocation.

Recommendations	We do not recommend a change as this would greatly increase complexity. However, this issue should be noted.
Comments / Resolution	

Issue_37	Dust can be stolen within initiatorWithdraw due to truncation
Severity	**Low**
Description	The initiatorWithdraw function allows the project owner to withdraw all raised paymentTokens. This function executes the following normalization: *uint256 normalizedAmount = normalizeTokenAmount(amount, IERC20Metadata(address(paymentToken)).decimals());* However, at the end still the provided amount parameter is withdrawn: *paymentToken.safeTransfer(msg.sender, amount);* Since all checks are done using the normalizedAmount, dust can be stolen in the scenario where the token is a token with > 18 decimals.
Recommendations	Consider zeroing out all digits which are truncated during the normalization and only transfer out the adjusted amount.
Comments / Resolution	

Issue_38	Possibility of zero contribution
Severity	**Low**
Description	The contribute function does not prevent zero amount contributions. This can result in unexpected side-effects.
Recommendations	Consider preventing zero amount contributions.
Comments / Resolution	

Issue_39	Initiators can create multiple projects
Severity	**Low**
Description	The createProject function allows whitelisted addresses to create projects for fundraising purposes. Currently, there is no limitation in how many projects can be created by whitelisted addresses.
Recommendations	Consider incorporating a nonce-scheme which limits the amount of projects which can be created. Alternatively this can be acknowledged if it is desired.
Comments / Resolution	

Issue_40	Transfer-tax check is not enforced for paymentToken
Severity	**Low**
Description	The updatePaymentTokens function allows for adding and removing paymentTokens. However, there is currently no transfer-tax check executed which will result in problems if ever a transfer tax token is being added as paymentToken.
Recommendations	Consider executing such a check. One can simply do a test transfer and a before-after check.
Comments / Resolution	

Issue_41	Griefing: Malicious user can prevent finalization before deadline
Severity	**Informational**

Description	Within the contribute function, the project is moved to the waiting state once the exact desired allocation has been raised:
	if (state.raisedAmount == project.allocation)
	state.status = ILiquify.ProjectStatus.Waiting;
	It is possible for users to prevent this state from happening to contribute a small amount while frontrunning a legitimate deposit which would result in raisedAmount = allocation.

Recommendations	We do not recommend a change.
Comments / Resolution	

Issue_42	Redundant declaration of totalClaimed
Severity	**Informational**

Description	The claimSyntheticTokens function declares the totalClaimed variable but never returns it:
	uint256 totalClaimed;
	...
	totalClaimed += tokensToClaim;

Recommendations	Consider returning totalClaimed.
Comments / Resolution	

Issue_43	Violation of checks-effects-interactions pattern
Severity	**Informational**

Description	Within the claimSyntheticTokens function, the cei pattern is violated:
	IERC20(tokenClone).safeTransfer(msg.sender, tokensToClaim);
	projectState.tokenEntitlements[msg.sender][stage].entitlements = 0;
	totalClaimed += tokensToClaim;

Recommendations	Consider following the cei pattern.
Comments / Resolution	