QuillAudits

AUDIT REPORT

November 2025

For

SkyTrade

Table of Content

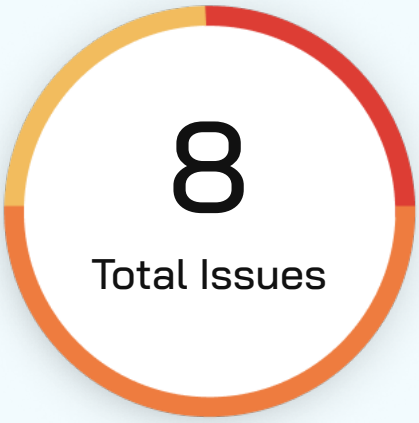
Executive Summary

Project Name	SkyTrade
Project URL	https://skytrade.finance/
Overview	Skytrade Smart contracts provide a system for launching regulatory-compliant securities tokens on a decentralized blockchain. This particular repository is the implementation of a system that allows for the creation of ST-20-compatible tokens. This system has a modular design that promotes a variety of pluggable components for various types of issuances, legal requirements, and offering processes.
Source Code link	https://github.com/SkyTrade-Finance/contracts/tree/feat-audit
Branch	Main
Commit Hash	6df32bca6100b8da6b1193e2fad3132bbb37dec2
Language	Solidity
Blockchain	EVM
Method	Manual Analysis, Functional Testing, Automated Testing
Review 1	21st October 2025 - 27th October 2025
Updated Code Received	4th November & 10th November 2025
Review 2	5th November 2025 - 12th November 2025

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	4 (50%)
Low	2 (25%)
Informational	0 (0%)

Severity

Issues	Critical	High	Medium	Low	Informational
Open	0	0	0	0	0
Acknowledged	0	1	0	0	0
Partially Resolved	0	0	0	0	0
Resolved	0	1	4	2	0

Summary of Issues

Issue No.	Issue Title	Severity	Status
1	Signature replay attack enables unauthorized merkle root modifications through reused signatures	High	Acknowledged
2	Zero minimum token protection enables slippage attacks during tier transitions in STO purchases	High	Resolved
3	Root expiry remains stale when merkle root updates occur after initialization	Medium	Resolved
4	Missing root expiry validation enables verification against expired merkle trees and wastes gas through unnecessary proof storage	Medium	Resolved
5	Empty receive function enables accidental ETH loss without token issuance in STO contracts	Medium	Resolved
6	Incompatible ERC20 token transfers cause transaction failures with non-standard stable coins	Medium	Resolved
7	Gas limit issues with transfer	Low	Resolved
8	ERC1594 interface violation breaks standard compliance with incorrect status code type	Low	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
- Upgradeable safety
- Using throw
- Using inline assembly
- Style guide violation
- Unsafe type inference
- 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
- 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

Remix IDE, Foundry, Solhint, Mythril, Slither, Solidity Static Analysis.

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

Signature replay attack enables unauthorized merkle root modifications through reused signatures

Acknowledged

Path

contracts/external/TradingRestrictionManager/TradingRestrictionManager.sol:63-85

Function Name

`updateMerkleRootWithSignature`

Description

The updateMerkleRootWithSignature function accepts operator signatures to update the merkle root and expiry but lacks replay attack protection mechanisms such as nonces or unique identifiers. The function verifies that the signature comes from an authorized operator by recovering the signer address from the signature and checking against the isOperator mapping. However, the signature verification process only validates the authenticity of the signature against the provided root and expiry parameters without implementing any safeguards to prevent the reuse of previously valid signatures.

The signature creation process at lines 75-76 generates a message hash using only the root and expiry parameters, making it susceptible to replay attacks when the same combination of root and expiry values are used multiple times. Since the function does not maintain any record of previously used signatures or implement a nonce-based system, an attacker who obtains a valid signature can replay it indefinitely as long as the expiry timestamp remains in the future and is greater than or equal to the current root expiry.

Consequently, malicious actors who intercept or obtain valid operator signatures can repeatedly execute the updateMerkleRootWithSignature function with the same parameters, potentially overriding legitimate merkle root updates with previously used values. This vulnerability allows attackers to revert the contract state to outdated KYC data sets, effectively undermining the integrity of the investor verification system and potentially enabling unauthorized trading activities by investors whose KYC status should have been updated or revoked.

Recommendation

We recommend implementing a nonce-based replay protection mechanism by adding a mapping to track used nonces for each operator and including the nonce value in the signature message hash. The function should verify that the provided nonce has not been used previously and increment the nonce counter after successful verification to prevent signature reuse.

SkyTrade team's Comment

The system was intentionally designed to allow users to reuse the same valid signature for multiple token purchases, provided it is a legitimate signature from the operator and has not expired. This behavior is expected and aligns with the intended design of the signature validation mechanism. The signatures we are going to create will be with the expiry time set to a very short time, like 10 minutes; hence, the ability to execute this form of attack, as explained in the analysis, is negligible.

reason: The intentional design of signature reuse is advantageous to be used on multiple purchases and minimising the need for regenerating the signedRoot and signature from an operator every time for a new purchase through buyWithUSD - **https://github.com/SkyTrade-Finance/contracts/blob/b438b47532c8072bea30d0f6a1fe225b47f53179/contract-hardhat/contracts/modules/STO/USDTiered/USDTieredSTO.sol#L417**, also effectively eliminating another transaction to update existing KYC data before purchase

Zero minimum token protection enables slippage attacks during tier transitions in STO purchases

Resolved

Path

contracts/modules/STO/USDTiered/USDTieredSTO.sol:424-426

Function Name

`buyWithUSD`

Description

The buyWithUSD function calls buyWithUSDRateLimited with a hardcoded minimum token value of zero, removing slippage protection for investors purchasing tokens through the tiered pricing system. The function processes investments sequentially through pricing tiers that can advance between transaction submission and execution as other investors deplete current tier token supplies.

However, setting minimum tokens to zero allows investors to receive significantly fewer tokens than expected when tier progression occurs during transaction processing. When multiple transactions compete for limited tier allocations or large investments exhaust current tier availability, subsequent transactions execute at higher tier prices without adequate protection against unfavorable pricing changes. The tiered mechanism advances dynamically during execution, making final token allocation unpredictable at transaction submission time.

Consequently, investors face unlimited slippage exposure during high-demand periods when tier boundaries shift rapidly due to competing transactions. Sophisticated actors can exploit this vulnerability through front-running attacks, deliberately exhausting favorable pricing tiers to force legitimate investors into less advantageous positions. The zero minimum token requirement provides no recourse for investors who receive substantially fewer tokens than anticipated, creating unfair market conditions that particularly disadvantage retail participants who cannot monitor real-time tier progression.

Recommendation

We recommend replacing the hardcoded zero value with a calculated minimum token amount based on current tier pricing and investor expectations, or implementing a slippage tolerance parameter that allows investors to specify their acceptable minimum token allocation relative to their investment amount.

Medium Severity Issues

Root expiry remains stale when merkle root updates occur after initialization

Resolved

Path

contracts/external/TradingRestrictionManager/TradingRestrictionManager.sol:48-61

Function Name

`modifyKYCData`

Description

The modifyKYCData function updates the merkle root for KYC validation but only sets the expiry timestamp during the initial root setup when _root equals bytes32(0). The function checks if this is the first time setting the root and only then updates _rootExpiry to one week from the current timestamp. However, when the root is updated in subsequent calls, the function skips the expiry update logic entirely, leaving _rootExpiry unchanged from its previous value. This creates a disconnect between the freshly updated root and its associated expiry timestamp.

The function design assumes that only the initial root setting requires an expiry update, but this assumption fails to account for the practical need to refresh expiry dates when new KYC data becomes available.

Recommendation

We recommend modifying the modifyKYCData function to always update the _rootExpiry timestamp when setting a new root, regardless of whether this is the initial setup or a subsequent update. The expiry update logic should be moved outside the conditional check for _root == bytes32(0) to ensure that every root update includes a corresponding expiry refresh of one week from the current timestamp or modifyKYCData takes another parameter called _rootExpiry to set new expiry for given root.

Missing root expiry validation enables verification against expired merkle trees and wastes gas through unnecessary proof storage

Resolved

Path

contracts/external/TradingRestrictionManager/TradingRestrictionManager.sol:103-125

Function Name

`verifyInvestor`

Description

The verifyInvestor function validates investor KYC data using merkle proofs but fails to check whether the current merkle root has expired by comparing _rootExpiry against block.timestamp. The function only validates that the individual investor proof expiry is greater than the current timestamp at line 111 but does not verify the validity of the merkle root itself. Additionally, the function stores the complete proof array in the _kycData mapping at line 122, consuming significant gas for storage operations that serve no future purpose since proofs are never retrieved or validated again after initial verification.

However, this design creates a fundamental security flaw where investors can successfully verify their KYC status using proofs against expired merkle roots, potentially allowing access to outdated or revoked KYC information. The merkle root expiry mechanism exists to ensure that KYC data remains current and compliant with regulatory requirements, but the lack of root expiry validation undermines this security measure entirely. Furthermore, the function design requires each investor to execute a separate transaction for KYC verification, which contradicts the efficiency benefits that merkle trees are designed to provide in blockchain systems where gas costs and transaction throughput are critical considerations.

Recommendation

We recommend adding a root expiry validation check by comparing _rootExpiry with block.timestamp before proceeding with proof verification, and removing the proof storage from the _kycData mapping since proofs are not reused after verification. Additionally, integrate the KYC verification logic directly into the security token transfer mechanism to eliminate the need for separate investor verification transactions and leverage the merkle tree structure for efficient on-demand validation during actual trading operations.

Empty receive function enables accidental ETH loss without token issuance in STO contracts

Resolved

Path

contracts/modules/STO/USDTiered/USDTieredSTO.sol:350-352

Description

The contract implements an receive function that accepts ETH payments without executing any business logic or token issuance operations, allowing any address to send ETH directly to the contract address without triggering the proper investment flow.

However, this design creates a critical flaw where investors can lose funds by sending ETH directly to the contract instead of using designated purchase functions like buyWithETH or buyWithETHRateLimited. The receive function accepts payments without providing corresponding security tokens, creating an un-accounted one-way transfer which does not give any tokens in return to the investor.

Consequently, investors who accidentally send ETH to the contract address will lose their funds permanently since the empty receive function provides no mechanism for token issuance or fund recovery. This scenario particularly affects retail investors who may use simple ETH transfers instead of calling appropriate purchase functions.

Recommendation

We recommend removing the receive function entirely from both USDTieredSTO and SecurityToken contracts to prevent accidental ETH transfers that do not result in proper token issuance.

Incompatible ERC20 token transfers cause transaction failures with non-standard stable coins

Resolved

Path

contract-hardhat/contracts/modules/STO/USDTiered/USDTieredSTO.sol:567

Function Name

`_buyWithTokens`

Description

The _buyWithTokens function uses a require statement with _token.transferFrom(msg.sender, wallet, spentValue) to transfer stable coins from investors to the STO wallet. This implementation expects the transferFrom function to return a boolean value indicating success or failure, which is the standard EIP-20 behavior. However, many popular stable coins like USDT on Ethereum mainnet do not return boolean values from their transfer and transferFrom functions, instead implementing void functions that revert on failure. Additionally, some tokens may return false instead of reverting, which would cause the require statement to fail even for successful transfers.

Consequently, investors using non-standard stable coins like USDT would be unable to participate in the STO, as their transactions would revert when the contract expects a boolean return value that these tokens do not provide. This significantly limits the contract's compatibility with widely-used stable coins and reduces the potential investor base. Furthermore, the contract becomes vulnerable to inconsistent behavior across different ERC20 implementations, creating an unreliable user experience where some tokens work while others fail unexpectedly.

Recommendation

We recommend replacing the current require(_token.transferFrom(msg.sender, wallet, spentValue), "Transfer failed") implementation with OpenZeppelin SafeERC20 library's safeTransferFrom function, which handles both standard and non-standard ERC20 implementations by checking return values when present and treating successful execution without reversion as success when no return value exists.

Low Severity Issues

Gas limit issues with transfer

Resolved

Path

contract-hardhat/contracts/modules/STO/USDTiered/USDTieredSTO.sol:468-470

Function Name

`buyWithETHRateLimited`

Description

The buyWithETHRateLimited function uses the transfer method to send ETH to both the issuer wallet and refund excess ETH to the investor. The transfer method forwards a fixed gas stipend of 2300 gas, which may be insufficient for recipient contracts with complex fallback functions or those interacting with proxy patterns.

Consequently, the current implementation could fail to transfer funds to legitimate smart contract wallets or multisig contracts that require more than 2300 gas for execution. Additionally, if the issuer wallet or investor address is a contract that reverts on ETH receipt, the entire STO transaction would fail, preventing token purchases. The fixed gas limit also creates compatibility issues with future Ethereum protocol changes that might alter gas costs for basic operations, making the contract less future-proof.

Recommendation

We recommend replacing the transfer calls with low-level call methods and adding the nonReentrant modifier from OpenZeppelin ReentrancyGuard to the buyWithETHRateLimited function to prevent reentrancy attacks while maintaining compatibility with all recipient contract types and ensuring reliable ETH transfers regardless of recipient contract complexity.

ERC1594 interface violation breaks standard compliance with incorrect status code type

Resolved

Path

contract-hardhat/contracts/interfaces/ISecurit yToken.sol:34

Function Name

`canTransfer`

Description

The canTransfer function declares its return type as (bytes32 statusCode, bytes32 reasonCode) which violates the ERC1594 standard interface specification. The ERC1594 standard explicitly defines the canTransfer function to return (bytes1, bytes32) where the first parameter represents a single-byte status code following EIP-1066 Ethereum Status Codes. However, the current interface incorrectly specifies the status code as bytes32 instead of bytes1, creating a fundamental type mismatch that breaks standard compliance and interoperability with ERC1594-compliant systems.

Consequently, this interface violation prevents proper integration with external systems and protocols that expect strict ERC1594 compliance, potentially causing transaction failures or unexpected behavior when interacting with compliant smart contracts or decentralized applications. The inconsistency also creates confusion for developers implementing the interface, as the actual SecurityToken implementation correctly uses bytes1 while the interface incorrectly specifies bytes32, leading to potential compilation errors and integration issues.

Recommendation

We recommend changing the canTransfer function signature in ISecurity Token interface from returns (bytes32 statusCode, bytes32 reasonCode) to returns (bytes1 statusCode, bytes32 reasonCode) to align with the ERC1594 standard specification and maintain consistency with the actual implementation.

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 SkyTrade. We performed our audit according to the procedure described above.

Issues of high , medium , low issues were found. Audit was focused on the new changes introduced by the SkyTrade team within the existing polymath-core codebase. We strongly recommend conducting multiple audits and participating in bug bounty programs to ensure the highest level of security

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.

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

Follow Our Journey

AUDIT REPORT

November 2025

For

SkyTrade

QuillAudits

Canada, India, Singapore, UAE, UK

www.quillaudits.com audits@quillaudits.com