QuillAudits

AUDIT REPORT

July 2025

For

Zynk Labs

Table of Content

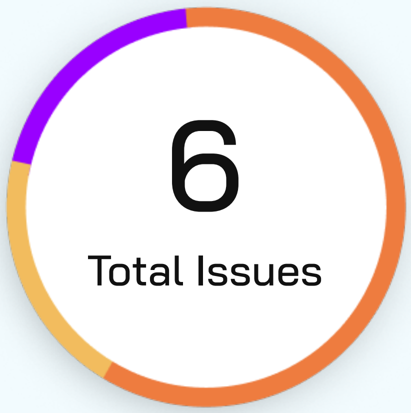
Executive Summary

Project Name	ZYNK Labs
Protocol Type	Token Transfer
Project URL	https://www.zynklabs.xyz/
Overview	The Zynk Protocol is a Solana-based smart contract that facilitates token transfers between different parties through a managed operator system. It implements a sophisticated architecture with timelock mechanisms, signature verification, and order tracking capabilities.
Audit Scope	The scope of this Audit was to analyze the Zynk Labs Smart Contracts for quality, security, and correctness.
Source Code link	https://github.com/Zynklabs/zynk-protocol-smart-contracts_solana/tree/5b8f6f17369b26aeeffa971a0597320f493ec47d
Branch	contract -refresh
Contracts in Scope	lib.rs
Commit Hash	5b8f6f17369b26aeeffa971a0597320f493ec47d
Language	Rust
Blockchain	Solana
Method	Manual Analysis, Functional Testing, Automated Testing
Review 1	10th July 2025 - 18th July 2025
Updated Code Received	18th July 2025 and 27th July 2025
Review 2	23th July 2025 - 28th July 2025
Fixed In	05b043b5dde4639ba9b57b81e829019e8913e173

Verify the Authenticity of Report on QuillAudits Leaderboard:

https://www.quillaudits.com/leaderboard

Number of Issues per Severity



🟥	Critical	0 (0.0%)
🟥	High	0 (0.0%)
🟧	Medium	3 (50%)
🟨	Low	2 (33.3%)
🟪	Informational	1 (16.7%)

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	0	3	2	1

Summary of Issues

Issue No.	Issue Title	Severity	Status
1	Replenish allows worthless tokens	Medium	Resolved
2	Guardian Initialization can be frontran	Medium	Resolved
3	Unpause Execution Does Not Verify Timelock Action Type	Medium	Resolved
4	Rent refund sent to manager instead of zynk operator wallet	Low	Resolved
5	Redundant account closure in CloseOrder	Low	Resolved
6	Non deterministic Order account creation	Informational	Resolved

Checked Vulnerabilities

We have scanned the solana program for commonly known and more specific vulnerabilities. Here are some of the commonly known vulnerabilities that we considered:

- ✓ Signer authorization
- ✓ Account data matching
- ✓ Sysvar address checking
- ✓ Owner checks
- ✓ Type cosplay
- ✓ Initialization
- ✓ Arbitrary cpi
- ✓ Duplicate mutable accounts
- ✓ Bump seed canonicalization
- ✓ PDA Sharing
- ✓ Incorrect closing accounts
- ✓ Missing rent exemption checks
- ✓ Arithmetic overflows/underflows
- ✓ Numerical precision errors
- ✓ Solana account confusions
- ✓ Casting truncation
- ✓ Insufficient SPL token account verification
- ✓ Signed invocation of unverified programs

Techniques and Methods

Throughout the audit of Solana Programs, 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
- 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.

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.

Medium Severity Issues

Replenish allows worthless tokens

Resolved

Path

lib.rs

Function Name

replenish

Likelihood

Low

Impact

High

Description

The `replenish` instruction is designed to allow the partner deposit wallet to replenish tokens to the protocol's operator wallet's token account. However, the current implementation does not properly validate the mint of the token accounts involved in the transfer. Specifically,the only constraint on the destination account (`zow_token_account`) is that it is owned by `zynk_op_wallet`:

~~~
```rust
#[account(
   mut,
constraint = zow_token_account.owner == config.zynk_op_wallet @
CustomError::InvalidTokenMint
)]
pub zow_token_account: Box<Account<‘info, TokenAccount>>,
```
~~~

On Solana, anyone can create a token account for any mint and set the owner to any address. This means a malicious partner deposit wallet can create a token account for a worthless mint (not the intended protocol mint), set its owner to `zynk_op_wallet`, and supply it as `zow_token_account`. The same applies to the source account (`pdw_token_account`). The replenish instruction will then transfer worthless tokens and increment `order_tracker.amount_in` as if valid tokens were replenished.

There is no check that the mint of `pdw_token_account` and `zow_token_account` matches the expected protocol mint or each other. This allows attackers to replenish with arbitrary, valueless tokens, undermining the integrity of the protocol's accounting and token balances.

Impact

- **Protocol Accounting Corruption**: Attackers can artificially inflate `order_tracker.amount_in` with worthless tokens, making the protocol believe valid tokens have been replenished.
- **Loss of Funds**: The protocol may release assets or perform actions based on fake replenishments, leading to financial loss or theft.
- **Bypass of Business Logic**: Any logic relying on the replenished amount (e.g., order closure mechanism) can be bypassed or abused.

Recommendation

- **Enforce Mint Validation**: Add constraints to ensure both `pdw_token_account` and `zow_token_account` have the same mint and that this mint matches the intended protocol mint. For example:

~~~
   ```rust
   #[account(
      mut,
      constraint = pdw_token_account.mint == zow_token_account.mint @
CustomError::InvalidTokenMint,
      constraint = zow_token_account.mint == EXPECTED_PROTOCOL_MINT @
CustomError::InvalidTokenMint,
      constraint = zow_token_account.owner == config.zynk_op_wallet @
CustomError::InvalidTokenMint
   )]
   pub zow_token_account: Box<Account<'info, TokenAccount>>,
   ```
~~~

Where **`EXPECTED_PROTOCOL_MINT`** is a constant or a field in your config/account that stores the correct mint address.

- Validate in Instruction Logic: In addition to account constraints, validate the mint addresses in the instruction logic before performing the transfer and updating `order_tracker.amount_in`.

Specific Fixed In Commit

b6276f24cca9e02b72f95e014dbe2b2008862265

Zynk Labs team's Response

The likelihood of the above attack vector is very low, as mentioned earlier, PDWs are wallets totally controlled by us. We create, track and validate the PDWs offchain. In other words, the `replenish()` function can never be invoked by any external user, unless they gain access to the private key of the PDW. But still, for the sake of robustness, we have updated the `create_order()` function to accept PDW's ATA of the concerned token instead of the PDW itself, and have applied checks for the ATA.

2nd update: The upgraded implementation restricts the `create_order()` to accept only a certain token account of the partner wallet. Since,the aforesaid function is a controlled feature having ZOWas the signer, we declare which token account can be used during `replenish()`. The constraints in the `create_order()` boils down to

PDW token account mint == beneficiary token account mint == ZOW token account mint.

Once the PDW token account gets embedded inside `orderTracker`, the PDW token account being passed in the `replenish()` must strictly match with the one in the `orderTracker`. Now,even if an attacker tries to manipulate the token accounts during `replenish()`, it is impossible for the transaction to go through, unless s/he has the access to the ZOW's private key beforehand for creating the order.

```
    let pdw_token_account = ctx.accounts.pdw_token_account.key();
    // Verify that the pdw_token_account is authorized by comparing it with the stored pdw_token_account.
      require!(
          pdw_token_account == order_tracker.pdw_token_account,
          CustomError::UnauthorizedSigner
      );
```

Guardian Initialization can be frontran

Resolved

Path

lib.rs

Path

Initialize and set_guardian

Likelihood

Low

Impact

High

Description

The protocol’s `Config` struct includes a `guardian` field, which is critical for security-sensitive operations such as pausing the contract and bypassing timelocks. However, the `initialize` function does not set the `guardian` field during deployment, leaving it as `Pubkey::default()`. The `set_guardian` function is then used to assign the guardian, but it currently allows any account to call it as long as the guardian is unset. This creates a frontrunning vulnerability: any user can race to set themselves as guardian before the intended admin, potentially taking control of privileged actions.

Impact

- UnauthorizedAccess: Any user can become the protocol guardian if they call `set_guardian` before the admin, gaining control over critical functions.
- Loss of Control: The intended admin or team may lose the ability to manage or recover the protocol if a malicious actor sets themselves as guardian.
- Protocol Integrity Risk:The guardian role can bypass timelocks and pause/unpause the contract, so unauthorized access could disrupt protocol operations or enable malicious actions.

Recommendation

- SetGuardian During Initialization: Add a `guardian` parameter to the `initialize` function and set it at deployment, removing the need for a separate assignment.
- RestrictAccess to set_guardian: If post-deployment assignment is required, restrict `set_guardian` so only the admin can call it:

```rust
    pub fn set_guardian(ctx: Context<Misc>) -> Result<()> {
        let config = &mut ctx.accounts.config;
        require!(config.guardian == Pubkey::default(),
CustomError::AlreadyExecuted);
        require!(ctx.accounts.authority.key == config.admin,
CustomError::UnauthorizedAdmin);
        config.guardian = ctx.accounts.authority.key();
        Ok(())
    }
```

Specific Fixed In Commit

7d52219a0c3d4f43d4ff6251dccf519ce08032d7

Zynk Labs team's Response

Removed the redundant `setGuardian()` function and included `guardian` in the `initialize()` function.

Unpause Execution Does Not Verify Timelock Action Type

Resolved

Description

The execute_unpause function does not verify that the timelock request being executed is actually an "Unpause"action. It only checks the execution status and timing/ack conditions, but does not enforce that req.action corresponds to the TimelockAction::Unpause variant.

Impact

Any valid timelock request (with correct timing or ack) could be used to unpause the protocol, regardless of its intended action. This could allow unauthorized or unintended unpausing, potentially bypassing governance or security controls.

Recommendation

Add a check to ensure that req.action is equal to the value for TimelockAction::Unpause before proceeding with the unpause logic. For example:

```
// …existing code…
let action: TimelockAction = req.action.try_into()?;
require!(action == TimelockAction::Unpause, CustomError::InvalidAction);
// …existing code…
```

Specific Fixed In Commit

05b043b5dde4639ba9b57b81e829019e8913e173

Zynk Labs team's Response

Applied the necessary check

Low Severity Issues

Rent refund sent to manager instead of zynk operator wallet

Resolved

Path

lib.rs

Function

close_order

Description

In the `CloseOrder`account struct, the `order_tracker` account uses the Anchor `close` constraint to send the remaining rent (lamports) to the `manager` when the account is closed:

~~~
```rust
#[account(
   mut,
   close = manager
)]
pub order_tracker: Account<‘info, OrderTracker>,
```
~~~

However, the `order_tracker` account is originally created and funded by the `zynk_op_wallet` (the protocol operator) in the `pull_and_create_order()` and `create_order` instructions. This means the rent paid for account creation comes from the zync operator wallet, but when the account is closed, the refund is sent to the manager instead.

Recommendation

Pass the `zynk_op_wallet` in instruction as mutable and change the `close` constraint in the `CloseOrder` struct to send the rent refund to the `zynk_op_wallet` instead of the manager:

~~~
```rust
#[account(
   mut,
   close = zynk_op_wallet
)]
pub order_tracker: Account<‘info, OrderTracker>,
```
~~~

Zynk Labs team’s Response

This is an intended functionality.

Redundant account closure in CloseOrder

Resolved

Path

lib.rs

Function Name

close_order

Description

In the `CloseOrder`struct, the `order_tracker` account uses the Anchor `close` constraint to automatically close the account and transfer its lamports to the `manager` upon successful instruction execution:

```rust
#[account(
   mut,
   close = manager
)]
pub order_tracker: Account<‘info, OrderTracker>,
```

However, within the `close_order` instruction logic, the account is also manually closed using the `close_account` helper function:

```rust
close_account(&ctx.accounts.order_tracker, &ctx.accounts.manager)?;
```

This results in redundant attempts to close the same account, which is unnecessary and could lead to confusion or unexpected behavior.

Recommendation

Remove the manual call to `close_account` in the instruction logic and rely solely on the Anchor `close` constraint for account closure. This will simplify the code and avoid redundant operations.

Specific Fixed In Commit

7d52219a0c3d4f43d4ff6251dccf519ce08032d7

Zynk Labs team’s Response

Removed the redundant `close_account()` function invocation.

Informational Issues

Non deterministic Order account creation

Resolved

Path

lib.rs

Function Name

pull_and_create_order(), create_order()

Description

Currently, the `order_tracker` account for each order is created using the default Anchor `init` constraint, which does not use deterministic seeds. This means the address of each order account is not predictable and cannot be derived off-chain using known parameters. By incorporating the order nonce as seeds when creating the order account, we can make the order account address deterministic and easily discoverable.

Recommendation

Update the order account creation to use deterministic seeds, such as:

~~~~
```rust
#[account(
   init,
   payer = zynk_op_wallet,
   space = OrderTracker::LEN,
   seeds = [b"order", &nonce.to_le_bytes()],
   bump
)]
pub order_tracker: Account<'info, OrderTracker>,
```
~~~~

This will make each order account address predictable and tied to its unique nonce.

Zynk Labs team's Response

We are not using `order_tracker` as a PDA but a generated Keypair, used only once when signing the `create_order()` transaction.

Functional Tests

initialize

- ✓ Initializes config with correct admin, zynk_op_wallet, guardian, manager, paused = false, current_nonce = 0.
- ✓ Fails if payer is not the admin.

pull_and_create_order

- ✓ Succeeds when all accounts and signature are valid, protocol is not paused.
- ✓ Fails if protocol is paused.
- ✓ Fails if signature is invalid.
- ✓ Fails if token mints do not match.
- ✓ Fails if partner_deposit_wallet is not a signer.
- ✓ Emits OrderCreation event with correct fields.
- ✓ Increments nonce and stores order details.

create_order

- ✓ Succeeds when all accounts and signature are valid, protocol is not paused
- ✓ Fails if protocol is paused.
- ✓ Fails if signature is invalid.
- ✓ Fails if token mints do not match.
- ✓ Emits OrderCreation event with correct fields.
- ✓ Increments nonce and stores order details.
- ✓ If amount is zero, does not transfer tokens.

replenish

- ✓ Succeeds when protocol is not paused, order_id matches, validity is in future, amount > 0, pdw_token_account matches.
- ✓ Fails if protocol is paused.
- ✓ Fails if order_id does not match.

- ✓ Fails if validity is not in the future.
- ✓ Fails if amount is zero.
- ✓ Fails if pdw_token_account does not match order_tracker.
- ✓ Emits OrderReplenish event with correct fields.
- ✓ Increases amount_in in order_tracker.

close_order

- ✓ Succeeds if order_id matches and amount_in >= amount_out.
- ✓ Fails if order_id does not match.
- ✓ Fails if amount_in < amount_out.
- ✓ Emits OrderClosure event with correct fields.
- ✓ Transfers lamports to manager and closes the order_tracker account.

request_timelock

- ✓ Succeeds for valid action and value.
- ✓ Fails for invalid action.
- ✓ Sets correct eta and value.
- ✓ Emits Action event with status Initiated.

revoke_timelock

- ✓ Succeeds if not executed and acked.
- ✓ Fails if already executed.
- ✓ Fails if not acked.
- ✓ Emits Action event with status Revoked.

ack_timelock

- ✓ Succeeds if not executed.
- ✓ Fails if already executed.
- ✓ Sets ack to true.
- ✓ Emits Action event with status Acked.

execute_wallet_update

- ✓ Succeeds if not executed, consensus is false, eta has passed or action is acked, and value is valid.
- ✓ Fails if already executed.
- ✓ Fails if consensus is true.
- ✓ Fails if eta has not passed and action is not acked.
- ✓ Fails if value is invalid.
- ✓ Updates the appropriate field in the config.
- ✓ Emits an Action event with status Executed.

execute_unpause

- ✓ Succeeds if not executed, eta or acked.
- ✓ Fails if already executed.
- ✓ Fails if not ready (eta/acked).
- ✓ Sets config.paused = false.
- ✓ Emits Action event with status Executed.

pause

- ✓ Succeeds if called by admin, manager, or guardian.
- ✓ Fails if called by unauthorized signer.
- ✓ Sets config.paused = true.

request_consensus

- ✓ Succeeds for valid action and value.
- ✓ Fails for invalid action.
- ✓ Sets consensus = true.
- ✓ Emits Action event with status Initiated.

execute_consensus

✓ Succeeds if not executed, consensus is true, and value is valid.

✓ Fails if already executed.

✓ Fails if consensus is true.

✓ Fails if value is invalid.

✓ Updates the appropriate field in the config.

✓ Emits an Action event with status Executed.

✓ Closes timelock account and transfers lamports to guardian.

domain_separator

✓ Succeeds and logs correct DOMAIN_SEPARATOR value.

Threat Model

Contract	Function	Threats
ZynkProtocol	initialize	Zero-address parameters for zynk_op_wallet / manager, Centralized ownership risk, Missing guardian initialization
	pull_and_create_order	Manager signature replay attacks, Token freeze by mint authority, Nonce overflow DoS, Partner wallet compromise drains funds, Beneficiary receives frozen/ worthless tokens
	create_order	Arbitrary partner_deposit_wallet manipulation, Manager signature forgery, Token supply manipulation by mint authority, Zero-value orders waste gas
	replenish	Worthless token deposits to fulfill orders, Expired validity timestamps bypass, Partner wallet compromise, Token mint mismatch exploitation
	close_order	Manager-only closure creates centralization risk, Premature closure with insufficient replenishment, Double closure via constraint and manual call

Contract	Function	Threats
	request_timelock	Limited to 5 concurrent timelocks (action 0-4), Admin-only control creates single point of failure, Timelock griefing attacks
	execute_wallet_update	Guardian bypass without ETA validation, Missing guardian update action (hardcoded exclusion), Invalid timelock action processing
	execute_guardian_update	Requires both admin and guardian signatures, No guardian bypass option, Address validation can be bypassed
	set_guardian	Anyone can set guardian initially (critical vulnerability), One-time setting with no recovery mechanism, Race condition during initialization
	execute_unpause	Guardian can bypass timelock delays, No validation of unpause necessity, Admin dependency for unpause operations
	pause	Missing AND logic creates authorization bypass, Multiple actors can pause but only admin can unpause, No pause duration limits
	verify_signature_syscall	Ed25519 instruction data manipulation, Signature replay across different contexts, Message format injection attacks

Contract	Function	Threats
	validate_address	Only checks for default pubkey, No validation against protocol-controlled addresses, System program addresses allowed
	close_account	Manual closure combined with Anchor's close constraint, Lamport drain attacks, Account data not fully zeroed
Config	Storage	Mutable state without proper access controls, Guardian address can be overwritten, Nonce overflow potential
OrderTracker	Storage	No PDA derivation for unique orders, Amount tracking integer overflow, Partner wallet spoofing
TimelockRequest`	Storage	Fixed delay periods cannot be adjusted, Action enum limited to 5 values, ETA manipulation possible

Closing Summary

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

Issues of Medium, Low and informational severity were found. At the end the Zynk Labs team resolved all issues

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.

QuillAudits

7+ Years of Expertise	**1M+** Lines of Code Audited
50+ Chains Supported	**1400+** Projects Secured

Follow Our Journey

AUDIT REPORT

July 2025

For

Zynk Labs

QuillAudits

Canada, India, Singapore, UAE, UK

www.quillaudits.com audits@quillaudits.com