QuillAudits

AUDIT REPORT

June 2025

For

alkimi

Table of Content

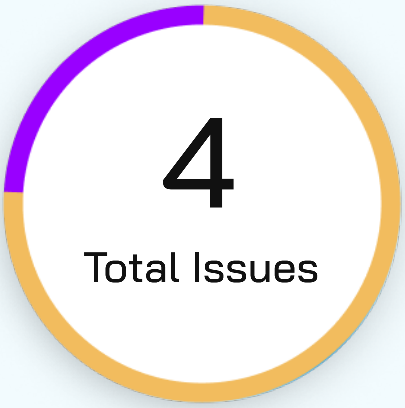
Executive Summary

Project Name	Alkimi
Protocol Type	Token Contract
Project URL	https://www.alkimi.org/
Overview	A single smart contract for minting $ALKIMI tokens on the Sui blockchain. The Alkimi token is implemented on the Sui blockchain using the Move language. The contract is designed as a fungible token with: * Strict administrative controls * Secure minting and burning operations
Audit Scope	The scope of this Audit was to analyze the Alkimi Smart Contracts for quality, security, and correctness.
Source Code link	https://github.com/Alkimi-Exchange/Quills/blob/main/token_gen_move_contract/sources/token.move
Contracts in Scope	Token.move
Branch	main
Commit Hash	69c1390e5e6269a35dc599fd1868bca1b5c58d17
Language	Move
Blockchain	Sui
Method	Manual Analysis, Functional Testing, Automated Testing
Review 1	28th May 2025 - 6th June 2025
Updated Code Received	24th June 2025
Review 2	24th June 2025 - 26th June 2025
Fixed In	4277fb11b15c95220027860fa83ead814b371526

Verify the Authenticity of Report on QuillAudits Leaderboard:

https://www.quillaudits.com/leaderboard

Number of Issues per Severity



Critical		0 (0%)
High		0 (0%)
Medium		0 (0%)
Low		3 (75%)
Informational		1 (25%)

Severity

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

Summary of Issues

Issue No.	Issue Title	Severity	Status
1	Potential Integer Overflow in TokenMinted and TokenBurned	Low	Resolved
2	Treasury balance Field Not Updated	Low	Resolved
3	max_admins hard limit will be effectively permanent	Low	Acknowledged
4	No-Op TreasuryCap Removal and Reinsertion	Informational	Resolved

Checked Vulnerabilities

- ✔ Transaction-ordering dependence
- ✔ Integer overflow/underflow by bit operations
- ✔ Timestamp dependence
- ✔ Number of rounding errors
- ✔ Denial of service / logical oversights
- ✔ Business logic contradicting the specification
- ✔ Timestamp dependence
- ✔ Number of rounding errors
- ✔ Access control
- ✔ Gas usage
- ✔ Code clones, functionality duplication
- ✔ Unchecked CALL Return Values
- ✔ Witness Type
- ✔ Centralization of power

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.

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

Likelihood \ Impact	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.

Low Severity Issues

Potential Integer Overflow in TokenMinted and TokenBurned

Resolved

Path

token.move

Description

The TokenMinted and TokenBurned structs track cumulative totals of all minting and burning operations using u64 fields. These counters are incremented on every mint/burn operation without overflow protection. If the cumulative sum of all historical mints or burns exceeds the maximum u64 value (18,446,744,073,709,551,615), the addition operation will panic, causing a permanent denial of service for minting or burning operations.

POC

Permanent DoS: Once overflow occurs, no further mints or burns can be executed

Likelihood

Low

- Maximum u64 value is extremely large (~18.4 quintillion)
- With 9 decimals, this represents ~18.4 billion actual tokens
- Would require minting/burning the entire supply (~2.5B tokens) thousands of times

Recommendation

Document the Limitation: If keeping current design, clearly document the theoretical limit and monitor approaching thresholds.

Treasury balance Field Not Updated

Resolved

Path
token.move

Function
initialize

Description
The Treasury struct includes a balance: Balance<ALKIMI> field, presumably to track the internal token holdings of the treasury. However, in the mint function, this balance is never updated. The function instead mints new ALKIMI tokens and directly transfers them to treasury.treasury_wallet using transfer::public_transfer(...), without modifying the Treasury.balance field.

Furthermore, the Treasury object is passed by immutable reference (&Treasury), making any update to the balance field impossible even if intended.

```
/// @notice Treasury - holds the initial supply of tokens
/// @dev Stores the initial 2.5 billion tokens that can be distributed later
public struct Treasury has key, store {
    id: UID,
    balance: Balance<ALKIMI>, // <—
    treasury_wallet: address
}
```

Impact
If balance is meant to track treasury holdings this will be a broken implementation.

Likelihood
Low - While not causing immediate issues, it creates confusion and maintenance burden.

Recommendation
Consider following one of the two suggestions:

Suggestion 1: Remove the unused balance field
If you're always using the treasury_wallet address and tracking supply via TokenSupply, then this field might be unnecessary.

Suggestion 2: Actually use and update the balance field
Pass &mut Treasury instead of &Treasury
Update the balance with:

```
balance::add(&mut treasury.balance, amount);
```

This assumes you're not just transferring to an address, but want to account for it inside the Treasury object itself — e.g., if Treasury is stored on-chain and meant to track internal state.

max_admins hard limit will be effectively permanent

Acknowledged

Path

token.move

Description

In the AdminRegistry struct initialization:

```
// Create admin registry with deployer as the owner but no initial admins
// The owner is not automatically an admin, maintaining separate role
responsibilities
       let admin_registry = AdminRegistry {
           id: object::new(ctx),
           admins: admin_vector,
           max_admins: 2, // @note no way to increase max_admins
           owner: deployer,
      };
```

The max_admins field is hardcoded to 2, and is later enforced by an assertion:

```
     // Check if max admins reached
       assert!(vector::length(&registry.admins) < registry.max_admins,
E_MAX_ADMINS_REACHED);
```

However, there is no mechanism provided to modify max_admins post-deployment.

Impact

This hard limit is effectively permanent, with no function allowing the owner to increase it, not even via governance or future situations.

Likelihood

Low

Recommendation

Add an entry function, accessible only by the owner, to update max_admins. e.g:

```
entry fun update_max_admins(
    registry: &mut AdminRegistry,
    new_limit: u64,
    ctx: &TxContext
   ) {
     assert!(tx_context::sender(ctx) == registry.owner, E_NOT_OWNER);
     assert!(new_limit >= vector::length(&registry.admins), E_INVALID_LIMIT);
     registry.max_admins = new_limit;
   }
```

This provides future flexibility while preserving safety checks.

Informational Issues

No-Op TreasuryCap Removal and Reinsertion

Resolved

Path

token.move

Function

transfer_full_ownership()

Description

The function removes the TreasuryCap<ALKIMI> from the ProtectedTreasury object:

```
// Extract the TreasuryCap from the protected treasury
 let treasury_cap = remove_cap(protected_treasury);
```

Then, after a check:

```
// Verify old treasury is now empty (additional security check)
    assert!(!dof::exists_(&protected_treasury.id, TreasuryCapKey {}),
        E_TREASURY_NOT_EMPTY);
```

It re-adds the exact same cap back to the same object:

```
dof::add(&mut protected_treasury.id, TreasuryCapKey {}, treasury_cap);
```

This operation effectively performs no change in system state — the cap remains in the same place under the same control.

Recommendation

No immediate action required. However, if the intent was to transfer actual control over the TreasuryCap, consider reworking this logic to ensure access is passed to the new owner. Otherwise, this can be safely considered a no-op verification pattern.

Functional Tests

Some of the tests performed are mentioned below:

- ✔ Unauthorised Use of Shared Objects
- ✔ Mint Integrity (Only Admin or Owner can mint up to max supply)
- ✔ Burn Integrity (Only Admin or Owner can burn tokens)
- ✔ Increase Supply (Only Admin or Owner can change, Can't be less than or equal to max supply)
- ✔ Decrease Supply (Only Admin or Owner can change, Can't be greater than or equal to max supply and shouldn't be less than or equal to current supply)
- ✔ Add admin (Only Owner can call this function, admin should not be owner)
- ✔ Remove Admin (Only owner can call this function to remove existing admin)
- ✔ Set treasury wallet (Only owner can update wallet address)
- ✔ Transfer Ownership (Only current owner can transfer this ownership to a non admin address)

Threat Model

Contract	Function	Threats
test_token::test_v7	mint()	• Supply Manipulation: Malicious admin mints to max supply, blocking future mints • Treasury Hijacking: Compromised admin repeatedly mints to drain protocol resources • Front-running: Attacker front-runs mint with burn to manipulate metrics
test_token::test_v7	burn()	• Griefing Attack: Admin burns user tokens if they gain coin access • Economic Attack: Coordinated burns to artificially increase scarcity • Zero-amount Spam: Repeated zero burns to bloat event logs
test_token::test_v7	add_admin() remove_admin()	• Privilege Escalation: Compromised owner adds malicious admins (2 new attack vectors) • Admin Lock-out: Owner removes all admins then loses keys (single point of failure)

Contract	Function	Threats
test_token::test_v7	increase_max_supply() decrease_max_supply()	• Social Engineering: Phishing attack to add attacker as admin • Supply Cap Attack: Set max supply to u64::MAX, removing limit • Liquidity Trap: Decrease below current supply, preventing mint • Market Manipulation: Coordinated changes to influence price
test_token::test_v7	transfer_full_ownership()	• Ownership Hijacking: Compromised owner transfers to attacker • Dead Treasury: Transfer to address without private keys • Orphaned Treasury DoS: Repeated transfers create zombie objects
test_token::test_v7	set_treasury_wallet()	• Fund Redirection: Redirect future mints to attacker address • Zero Address Lock: Set to 0x0 to block mints (fails with E_ZERO_ADDRESS) • MEV Exploitation: Monitor changes and front-run large mints

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

Issues of low and informational severity were found. Alkimi team acknowledged one and resolved others

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 six years of expertise, we've secured over 1000 projects globally, averting over $30 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
$30B+ Secured in Digital Assets	**1400+** Projects Secured

Follow Our Journey

AUDIT REPORT

June 2025

For

alkimi

QuillAudits

Canada, India, Singapore, UAE, UK

www.quillaudits.com audits@quillaudits.com