QuillAudits

AUDIT REPORT

August 2026

For

Table of Content

Executive Summary

Project Name	SL Labs
Protocol Type	ERC-20 Token & Linear Vesting
Project URL	https://www.savethelife.io/
Overview	The reviewed system mints a fixed-supply ERC-20 token and distributes it across ten allocation buckets: three are transferred outright at launch, one is burned, and six are escrowed in a vesting contract that releases them on a cliff-plus-linear schedule over 48 months.
Audit Scope	The scope of this audit was to analyze the SL Labs Smart Contracts for quality, security, and correctness.
Source Code link	https://github.com/lch5482/SL_TOKEN
Contracts in Scope	contracts/SLToken.sol contracts/SLVesting.sol
Branch	main
Commit Hash	c2e7899
Language	Solidity
Blockchain	EVM (opBNB)
Method	Manual Analysis, Threat Modelling
Review 1	30th-31st July, 2026
Updated Code Received	4th Aug, 2026
Review 2	4th Aug, 2026
Fixed In	13dad9d

Executive Summary

✅ Verify the Authenticity of Report on QuillAudits Leaderboard:
https://www.quillaudits.com/leaderboard

Number of Issues per Severity

Critical	0 (0%)
High	2 (25%)
Medium	3 (38%)
Low	1 (12%)
Informational	2 (25%)

Severity

Issues	Critical	High	Medium	Low	Informational
Open	0	0	0	0	0
Acknowledged	0	0	0	0	0
Partially Resolved	0	0	0	0	0
Resolved	0	**2**	**3**	**1**	**2**

Summary of Issues

Issue No.	Issue Title	Severity	Status
1	Owner Can Redirect Vested Tokens To An Arbitrary Address At Any Time	High	Resolved
2	Owner Can Create A Vesting Schedule That Is Already Unlocked At Creation	High	Resolved
3	Schedules Are Created Without Any Solvency Check Against The Contract Balance	Medium	Resolved
4	release() Is All-Or-Nothing And A Shortfall Strands The Entire Residual Balance	Medium	Resolved
5	Schedule Time Parameters Are Unvalidated And Permanently Uncorrectable	Medium	Resolved
6	Surplus Or Misdirected Tokens Are Permanently Locked With No Rescue Path	Low	Resolved
7	TOTAL_SUPPLY Constant Diverges From Live totalSupply()	Informational	Resolved
8	SLToken Inherits Ownable With No Owner-Gated Function	Informational	Resolved

Checked Vulnerabilities

- Access Management
- Arbitrary write to storage
- Centralization of control
- Ether theft
- Improper or missing events
- Logical issues and flaws
- Arithmetic Computations Correctness
- Race conditions/front running
- SWC Registry
- Re-entrancy
- Timestamp Dependence
- Gas Limit and Loops
- Exception Disorder
- Gasless Send
- Use of tx.origin
- Malicious libraries
- Compiler version not fixed
- Address hardcoded
- Divide before multiply
- Integer overflow/underflow
- ERC's conformance
- Dangerous strict equalities
- Tautology or contradiction
- Return values of low-level calls
- Missing Zero Address Validation
- Private modifier
- Revert/require functions
- Multiple Sends
- Using suicide
- Using delegatecall

- [x] Upgradeable safety
- [x] Style guide violation
- [x] Using throw
- [x] Unsafe type inference
- [x] Using inline assembly
- [x] Implicit visibility level

Techniques and Methods

Throughout the audit of smart contracts, care was taken to ensure:

- The overall quality of code.
- Use of best practices.
- Code documentation and comments, match logic and expected behavior.
- Token distribution and calculations are as per the intended behavior mentioned in the whitepaper.
- Implementation of ERC standards.
- Efficient use of gas.
- Code is safe from re-entrancy and other vulnerabilities.

The following techniques, methods, and tools were used to review all the smart contracts:

Structural Analysis

In this step, we have analyzed the design patterns and structure of smart contracts. A thorough check was done to ensure the smart contract is structured in a way that will not result in future problems.

Static Analysis

A static Analysis of Smart Contracts was done to identify contract vulnerabilities. In this step, a series of automated tools are used to test the security of smart contracts.

Code Review / Manual Analysis

Manual Analysis or review of code was done to identify new vulnerabilities or verify the vulnerabilities found during the static analysis. Contracts were completely manually analyzed, their logic was checked and compared with the one described in the whitepaper. Besides, the results of the automated analysis were manually verified.

Gas Consumption

In this step, we have checked the behavior of smart contracts in production. Checks were done to know how much gas gets consumed and the possibilities of optimization of code to reduce gas consumption.

Tools and Platforms Used for Audit

Manual analysis, threat modelling, functional testing.

Types of Severity

Every issue in this report has been assigned to a severity level. There are five levels of severity, and each of them has been explained below.

Critical: Immediate and Catastrophic Impact

Critical issues are the ones that an attacker could exploit with relative ease, potentially leading to an immediate and complete loss of user funds, a total takeover of the protocol's functionality, or other catastrophic failures. Critical vulnerabilities are non-negotiable; they absolutely must be fixed.

High (H): Significant Risk of Major Loss or Compromise

High-severity issues represent serious weaknesses that could result in significant financial losses for users, major malfunctions within the protocol, or substantial compromise of its intended operations. While exploiting these vulnerabilities might require specific conditions to be met or a moderate level of technical skill, the potential damage is considerable. These findings are critical and should be addressed and resolved thoroughly before the contract is put into the Mainnet.

Medium (M): Potential for Moderate Harm Under Specific Circumstances

Medium-severity bugs are loopholes in the protocol that could lead to moderate financial losses or partial disruptions of the protocol's intended behavior. However, exploiting these vulnerabilities typically requires more specific and less common conditions to occur, and the overall impact is generally lower compared to high or critical issues. While not as immediately threatening, it's still highly recommended to address these findings to enhance the contract's robustness and prevent potential problems down the line.

Low (L): Minor Imperfections with Limited Repercussions

Low-severity issues are essentially minor imperfections in the smart contract that have a limited impact on user funds or the core functionality of the protocol. Exploiting these would usually require very specific and unlikely scenarios and would yield minimal gain for an attacker. While these findings don't pose an immediate threat, addressing them when feasible can contribute to a more polished and well-maintained codebase.

Informational (I): Opportunities for Improvement, Not Immediate Risks

Informational findings aren't security vulnerabilities in the traditional sense. Instead, they highlight areas related to the clarity and efficiency of the code, gas optimization, the quality of documentation, or adherence to best development practices. These findings don't represent any immediate risk to the security or functionality of the contract but offer valuable insights for improving its overall quality and maintainability. Addressing these is optional but often beneficial for long-term health and clarity.

Types of Issues

Open

Security vulnerabilities identified that must be resolved and are currently unresolved.

Resolved

These are the issues identified in the initial audit and have been successfully fixed.

Acknowledged

Vulnerabilities which have been acknowledged but are yet to be resolved.

Partially Resolved

Considerable efforts have been invested to reduce the risk/impact of the security issue, but are not completely resolved.

Severity Matrix

Impact

Likelihood	High	Medium	Low
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

Impact

- **High** - leads to a significant material loss of assets in the protocol or significantly harms a group of users.
- **Medium** - only a small amount of funds can be lost (such as leakage of value) or a core functionality of the protocol is affected.
- **Low** - can lead to any kind of unexpected behavior with some of the protocol's functionalities that's not so critical.

Likelihood

- **High** - attack path is possible with reasonable assumptions that mimic on-chain conditions, and the cost of the attack is relatively low compared to the amount of funds that can be stolen or lost.
- **Medium** - only a conditionally incentivized attack vector, but still relatively likely.
- **Low** - has too many or too unlikely assumptions or requires a significant stake by the attacker with little or no incentive.

High Severity Issues

Owner Can Redirect Vested Tokens To An Arbitrary Address At Any Time

Resolved

Path

`SLVesting.sol`

Function Name

`updateBeneficiary()`

Description

updateBeneficiary overwrites s.beneficiary for any existing schedule with no timelock, no acceptance step, and no consent from the outgoing beneficiary. It does not settle the accrued but unclaimed balance first, and release resolves the destination at execution time:

```
s.beneficiary = newBeneficiary; // updateBeneficiary
token.safeTransfer(s.beneficiary, amount); // release
```

Impact

The owner can seize the entire unreleased balance of every schedule. Because release is permissionless and reads the beneficiary at execution time, the owner can also front-run a beneficiary's own claim, so the victim's gas pays to deliver the tokens to the owner's address. A beneficiary has no reliable defence beyond claiming continuously.

Recommendation

Settle the accrued balance to the outgoing beneficiary before reassigning. Transferring vesting ownership to a timelocked multisig immediately after deployment is required regardless of this change.

High Severity Issues

Owner Can Create A Vesting Schedule That Is Already Unlocked At Creation

Resolved

Path

`SLVesting.sol`

Function Name

`createSchedule()`

Description

createSchedule validates the id, the beneficiary, the total and tgeBps, but it never bounds start against the current block. The value is stored verbatim, and vestedAmount reads it as the origin of the whole schedule, so any schedule created with a start earlier than the creation block is treated as though it had been running since that date:

```
start: start; // createSchedule, stored unchecked
if (timestamp < s.start) return 0; // vestedAmount, the only check on start
```

The cliff is measured from that origin, so a staleness of at least the cliff length serves the cliff immediately, and every further second of staleness elapses a proportional share of the linear ramp. The amount claimable in the creating block is therefore the full TGE tranche plus remaining multiplied by the elapsed fraction of duration, and once the staleness reaches cliff plus duration the entire allocation is claimable at once. This affects every schedule the contract can hold, not any particular allocation, and it requires no extreme or sentinel value: a start stale by days already converts directly into unlocked value. The same missing check leaves start unbounded above, so a TGE expressed in milliseconds rather than seconds, or any other mistyped future date, is accepted as a valid uint64 tens of thousands of years out, releasable stays 0 permanently and release reverts "nothing to release" with no panic or event to signal it.

Impact

Tokens intended to be locked for years are transferred irreversibly in the block the schedule is created, and the mistake cannot be undone: exists is write-once so recreating the id reverts "exists", updateBeneficiary only redirects value that is already claimable, and release is permissionless with no timelock, so the beneficiary can claim immediately after creation before the owner has any opportunity to react. In the opposite direction a mistyped future start silently locks the allocation permanently, since the contract has no sweep, revoke or amend path.

Owner Can Create A Vesting Schedule That Is Already Unlocked At Creation *(continued)*

Recommendation

Bound start on both sides inside createSchedule, and keep the bound as an on-chain require rather than a convention in the deployment script:
uint64 constant MAX_START_DELAY = 365 days;
require(start >= uint64(block.timestamp), "start in past");
require(start <= uint64(block.timestamp) + MAX_START_DELAY, "start too far");

Medium Severity Issues

Schedules Are Created Without Any Solvency Check Against The Contract Balance

Resolved

Path

`SLVesting.sol`

Function Name

`createSchedule()`

Description

createSchedule validates only !exists, beneficiary != 0, total > 0, and tgeBps <= 10000. It never reads token.balanceOf(address(this)), and the contract keeps no accumulator of committed totals, so the sum of schedule claims can silently exceed the tokens actually held.

All six schedules draw from one undivided balance, and the deployment script funds each bucket in a transfer transaction separate from the createSchedule transaction that registers it.

Impact

An over committed pool is absorbed entirely by whichever beneficiary claims last, whose safeTransfer reverts permanently. Because the contract exposes no revoke, sweep, or rescue function, that beneficiary's allocation is unrecoverable.

A failure between the funding transfer and the schedule registration produces the same outcome silently, with no on-chain signal that the books do not balance.

Recommendation

Track committed totals and require the contract to be funded at creation time.

Medium Severity Issues

release() Is All-Or-Nothing And A Shortfall Strands The Entire Residual Balance

Resolved

Path

`SLVesting.sol`

Function Name

`release()`

Description

release() takes no amount parameter and transfers the full accrued amount with no clamp against the contract's actual balance:

```
uint256 amount = vestedAmount(id, uint64(block.timestamp)) - s.released;
require(amount > 0, "nothing to release");
s.released += amount;
token.safeTransfer(s.beneficiary, amount);
```

Impact

A beneficiary whose accrued amount exceeds the pooled balance by even one wei receives zero rather than the residual. A small accounting shortfall is amplified into a permanent freeze of a much larger balance, and no rescue function exists to recover it.

Recommendation

Pay the lesser of the accrued amount and the available balance so a shortfall degrades gracefully instead of freezing.

Medium Severity Issues

Schedule Time Parameters Are Unvalidated And Permanently Uncorrectable

Resolved

Path

`SLVesting.sol`

Function Name

`createSchedule()`

Description

start, cliffSeconds, and durationSeconds are accepted with no bounds of any kind. require(!schedules[id].exists) then blocks any correction, and the contract exposes no schedule-amendment, revoke, or sweep function. The deployment script reads start from the TGE_TIMESTAMP environment variable without validating it.

Impact

A millisecond-scaled timestamp pushes every unlock beyond any realistic horizon and permanently strands the affected allocation.

The inverse error, a past start combined with duration = 0, converts an advertised multi-year lockup into an instant full unlock that a mempool observer can back-run through the permissionless release before the owner can react. Neither outcome is correctable after creation.

Recommendation

Bound the inputs at creation, and consider allowing the owner to amend a schedule that has not yet begun vesting.

Low Severity Issues

Surplus Or Misdirected Tokens Are Permanently Locked With No Rescue Path

Resolved

Path

`SLVesting.sol`

Function Name

`contract-level`

Description

Lifetime outflow from the vesting contract is hard-capped at the sum of schedule totals, because release can never pay more than total for a given identifier. The contract has no sweep, withdraw, revoke, or rescue function, and inherits OpenZeppelin's renounceOwnership un-overridden.

Any balance beyond scheduled claims, whether from a failed deployment step, double funding, or a user transferring to the published vesting address, is unreachable by every caller including the owner.

Impact

Surplus tokens are permanently lost. Note the sequencing risk: fixing the solvency gap with a strict invariant, without simultaneously adding a sweep, converts today's recoverable surplus into a provable permanent lock.

Recommendation

Add an owner-only sweep restricted to the provable surplus.

Informational Severity Issues

TOTAL_SUPPLY Constant Diverges From Live totalSupply()

Resolved

Path

`SLToken.sol`

Function Name

`TOTAL_SUPPLY()`

Description

TOTAL_SUPPLY is a public constant fixed at 2,000,000,000e18, while ERC20Burnable makes the real supply strictly decreasing. The deployment script burns the Protocol Reserve allocation during deployment, so from the first post-deploy block the constant reads 2.0e27 while totalSupply() reads 1.9e27.

This is a permanent 5% divergence that grows with every subsequent holder burn. The project's own verification script hardcodes the same 2,000,000,000 figure when computing the burned amount.

Impact

Any exchange, explorer, or integrator computing circulating supply or governance thresholds from the public constant silently inherits the wrong denominator.

Recommendation

Rename the constant to INITIAL_SUPPLY to remove the ambiguity, and have off-chain tooling read circulating supply from totalSupply() rather than from the constant.

Informational Severity Issues

SLToken Inherits Ownable With No Owner-Gated Function

Resolved

Path

`SLToken.sol`

Function Name

`constructor()`

Description

SLToken declares Ownable and assigns an owner at construction, but no onlyOwner function exists anywhere in the contract. The owner has no capability whatsoever, yet owner(), transferOwnership, and renounceOwnership remain live in the ABI.

The deployment script also passes the same address as both initialHolder and owner, conflating two unrelated roles for no functional reason.

Impact

Explorers and automated risk scanners commonly read a live owner as retained mint or pause authority, and a future derived contract would inherit an already-set owner rather than being forced into a deliberate access-control decision.

Recommendation

Remove Ownable from SLToken. If it is retained as a placeholder for future functionality, document that intent and call renounceOwnership() immediately after deployment.

Functional Tests

SLToken

- constructor mints the entire 2,000,000,000 SL supply to initialHolder
- constructor assigns initialHolder as the owner
- constructor reverts when initialHolder is the zero address
- TOTAL_SUPPLY constant equals 2,000,000,000e18
- name, symbol and decimals return "SL Token", "SL" and 18
- no mint path exists after construction
- transfer moves the exact amount with no fee, rebase, or recipient callback
- transfer reverts when the sender balance is insufficient
- transfer reverts on the zero recipient address
- approve sets the allowance and transferFrom consumes it exactly
- transferFrom reverts when the allowance is insufficient
- burn destroys the caller's tokens and reduces totalSupply irreversibly
- burn reverts when the caller balance is insufficient
- burnFrom requires an allowance, so escrowed vesting balances cannot be destroyed by a third party
- TOTAL_SUPPLY diverges from totalSupply from the first burn onward
- owner holds no capability; the contract exposes no owner-gated function
- transferOwnership and renounceOwnership change owner state without conferring authority

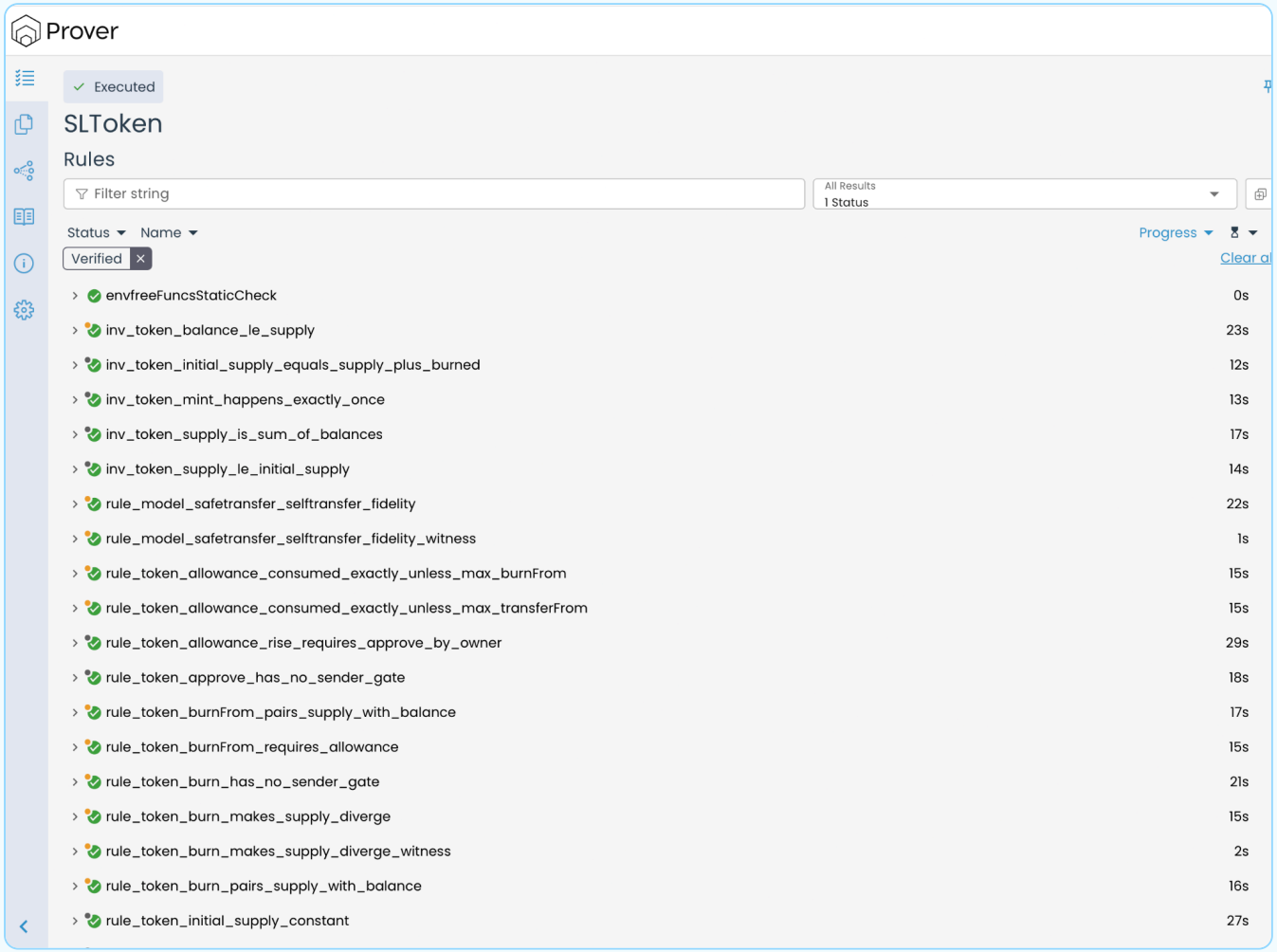
SLVesting

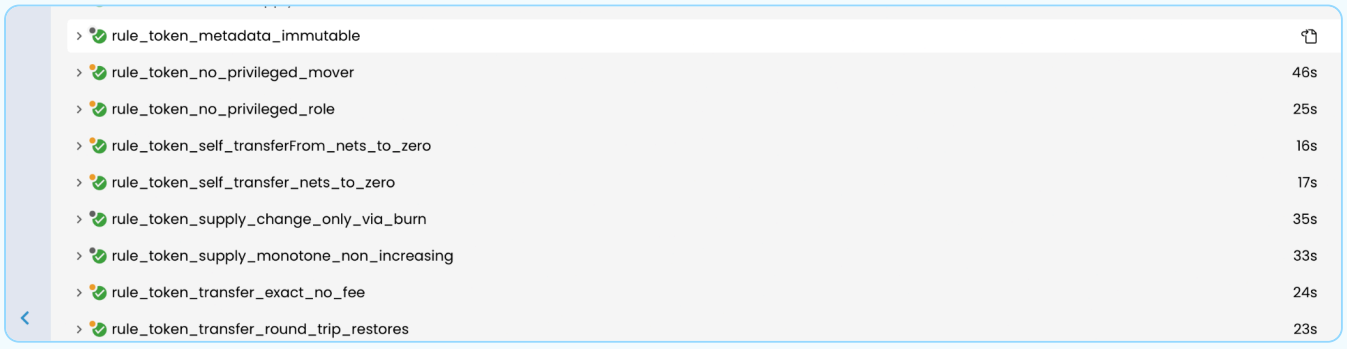
- constructor reverts if the token address is zero
- constructor reverts if the owner address is zero
- constructor stores the token address immutably
- createSchedule reverts when the caller is not the owner

- createSchedule reverts on a duplicate identifier
- createSchedule reverts on a zero beneficiary address
- createSchedule reverts on a zero total
- createSchedule reverts when tgeBps exceeds 10000
- createSchedule stores all eight schedule fields and initialises released to zero
- createSchedule appends the identifier to scheduleIds and increments scheduleCount
- createSchedule emits ScheduleCreated with the identifier, beneficiary and total
- createSchedule accepts a past start and unbounded cliff and duration without validation
- vestedAmount reverts for an unregistered identifier
- vestedAmount returns zero before start
- vestedAmount returns only the TGE share during the cliff
- vestedAmount returns zero at the exact cliff boundary when tgeBps is zero
- vestedAmount returns the proportional share inside the linear window
- vestedAmount returns the full total at the exact end of the linear window
- vestedAmount returns the full total when duration is zero and the cliff has passed
- vestedAmount never exceeds the schedule total at any timestamp
- vestedAmount is monotonically non-decreasing in timestamp
- vestedAmount reconciles to the exact total with no stranded dust
- releasable equals vestedAmount at the current block minus released
- releasable reverts for an unregistered identifier
- release reverts before the cliff has elapsed
- release reverts on a second call once the schedule is fully vested
- release transfers the accrued amount to the current beneficiary and advances released
- release emits Released with the identifier, beneficiary and amount
- release is callable by any account and always pays s.beneficiary
- release updates released before transferring, so re-entry computes a zero amount
- release reverts entirely when the pooled balance is short of the accrued amount
- claiming repeatedly yields the same total as claiming once
- updateBeneficiary reverts when the caller is not the owner
- updateBeneficiary reverts on a zero address
- updateBeneficiary reverts for an unregistered identifier

Formal Verification Results

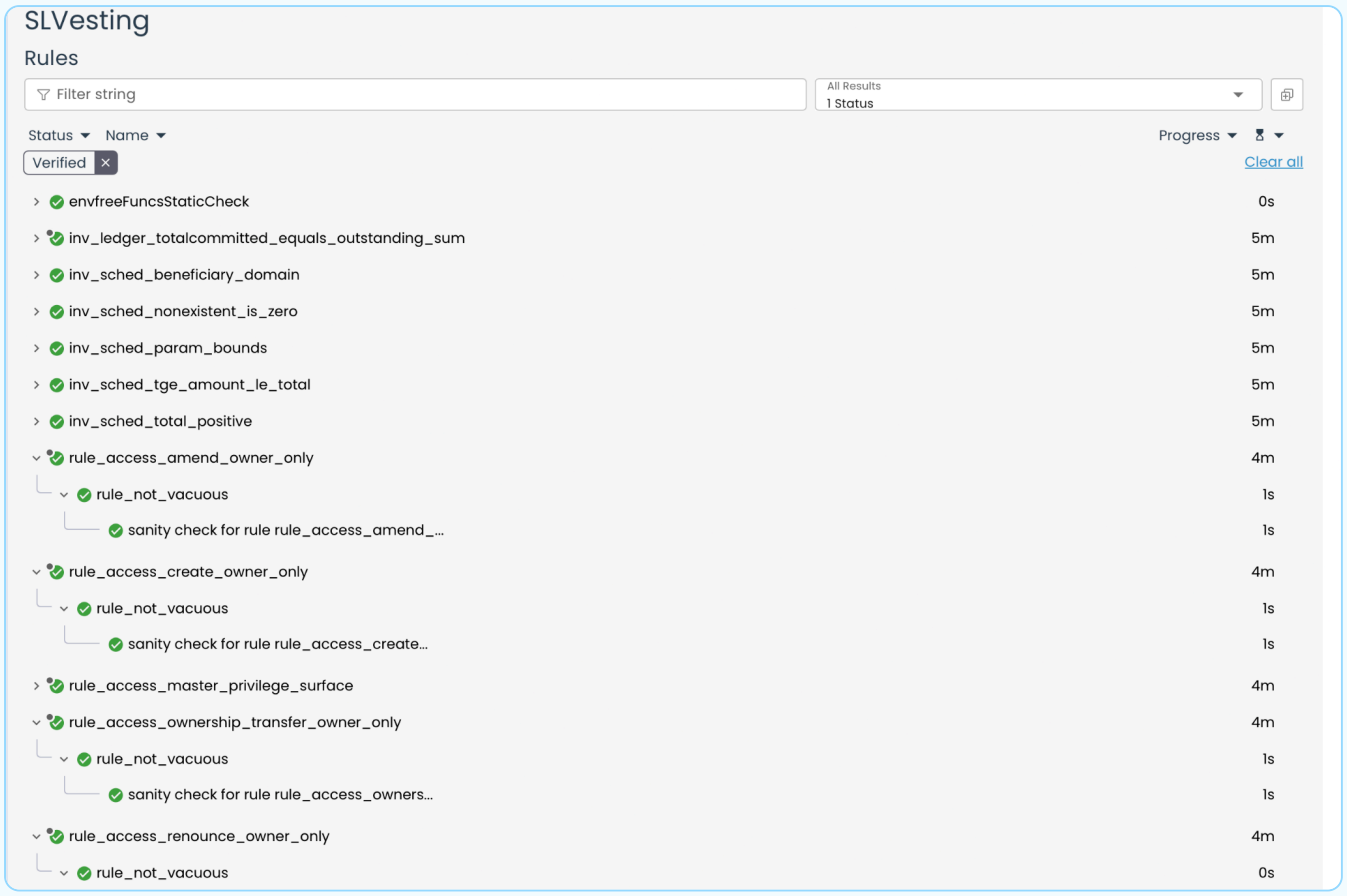
SLToken





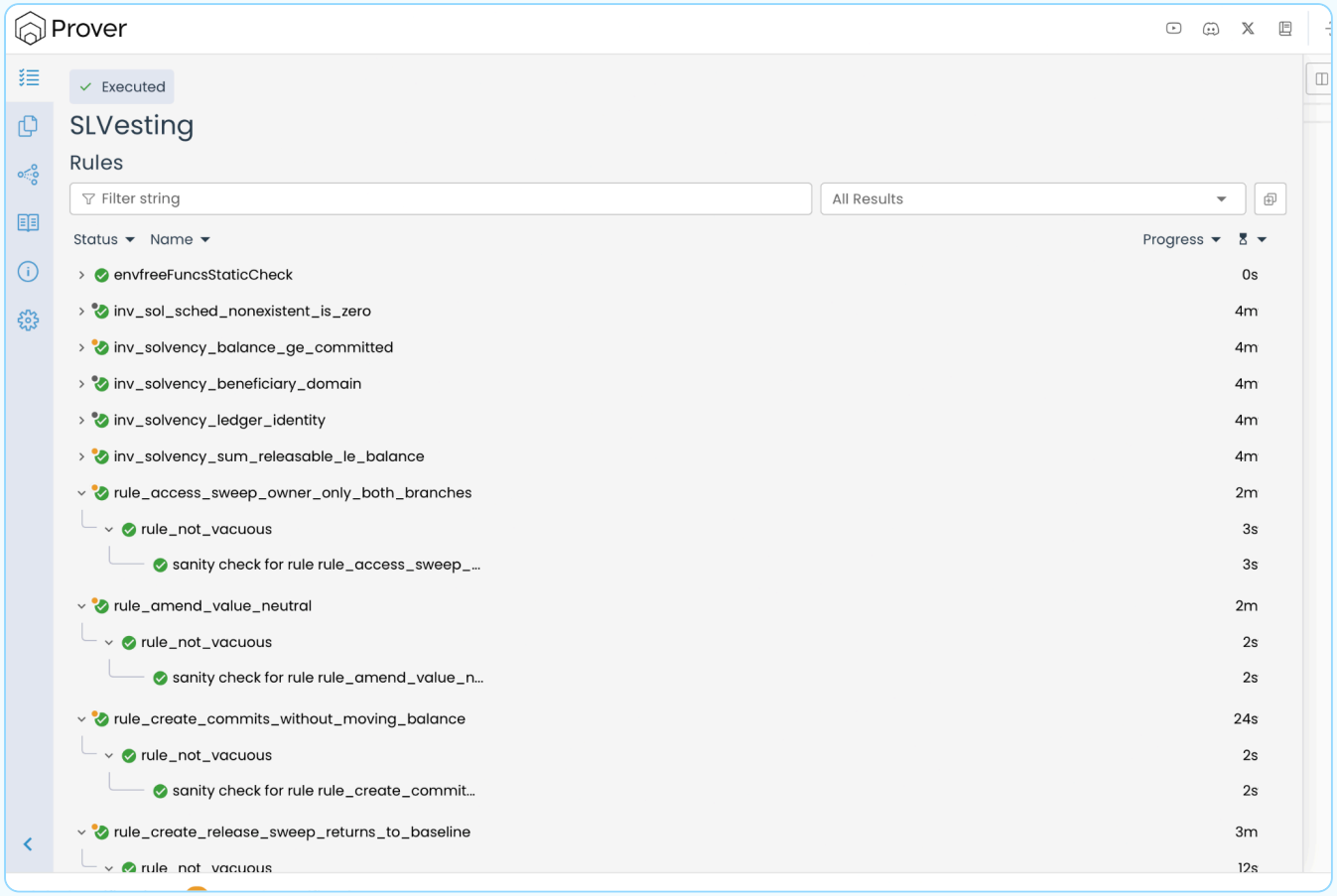
Result Link: https://prover.certora.com/output/1125024/f3294c65c77d47cd8a13ddef9fcf9fe6?anonymousKey=4096d68881baa72191ae5d4ede6ff979585b0468

SLVesting



Result Link: https://prover.certora.com/output/1125024/7332e1a5b9cc4dcd8029ad5c92432f15?anonymousKey=ad007a358a44cf867179fd798bdfa7217f77e9d5

SL_Token-final-systemSolvancy-SLVesting



Result Link: https://prover.certora.com/output/1125024/496ef40111cc4d7d90d9f497ce3201e1?anonymousKey=daddac2e9693ff393a2e75eb2261af48846c274f

Threat Model

1. External Dependencies & Trust Boundaries

This section enumerates everything the protocol does not fully control.

Dependency	Type	How Used	Trust Assumption	Risks
SLVesting owner	Admin / identity	Gates createSchedule and updateBeneficiary; set to the deploying address by the deployment script.	The owner key remains honest and available, and is rotated to a multisig before production distribution.	Owner can redirect any schedule's unreleased balance to an arbitrary address at any time with no timelock and no beneficiary consent.
SLToken owner	Admin / identity (vestigial)	Assigned by the constructor through Ownable(initialHolder); no onlyOwner function exists in the token.	Owner confers no capability and is relied upon by no execution path.	Live owner(), transferOwnership, and renounceOwnership present an admin surface that explorers and automated risk scanners can read as retained mint or pause authority.

Dependency	Type	How Used	Trust Assumption	Risks
Beneficiary addresses supplied to createSchedule	Deployment trust boundary	Stored as s.beneficiary and resolved at transfer time by release.	Each address is a correct, controllable wallet, verified before schedule creation.	A wrong address permanently misdirects an entire bucket and is correctable only through the owner-only updateBeneficiary path; validation rejects address(0) alone, so address(this) and unreachable contracts are accepted.
Off-chain allocation configuration	Deployment configuration	config/allocations.js supplies every bucket amount, type, cliff, duration, tgeBps, and destination wallet to the deployment script.	Configuration is reconciled against the published allocation table and all wallet fields are populated before a production run.	All ten buckets ship with an empty wallet field, which the script silently substitutes with the deploying address, collapsing every beneficiary and all direct allocations onto a single externally owned account.

Dependency	Type	How Used	Trust Assumption	Risks
TGE timestamp	Deployment parameter	Read from the TGE_TIMESTAMP environment variable or defaulted to the deployment block timestamp, then written to every schedule as start.	The supplied value is a unix-second timestamp consistent with the published launch date.	createSchedule applies no bounds to start; a millisecond-scaled or malformed value pushes all unlocks beyond any realistic horizon, while a past value with zero duration converts an advertised lockup into an immediate unlock, and neither is correctable.
Token address bound in the SLVesting constructor	External contract binding	Stored as an immutable IERC20 and used as the target of every safeTransfer in release.	The address is the deployed SLToken and exhibits standard, non-elastic transfer semantics.	Only a non-zero check is performed; a fee-on-transfer, rebasing, or blacklisting token would break schedule accounting, and a codeless address makes every claim revert permanently with no recovery function.

Dependency	Type	How Used	Trust Assumption	Risks
Deployment funding sequence	Operational process	Each vesting bucket is funded by a transfer and registered by a separate createSchedule in a second transaction.	Both transactions land successfully and in order for every bucket.	The two steps are not atomic and no solvency invariant links them, so a failure between them leaves either an unbacked schedule or unreachable tokens, neither of which any contract function can detect or repair.
Block timestamp	Chain runtime	releasable and release evaluate the release curve at uint64(block.timestamp).	Chain time advances monotonically and is not materially manipulable by any participant.	Sequencer or validator timestamp drift shifts accrual by seconds, which is immaterial against multi-year schedules but is the only time source the contract consults.

Dependency	Type	How Used	Trust Assumption	Risks
Event consumers	Off-chain / indexing	Indexers, dashboards, and monitoring observe ScheduleCreated, Released, and BeneficiaryUpdated.	Consumers read schedule terms from the schedules getter rather than reconstructing them from logs.	ScheduleCreated omits start, cliff, duration, and tgeBps, and BeneficiaryUpdated indexes neither address, so terms cannot be recovered from logs and fund redirection cannot be filtered by address topic.
Direct-allocation recipients	Off-chain distribution	The Network Access Credit Pool, Participant Program, and Liquidity buckets are transferred outright during deployment, with no vesting schedule and no on-chain lock.	Recipients hold and distribute the allocation according to the published policy.	No contract constrains these balances after transfer; the credit-conversion and activity-based distribution logic described in the configuration does not exist on-chain.
External liquidity locker	Out-of-scope contract	The Liquidity bucket is transferred at TGE with the stated intention that the resulting LP tokens are locked for 24 months.	A separate locker contract exists and is used promptly after liquidity provision.	No locker is present in the reviewed scope; between transfer and lock the entire Liquidity allocation is unconstrained.

Dependency	Type	How Used	Trust Assumption	Risks
Foundation governance	Off-chain process	Treasury and Ecosystem Reserve unlocks are documented as subject to a foundation governance vote.	The foundation observes its published governance process for released funds.	SLToken implements no voting extension and no governor contract is in scope, so every immediately unlocked allocation reaches ordinary wallets with no on-chain gate.

2. Entry & Exit Points Analysis (Function-Level)

This is the core of the threat model. Every externally callable function must appear here.

Contract Name	Function	Category
SLToken	`constructor(address)`	**Main invariant(s)** _mint executes exactly once; total supply is fixed at construction and can thereafter only decrease. **Access level** Deployment only. **What this function can do** Mint the entire 2,000,000,000 SL supply to initialHolder and assign the same address as owner. **What this function cannot/should not do** It should not be reachable after deployment and should not leave any further mint path open.
SLToken	`transfer(address,uint256)`	**Main invariant(s)** Balance delta equals the transferred amount exactly; total supply is unchanged. **Access level** Any holder. **What this function can do** Move the caller's SL to an arbitrary recipient, including the vesting contract. **What this function cannot/should not do** It should not apply a fee, rebase the amount, or invoke a recipient callback.

Contract Name	Function	Category
SLToken	`approve(address,uint256)`	**Main invariant(s)** Allowance is set to the supplied value, overwriting any prior value. **Access level** Any holder. **What this function can do** Set a spender allowance over the caller's balance. **What this function cannot/should not do** It should not be granted by the vesting contract to any party.
SLToken	`transferFrom(address,address,uint256)`	**Main invariant(s)** Spend is bounded by the recorded allowance and the holder's balance. **Access level** Any approved spender. **What this function can do** Move SL from an approving holder to an arbitrary recipient and consume the allowance. **What this function cannot/should not do** It should not move escrowed tokens, since the vesting contract grants no allowance.

Contract Name	Function	Category
SLToken	`burn(uint256)`	**Main invariant(s)** Supply is monotonically non-increasing; burnt tokens are unrecoverable. **Access level** Any holder. **What this function can do** Destroy the caller's own SL and reduce total supply irreversibly. **What this function cannot/should not do** It should not be assumed exclusive to the deployment-time Protocol Reserve retirement.
SLToken	`burnFrom(address,uint256)`	**Main invariant(s)** An allowance must exist; the vesting contract never grants one, so escrowed tokens cannot be destroyed externally. **Access level** Any approved spender. **What this function can do** Destroy an approving holder's SL and consume the allowance. **What this function cannot/should not do** It should not reach tokens held by the vesting contract.
		Main invariant(s) Ownership state changes but no execution path in the token consults it. **Access level**

Contract Name	Function	Category
SLToken	`balanceOf(address)` / `totalSupply()` / `allowance(address,address)`	**Main invariant(s)** Returned values reflect current ledger state including all burns. **Access level** Public view. **What this function can do** Expose live balance, supply, and allowance state. **What this function cannot/should not do** They should not be substituted by the TOTAL_SUPPLY constant when computing circulating supply.
SLToken	`TOTAL_SUPPLY()`	**Main invariant(s)** The constant is immutable and diverges from totalSupply() from the first post-deployment block onward. **Access level** Public view constant getter. **What this function can do** Expose the compile-time constant 2,000,000,000e18 representing the initial mint. **What this function cannot/should not do** It should not be treated as the live supply, and should not be used as a denominator by integrators.

Contract Name	Function	Category
SLToken	`name() / symbol() / decimals()`	**Main invariant(s)** Values are fixed at construction and are never mutated. **Access level** Public view. **What this function can do** Expose ERC-20 metadata for wallets, explorers, and integrators. **What this function cannot/should not do** They should not be relied upon for authorization or accounting decisions.
SLVesting	`constructor(address,address)`	**Main invariant(s)** token_ is non-zero and immutable thereafter; owner_ is non-zero by inherited validation. **Access level** Deployment only. **What this function can do** Bind the token address immutably and assign the vesting owner. **What this function cannot/should not do** It should not accept a token address that is not the deployed SLToken.

Contract Name	Function	Category
SLVesting	`createSchedule(bytes32,address,uint256,uint64,uint64,uint64,uint16)`	**Main invariant(s)** Identifiers are single-use; tgeBps is capped at 10000, which is the sole guarantee that total - tgeAmount cannot underflow. No check ties the sum of schedule totals to the contract's balance. **Access level** Owner only. **What this function can do** Register one schedule, fixing beneficiary, total, start, cliff, duration, and TGE share permanently. **What this function cannot/should not do** It should not register a schedule the contract cannot fund, and should not accept unbounded time parameters.
SLVesting	`release(bytes32)`	**Main invariant(s)** released is incremented before the transfer, so re-entry computes a zero amount; the payout has no amount parameter and no balance clamp. **Access level** Permissionless; any account may call, and funds always route to s.beneficiary. **What this function can do** Transfer the full accrued-but-unclaimed amount to the schedule's current beneficiary and advance released. **What this function cannot/should not do** It should not resolve the

Contract Name	Function	Category
SLVesting	`updateBeneficiary(bytes32, address)`	**Main invariant(s)** Only address(0) is rejected; s.released is not settled first, so accrued-but-unclaimed value follows the new address. **Access level** Owner only. **What this function can do** Overwrite the recipient recorded for an existing schedule. **What this function cannot/should not do** It should not transfer entitlement already accrued to the outgoing beneficiary, and should not take effect without notice or delay.
SLVesting	`transferOwnership(address)` / `renounceOwnership()`	**Main invariant(s)** Inherited single-step behaviour; renouncing permanently disables createSchedule and updateBeneficiary. **Access level** Current owner. **What this function can do** Reassign or clear the vesting owner slot. **What this function cannot/should not do** They should not move redirection authority without an acceptance step, and renouncing should not be performed unless permanent loss of recovery is intended.

Contract Name	Function	Category
SLVesting	`vestedAmount(bytes32,uint64)`	**Main invariant(s)** The return is bounded by s.total, is monotonically non-decreasing in timestamp, and assigns the remainder directly in the terminal branch so full vest reconciles exactly with no stranded dust. **Access level** Public view. **What this function can do** Evaluate the cumulative vested amount for one schedule at any caller-supplied point in time, past or future. **What this function cannot/should not do** It should not be read as the claimable amount, since it does not subtract released.
SLVesting	`releasable(bytes32)`	**Main invariant(s)** Equals vestedAmount at the current timestamp minus released; the subtraction cannot underflow because the curve is monotonic and its parameters are immutable. **Access level** Public view. **What this function can do** Report the amount claimable at the current block for one schedule. **What this function cannot/should not do** It should not be expected to return zero for an unregistered identifier.

Contract Name	Function	Category
SLVesting	`schedules(bytes32)`	**Main invariant(s)** Returned values are the ones written at creation, with beneficiary and released the only mutable fields. **Access level** Public view autogenerated getter. **What this function can do** Expose every stored field of a schedule, including the timing terms omitted from the creation event. **What this function cannot/should not do** It should not be assumed reachable through event logs alone.
SLVesting	`scheduleIds(uint256)` / `scheduleCount()`	**Main invariant(s)** The array is append-only and is read by no other function in the contract. **Access level** Public view. **What this function can do** Enumerate registered schedule identifiers and report their number. **What this function cannot/should not do** They should not be relied upon for removal or reordering, which the contract does not support.
		Main invariant(s) Set once in the constructor and unchangeable thereafter.

Contract Name	Function	Category
SLVesting	`owner()`	**Main invariant(s)** Returned account is the live authority for both owner-gated functions. **Access level** Public view. **What this function can do** Expose the account holding schedule-creation and beneficiary-redirection authority. **What this function cannot/should not do** It should not be assumed to be a multisig or timelock in the shipped deployment configuration.

3. Asset Flow Mapping (Critical Contracts)

This section identifies where money actually lives and how it moves.

3.1 Asset-Holding Contracts

Contract	Asset Type	Custodied Assets
SLVesting	SL (ERC-20)	The full vesting allocation backing every schedule, held as one undivided balance with no per-schedule segregation; the only outflow path is release.
SLToken	None in reviewed contracts	The token contract is the ledger itself and custodies no balance by design; tokens sent to its own address would be permanently unrecoverable, as no rescue function exists.
Deploying address (transient)	SL (ERC-20)	Holds the entire 2,000,000,000 SL supply from the mint until the distribution loop completes; an externally owned account rather than a contract, and the sole custodian for the duration of the deployment sequence.

3.2 Asset Entry & Exit Functions

Contract	Function	Asset In	Asset Out	Caller	Risk Notes
SLToken	constructor(address)	None.	2,000,000,000 SL minted to initialHolder.	Deployer.	The entire supply is created in one step and held by a single address until distribution completes; no further mint path exists in the contract.
SLToken	transfer(address,uint256)	None.	Caller's SL to an arbitrary recipient.	Any holder.	The only funding path into SLVesting; no vesting function runs on receipt, so the contract cannot detect, attribute, or reject an incoming transfer.

Contract	Function	Asset In	Asset Out	Caller	Risk Notes
SLToken	transferFrom(address,address,uint256)	None.	SL from an approving holder to an arbitrary recipient.	Any approved spender.	SLVesting never grants an allowance, so escrowed tokens are unreachable through this path.
SLToken	burn(uint256)	Caller's SL.	Destroyed; total supply decreases.	Any holder.	Used during deployment to retire the Protocol Reserve allocation, but permanently available to every holder, so live supply diverges from the TOTAL_SUPPLY constant from the first block.
SLToken	burnFrom(address,uint256)	Approving holder's SL.	Destroyed; total supply decreases.	Any approved spender.	Requires an allowance the vesting contract never grants; escrowed tokens cannot be destroyed by an external caller.

Contract	Function	Asset In	Asset Out	Caller	Risk Notes
SLVesting	Direct token transfer to the vesting address	SL sent by any holder.	None during receipt.	Any holder.	The balance is pooled and unattributed; any surplus beyond scheduled claims is unreachable by every caller including the owner, as no sweep, revoke, or rescue function exists.
SLVesting	createSchedule(...)	None.	None.	Owner.	Moves no assets but creates a claim on the shared pool without reading token.balanceOf(address(this)); over-commitment surfaces only as a permanent revert for whichever beneficiary claims last.

Contract	Function	Asset In	Asset Out	Caller	Risk Notes
SLVesting	release(bytes32)	None.	Full vestedAmount - released to the current s.beneficiary.	Any account.	Permissionless and all-or-nothing, with no amount parameter and no balance clamp; the destination is resolved at execution time, so a beneficiary change landing immediately before the call redirects the entire payout.
SLVesting	updateBeneficiary(bytes32,address)	None.	No direct transfer; redirects all future and accrued-but-unclaimed outflow.	Owner.	Rewrites the payee of value already credited to the outgoing beneficiary because s.released is not settled first; up to the full unreleased balance of any schedule is affected.

Contract	Function	Asset In	Asset Out	Caller	Risk Notes
SLVesting	transferOwnership(address) / renounceOwnership()	None.	No direct transfer.	Owner.	Determines who may later redirect schedules; transfer is single-step with no acceptance, and renouncing permanently removes both createSchedule and the documented lost-wallet recovery path.
Deploying address	Distribution loop (scripts/deploy.js)	2,000,000,000 SL from the mint.	The direct allocations to their wallets, the vesting allocations to SLVesting, and the Protocol Reserve burned.	Deployer.	Sixteen non-atomic transactions with no post-run assertion that the deploying balance reaches zero or that vesting holdings equal the sum of registered schedules.

Automated Tests

No major issues were found. Some false positive errors were reported by the tools. All the other issues have been categorized above according to their level of severity.

Closing Summary

In this report, we have considered the security of SL Labs. We performed our audit according to the procedure described above.

2 High, 3 Medium, 1 Low and 2 Informational severity issues were found, all of which were fixed by SL Labs Team's remediation.

Disclaimer

At QuillAudits, we have spent years helping projects strengthen their smart contract security. However, security is not a one-time event—threats evolve, and so do attack vectors. Our audit provides a security assessment based on the best industry practices at the time of review, identifying known vulnerabilities in the received smart contract source code.

This report does not serve as a security guarantee, investment advice, or an endorsement of any platform. It reflects our findings based on the provided code at the time of analysis and may no longer be relevant after any modifications. The presence of an audit does not imply that the contract is free of vulnerabilities or fully secure.

While we have conducted a thorough review, security is an ongoing process. We strongly recommend multiple independent audits, continuous monitoring, and a public bug bounty program to enhance resilience against emerging threats.

Stay proactive. Stay secure.

About QuillAudits

QuillAudits is a leading name in Web3 security, offering top-notch solutions to safeguard projects across DeFi, GameFi, NFT gaming, and all blockchain layers. With seven years of expertise, we've secured over 1400 projects globally, averting over $3 billion in losses. Our specialists rigorously audit smart contracts and ensure DApp safety on major platforms like Ethereum, BSC, Arbitrum, Algorand, Tron, Polygon, Polkadot, Fantom, NEAR, Solana, and others, guaranteeing your project's security with cutting-edge practices.

8+ Years of Expertise	**1M+** Lines of Code Audited
50+ Chains Supported	**1500+** Projects Secured

Follow Our Journey

AUDIT REPORT

August 2026

For

QuillAudits

Canada, India, Singapore, UAE, UK

www.quillaudits.com audits@quillaudits.com