Veridise

Auditing Report

Hardening Blockchain Security with Formal Methods

FOR

FORTE

Sealance Compliance Technology for Aleo

Veridise Inc.
September 08, 2025

► **Prepared For:**

Forte Labs
`https://www.forte.io`

► **Prepared By:**

Alberto Gonzalez
Mark Anthony

► **Contact Us:**

`contact@veridise.com`

► **Version History:**

Sep. 29, 2025	V2
Sep. 25, 2025	V1

Contents

1 Executive Summary

From Sep. 08, 2025 to Sep. 19, 2025, Forte Labs engaged Veridise to conduct a security assessment of their Sealance Compliance Technology. The security assessment covered the Leo on-chain program source code as well as auxiliary TypeScript programs, including Merkle tree generation and deployment scripts, for Sealance's compliant token transfer protocol built on the Aleo blockchain. Veridise conducted the assessment over 4 person-weeks, with 2 security analysts reviewing the project over 2 weeks on commit `9bc54de`*. The review strategy involved a thorough, manual review of the program source code performed by Veridise security analysts.

Project Summary. The Sealance Compliance Technology for Aleo protocol tries to address the regulatory compliance gap in Aleo's privacy-focused blockchain by implementing token policies that selectively generate compliance records when specific criteria is triggered. The protocol maintains token transfers privacy by default while creating detailed records for designated investigator addresses based on configurable conditions such as daily transaction volume thresholds, or timelock requirements.

The source code is mainly composed of the following on-chain Leo programs:

- `Merkle_tree.leo`: This program provides functionality for verifying Merkle tree non-inclusion proofs, enabling privacy-preserving verification of address status in freeze lists without revealing the address publicly.
- `sealance_freezelist_registry.leo`: This program manages a registry of frozen addresses using both mappings for public non-inclusion verification and Merkle roots for private non-inclusion verification.
- `sealed_report_policy.leo`: This program implements a compliance token that integrates with the token registry and generates transfer reports in the form of Aleo records sent to investigator addresses, while enforcing that both sender and recipient are not on the freeze list (either publicly or privately verified).
- `sealed_threshold_report_policy.leo`: This program implements a compliance token that integrates with the token registry and generates transfer reports as Aleo records sent to investigator addresses only when daily transfer volumes exceed configured thresholds, while ensuring both sender and recipient are not on the freeze list.
- `sealed_timelock_policy.leo`: This program implements a compliance token that integrates with the token registry to enforce time-locked token transfers, preventing tokens from being transferred until a specified lock-up period expires, while ensuring both sender and recipient are not on the freeze list before processing transfers.
- `sealed_report_token.leo`:This program provides a self-contained compliance token with freeze list controls and transfers compliance reporting that operates independently without relying on the token registry program.

Code Assessment. The Sealance Compliance Technology developers provided the source code of the Sealance Compliance Technology contracts for the code review. The source code is based on the token implementation in the Aleo Token Registry program, and it appears to be

* The repository, if it is publicly available, can be found at https://github.com/sealance-io/compliant-transfer-aleo

mostly original code written by the Sealance Compliance Technology developers. It contains extensive documentation in the form of READMEs and inline comments on functions.

To facilitate the Veridise security assessment, the Sealance Compliance Technology developers met with the audit team to provide a detailed walkthrough of the protocol, including its structure, trust assumptions, and intended operation. This was supplemented by written external documentation, which further clarified design choices and implementation details.

The source code contained a robust test suite, which the Veridise security analysts noted adequately tested the important functional workflows including positive and negative paths.

Summary of Issues Detected. The security assessment uncovered 8 issues, 1 of which is assessed to be of a high severity by the Veridise analysts. Specifically, V-SLC-VUL-001 highlights how users could bypass the freeze list check by using Merkle proofs of intermediate nodes. The Veridise analysts also identified 1 medium-severity issue. In particular, V-SLC-VUL-002 describes how a missing ownership record check could have allowed users to bypass the freeze list check in the Timelock policy program. Additionally, the Veridise team identified 1 low-severity issue, 3 warnings, and 2 informational findings, such as V-SLC-VUL-004, which highlights that the current implementation does not align with the specification to support trees with a depth of 15 and V-SLC-APP-VUL-002, which describes how a potentially incorrect assumed block rate allows users to bypass the expected transfer volume threshold in the threshold policy token program.

Recommendations. After conducting the assessment of the protocol, the security analysts had a few suggestions to improve the Sealance Compliance Technology project. Overall, the codebase demonstrates high quality with a comprehensive test suite, and the identified issues were self-contained to specific lines of code requiring only small changes to fix with the issues originating from edge cases rather than fundamental design flaws.

Hardcoded Token Parameters Limit Reusability. The current policy implementations make extensive use of hardcoded constants that significantly limit their reusability across different token deployments. All policy programs (`sealed_timelock_policy.leo`, `sealed_threshold_report_policy.leo`, and `sealed_report_policy.leo`) embed token-specific metadata directly in the source code, including token ID, name, symbol, decimals and total supply. Additionally, the threshold policy hardcodes compliance parameters such as the transfer volume threshold and the epoch to reset users' transfer volume for transaction monitoring.

Any new token deployment requires modifying the source code. The team should document these limitations clearly for future developers and forks of the protocol, providing a checklist of all constants that must be updated when deploying new token variants. Consider including deployment templates or configuration guides that explicitly list all hardcoded values and their purposes.

Disclaimer. We hope that this report is informative but provide no warranty of any kind, explicit or implied. The contents of this report should not be construed as a complete guarantee that the system is secure in all dimensions. In no event shall Veridise or any of its employees be liable for any claim, damages or other liability, whether in an action of contract, tort or otherwise, arising from, out of or in connection with the results reported here.

2 Project Dashboard

Table 2.1: Application Summary.

Name	Version	Type	Platform
Sealance Compliance Technology	`9bc54de`	Leo	Aleo

Table 2.2: Engagement Summary.

Dates	Method	Consultants Engaged	Level of Effort
Sep. 08–Sep. 19, 2025	Manual	2	4 person-weeks

Table 2.3: Vulnerability Summary.

Name	Number	Acknowledged	Fixed
Critical-Severity Issues	0	0	0
High-Severity Issues	1	1	1
Medium-Severity Issues	1	1	1
Low-Severity Issues	1	1	1
Warning-Severity Issues	3	3	3
Informational-Severity Issues	2	2	2
TOTAL	8	8	8

Table 2.4: Category Breakdown.

Name	Number
Data Validation	2
Usability Issue	2
Maintainability	2
Logic Error	1
Authorization	1

3 Security Assessment Goals and Scope

3.1 Security Assessment Goals

The engagement was scoped to provide a security assessment of Sealance Compliance Technology's source code. During the assessment, the security analysts aimed to answer questions such as:

- Do the compliance tokens implement all required functionality correctly?
- Are policy compliance configurations consistent with actual Aleo network parameters?
- Is freezelist non-inclusion verified correctly?
- Are the Merkle proofs validated for correct depth? Do the proofs ensure the reported neighbors are actual leaf nodes?
- Are compliance records emitted as per the defined requirements? And can anyone other than the investigator access them?
- Are compliance requirements correctly enforced? Can any actor bypass them?
- Are privileged actions gated behind access control? And is the access control implemented correctly?
- Do the deployment and upgrade scripts handle private keys in a secure manner?
- Are updates to the public freezelist registry and the freezelist merkle tree consistent? And is the update process efficient and reliable?
- Is the protocol susceptible to common Leo-specific vulnerabilities?
- Is Aleo program upgradability implemented correctly?
- Do deployment scripts initialise the programs correctly? And is it done atomically?
- Are network-specific environment variables and configurations managed correctly?
- Is the Merkle proof construction consistent between the Aleo program and the off-chain scripts?
- Is Aleo address-to-field (and field-to-address) conversion implemented correctly?

3.2 Security Assessment Methodology & Scope

Security Assessment Methodology. To address the questions above, the security assessment involved a thorough manual review of the Leo program source code and the deployment scripts conducted by human experts.

Scope. The scope of this security assessment is limited to the (`programs/`), (`lib/`) and (`scripts/`) folders of the source code provided by Sealance Compliance Technology developers. These folders contain the Aleo programs that implement Sealance's compliant tokens, as well as the deployment scripts used to manage and operate them.

- `programs/merkle_tree.leo`
- `programs/sealance_freezelist_registry.leo`
- `programs/sealed_report_policy.leo`
- `programs/sealed_report_token.leo`
- `programs/sealed_threshold_report_policy.leo`
- `programs/sealed_timelock_policy.leo`

- `lib/Block.ts`
- `lib/Constant.ts`
- `lib/Conversion.ts`
- `lib/Deploy.ts`
- `lib/Freezelist.ts`
- `lib/Fund.ts`
- `lib/Initialize.ts`
- `lib/MerkleTree.ts`
- `lib/Role.ts`
- `lib/Token.ts`
- `lib/Upgrade.ts`
- `scripts/deploy-devnet.ts`
- `scripts/deploy-testnet.ts`
- `scripts/update-freeze-list.ts`
- `scripts/upgrade.ts`

Methodology. Veridise security analysts inspected the provided tests, went through the provided documentation, and then began a manual review of the code.

During the security assessment, the Veridise analysts regularly communicated with the developers to ask questions, discuss potential concerns, and provide updates on the review's progress.

3.3 Classification of Vulnerabilities

When Veridise security analysts discover a possible security vulnerability, they must estimate its severity by weighing its potential impact against the likelihood that a problem will arise.

The severity of a vulnerability is evaluated according to the Table 3.1.

Table 3.1: Severity Breakdown.

	Somewhat Bad	Bad	Very Bad	Protocol Breaking
Not Likely	Info	Warning	Low	Medium
Likely	Warning	Low	Medium	High
Very Likely	Low	Medium	High	Critical

The likelihood of a vulnerability is evaluated according to the Table 3.2.

Table 3.2: Likelihood Breakdown

Not Likely	A small set of users must make a specific mistake
Likely	Requires a complex series of steps by almost any user(s) - OR - Requires a small set of users to perform an action
Very Likely	Can be easily performed by almost anyone

The impact of a vulnerability is evaluated according to the Table 3.3:

Table 3.3: Impact Breakdown

Somewhat Bad	Inconveniences a small number of users and can be fixed by the user
Bad	Affects a large number of people and can be fixed by the user - OR - Affects a very small number of people and requires aid to fix
Very Bad	Affects a large number of people and requires aid to fix - OR - Disrupts the intended behavior of the protocol for a small group of users through no fault of their own
Protocol Breaking	Disrupts the intended behavior of the protocol for a large group of users through no fault of their own

4 Trust Model

4.1 Operational Assumptions

In addition to assuming that any out-of-scope components behave correctly, Veridise analysts assumed the following properties held when modeling security for Sealance Compliance Technology.

- **Policy Token Registration.** Except for the timelock policy, compliance policies are not self-registered by their respective programs. Instead, they are pre-registered in the *token registry* by an independent, private-key-controlled account. Analysts assume that this account correctly assigns the external authorization party to each policy program, and also configures the associated minting and burning privileges.
- **Administrative Burn Privileges.** With the exception of the timelock policy, all compliance-related tokens can be publicly burned by an administrative account. For the `report_token`, this authority resides with the program's admin. For the `report_policy` and `threshold_policy`, the burn privilege is held by the account that originally registered the token in the `token registry`. Analysts assume that these administrative accounts exercise their burn rights strictly in accordance with protocol requirements, and do not arbitrarily destroy tokens in a manner that could undermine user interests.

4.2 Privileged Roles

Roles. This section describes in detail the specific roles present in the system, and the actions each role is trusted to perform. The roles are grouped based on two characteristics: privilege-level and time-sensitivity. *Highly-privileged* roles may have a critical impact on the protocol if compromised, while *limited-authority* roles have a negative, but manageable impact if compromised. Time-sensitive *emergency* roles may be required to perform actions quickly based on real-time monitoring, while *non-emergency* roles perform actions like deployments and configurations which can be planned several hours or days in advance.

During the review, Veridise analysts assumed that the role operators perform their responsibilities as intended. Protocol exploits relying on the below roles acting outside of their privileged scope are considered outside of scope.

- Highly-privileged, emergency roles. Each policy program defines some highly privileged roles, but not all programs define the same set of roles. These roles and the privileged actions associated with each role are mentioned below.
  - *Admin*. Each policy program defines an admin role that holds the highest privileges. It can upgrade the program, update and assign other privileged roles, update the freeze list and mint and burn tokens.
  - *Minter*. An address with a minter role can mint policy tokens. The timelock policy, and sealed report token explicitly set a minter role. Whereas for the sealed report, and sealed threshold report policy the minter address is set through the token registry.
  - *Burner*. An address with a burner role, can burn tokens.
  - *Freezelist Manager*. A freeze list manager can update the freeze list.

  - *Supply Manager*. The supply manager can both mint and burn tokens.
- Highly-privileged, non-emergency roles:
  - *Deployer.* The deployer address is responsible for deploying the policy programs. It is intended to be different from the admin address.
  - *Investigator*. The policy programs emit compliance records as per the compliance requirements, for the investigator address.

Operational Recommendations. Highly-privileged, non-emergency operations should be operated by a multi-sig contract or decentralized governance system. These operations should be guarded by a timelock to ensure there is enough time for incident response. Highly-privileged, emergency operations should be tested in example scenarios to ensure the role operators are available and ready to respond when necessary.

Full validation of operational security practices is beyond the scope of this review. Users of the protocol should ensure they are confident that the operators of privileged keys are following best practices such as:

- Never storing a protocol key in plaintext, on a regularly used phone, laptop, or device, or relying on a custom solution for key management.
- Using separate keys for each separate function.
- Storing multi-sig keys in a diverse set of key management software/hardware services and geographic locations.
- Enabling 2FA for key management accounts. SMS should *not* be used for 2FA, nor should any account which uses SMS for 2FA. Authentication apps or hardware are preferred.
- Validating that no party has control over multiple multi-sig keys.
- Performing regularly scheduled key rotations for high-frequency operations.
- Securely storing physical, non-digital backups for critical keys.
- Actively monitoring for unexpected invocation of critical operations and/or deployed attack contracts.
- Regularly drilling responses to situations requiring emergency response such as pausing/unpausing.

5 Vulnerability Report

This section presents the vulnerabilities found during the security assessment. For each issue found, the type of the issue, its severity, location in the code base, and its current status (i.e., acknowledged, fixed, etc.) is specified. Table 5.1 summarizes the issues discovered:

Table 5.1: Summary of Discovered Vulnerabilities.

ID	Description	Severity	Status
V-SLC-VUL-001	Non-inclusion proofs can be forged with . . .	High	Fixed
V-SLC-VUL-002	Time lock policy does not validate record . . .	Medium	Fixed
V-SLC-VUL-003	Uninitialized admin roles expose . . .	Low	Fixed
V-SLC-VUL-004	Merkle tree depth mismatch limits . . .	Warning	Fixed
V-SLC-VUL-005	Missing leaf index bounds validation . . .	Warning	Fixed
V-SLC-VUL-006	Inefficient Freeze List Management Leads . . .	Warning	Fixed
V-SLC-VUL-007	Maintainability concerns	Info	Fixed
V-SLC-VUL-008	Unoptimized code	Info	Fixed

5.1 Detailed Description of Issues

5.1.1 V-SLC-VUL-001: Non-inclusion proofs can be forged with intermediate nodes

Severity	High	**Commit**	9bc54de
Type	Logic Error	**Status**	Fixed
Location(s)	`programs/merkle_tree.leo:51-79`		
Confirmed Fix At	https://github.com/sealance-io/compliant-transfer-aleo/pull/88, d9f3727		

Description The `verify_non_inclusion()` transition is designed to prove that a given address is absent from the freezelist registry merkle tree. It does this by checking the merkle proofs of two sibling leaves and confirming that the target address lies strictly between them. Since the tree is sorted in ascending order and does not contain duplicates, establishing that the address falls between two existing sibling leaves serves as evidence that the address itself is not part of the tree. See snippet below for the implementation.

```
transition verify_non_inclusion(addr: address, merkle_proofs: [MerkleProof;2]) -> field {
    let (root1, depth1): (field, u32)= calculate_root_depth_siblings(merkle_proofs[0u32]);
    let (root2, depth2): (field, u32) = calculate_root_depth_siblings(merkle_proofs[1u32]);

    // Ensure the roots from the merkle proofs are the same
    assert_eq(root1, root2);

    let addr_field: field = addr as field;
    if (merkle_proofs[0u32].leaf_index == merkle_proofs[1u32].leaf_index) {
        // Ensure that if the address is the most left leaf, it is less than the first sibling
        if (merkle_proofs[0u32].leaf_index == 0u32) {
            assert(addr_field < merkle_proofs[0u32].siblings[0u32]);
        } else {
            // Ensure that if the address is the most right leaf
            let last_index_leaf: u32 = 2u32 ** (depth1 - 1u32) - 1u32;
            assert_eq(merkle_proofs[0u32].leaf_index, last_index_leaf);
            // Ensure that the address is bigger than the first sibling
            assert(addr_field > merkle_proofs[0u32].siblings[0u32]);
        }
    } else {
        // Ensure the address is in between the provided leaves
        assert(addr_field > merkle_proofs[0u32].siblings[0u32]);
        assert(addr_field < merkle_proofs[1u32].siblings[0u32]);
        // Ensure the leaves are adjacent
        assert_eq(merkle_proofs[0u32].leaf_index + 1u32, merkle_proofs[1u32].leaf_index);
    }

    return root1;
}
```

However, the function does not validate the depth of the merkle proofs or ensure that the neighbors are in fact leaf nodes. As a result, an attacker can supply merkle proofs for two intermediate (non-leaf) nodes that still satisfy the ordering checks and build to the same root, thereby proving non-inclusion for an arbitrary address.

For instance, if an attacker can identify 2 intermediate non-leaf siblings such that
`(intermediate_left_sibling < address < intermediate_right_sibling)` or
`(address < intermediate_leftmost_sibling)` or
`(address > intermediate_rightmost_sibling)`, then they can use these siblings to prove non-inclusion. This attack is similar to the second preimage attack on standard merkle trees to prove inclusion, whereas here it can be leveraged to prove non-inclusion.

Impact Any system that relies on `verify_non_inclusion` for implementing a freeze-list, allowlist or access control can be bypassed. A malicious user can prove that an excluded address is not in the set, even when it is, which enables the user to perform unauthorized transfers.

Recommendation The function should verify that both sibling proofs correspond to leaf nodes of the freezelist registry merkle tree.

This can be done by introducing domain separators for the leaves and intermediate nodes, or verifying the depth of the merkle proofs against the depth of the merkle tree.

Developer Response The developers now hash the leaf nodes with `1field`, and the intermediate nodes with `0field`.

5.1.2 V-SLC-VUL-002: Time lock policy does not validate record owner during public transfers

Severity	Medium	**Commit**	9bc54de
Type	Data Validation	**Status**	Fixed
Location(s)	programs/sealed_timelock_policy.leo:169-212		
Confirmed Fix At	https://github.com/sealance-io/compliant-transfer-aleo/pull/89, https://github.com/sealance-io/compliant-transfer-aleo/pull/89/, 2562ab2, eee1d34		

Description The `transfer_public()` transition in the timelock policy enables public transfers of tokens to a recipient once the timelock period has expired. As part of the process, both the caller and the recipient are verified against the freezelist registry through non-inclusion proofs, and the provided `CompliantToken` record is consumed if the timelock condition is satisfied. See snippet below for reference.

```
async transition transfer_public(
    public recipient: address,
    public amount: u128,
    sealed_token: CompliantToken,
    lock_until: u32,
) -> (CompliantToken, CompliantToken, Future) {
    let verify_sender: Future = sealance_freezelist_registry.aleo/
    verify_non_inclusion_pub(self.caller);
    let verify_recipient: Future = sealance_freezelist_registry.aleo/
    verify_non_inclusion_pub(recipient);

        // Veridise - elided
```

The problem is that the transition does not enforce that the caller is also the owner of the `CompliantToken` record being transferred. This means a frozen user could route a transfer through an intermediate program that passes the non-inclusion checks, bypassing the restriction and moving funds they should not be able to use.

The same issue is also present in `transfer_public_to_priv()` within the timelock policy program.

Impact An attacker can circumvent the freeze-list restrictions and move funds to a recipient even when their address is frozen.

Recommendation In `transfer_public()` and `transfer_public_to_priv()`, verify that the `self.caller` is the `owner` of the sealed token record being consumed.

Developer Response The developers now verify that the caller is indeed the owner of the record.

5.1.3 V-SLC-VUL-003: Uninitialized admin roles expose programs to takeover risk

Severity	Low	**Commit**	9bc54de
Type	Authorization	**Status**	Fixed
Location(s)	`programs/` ▶ `sealance_freezelist_registry.leo:79` ▶ `sealed_report_policy.leo:152` ▶ `sealed_report_token.leo:160` ▶ `sealed_threshold_report_policy.leo:219`		
Confirmed Fix At	`https://github.com/sealance-io/compliant-transfer-aleo/pull/98`, b3883ec		

Description Across multiple programs, role checks rely on the pattern `roles.get_or_use(ADMIN_INDEX, caller)` instead of a stricter `roles.get()`. This means that if the `ADMIN_INDEX` role has not yet been initialized, the first caller of certain transitions can implicitly become the admin, even if that was never intended.

For example, in `update_role()`, the `get_or_use` call will default the admin role to the caller if it has not yet been set, and then assert that the caller is the admin. Similarly, in `finalize_mint_public()`, the same pattern allows any caller to mint tokens as long as the admin role has not been explicitly initialized. This assumption that initialization will always be performed immediately after deployment by a trusted party introduces a window of opportunity for an attacker to take over the program.

```
async transition update_role(public new_address: address, role: u8) -> Future {
    return f_update_role(new_address, self.caller, role);
}
async function f_update_role(new_address: address, caller: address, role: u8) {
    let admin_address: address = roles.get_or_use(ADMIN_INDEX, caller);
    assert_eq(admin_address, caller);
    roles.set(role, new_address);
}
```

The same pattern also appears in the upgrade logic of the policy programs via the custom `constructor`. Here, the admin is derived through `roles.get_or_use(ADMIN_INDEX, self.program_owner)`. If the admin role is not explicitly set during initialization, the first account to invoke the upgrade flow can silently assume control of the program logic.

```
@custom
async constructor() {
    let admin_address: address = roles.get_or_use(ADMIN_INDEX, self.program_owner);
    assert_eq(admin_address, self.program_owner);
}
```

Although there is an assumption that the admin role will be promptly initialized after deployment, relying on operational assumptions rather than explicit enforcement is historically dangerous. There are many instances where initialization flaws in the EVM ecosystem have previously been exploited (e.g. the CPIMP attack) showing that implicit initialization is dangerous in practice.

Impact If the admin role is not initialized immediately at deployment, an attacker can front-run or race to claim admin rights by invoking any of the affected transitions. This enables unauthorized role changes, minting of tokens, or even upgrades of program logic. Depending on the program, this could result in complete loss of control over governance and token supply.

Recommendation It is not advisable to rely implicitly on the assumption that the admin role will always be initialized immediately after deployment by a trusted party. Instead, the code should enforce this explicitly:

- Move the initialization of the `ADMIN` role into each program's `initialize` function, and restrict this function so that it can only be called by an expected Aleo account. Once all initializations have been completed, this account can be discarded since it no longer serves a purpose.
- Avoid using `get_or_use` for critical role checks. Instead, explicitly require that the `ADMIN_INDEX` role has been initialized before any transitions that rely on it are executed.
- For the first deployment only, use a predefined address, allowing only that account to perform the initial setup. Afterward, rely exclusively on the account with the `ADMIN` role to perform subsequent upgrades. This ensures the upgrade process cannot be exploited when the admin role is uninitialized.

Developer Response The developers are yet to respond with acknowledgment or fixes.

5.1.4 V-SLC-VUL-004: Merkle tree depth mismatch limits supported leaves

Severity	Warning	**Commit**	9bc54de
Type	Usability Issue	**Status**	Fixed
Location(s)	`programs/merkle_tree.leo:7, 29`		
Confirmed Fix At	https://github.com/sealance-io/compliant-transfer-aleo/pull/96/, 3e84d30		

Description The protocol implements Merkle tree-based non-inclusion proofs for freeze list private verification. The `merkle_tree.leo` program defines a `MerkleProof` struct with a `siblings` array of 16 elements and a `MAX_TREE_DEPTH` constant set to 15. The `calculate_root_depth_siblings()` function processes Merkle proofs by iterating through sibling nodes to reconstruct the tree root.

The core issue lies in the loop bounds of `calculate_root_depth_siblings()`. The function uses `for i: u32 in 2u32..MAX_TREE_DEPTH`, which with `MAX_TREE_DEPTH = 15` only iterates from index 2 to 14 (inclusive), considering that index 0 and 1 were processed outside the loop. This means the function will never access `siblings[15]`, the final element of the 16-element array. In Merkle trees, the depth determines the maximum number of leaves: a tree of depth n supports 2^n leaves. Since the function effectively operates with depth 14 (accessing siblings[0] through siblings[14]), it can only support 2^14 = 16,384 leaves instead of the intended 2^15 = 32,768 leaves.

The TypeScript constants file confirms the intention with `MAX_TREE_SIZE = 16`, and the `genLeaves()` function calculates `maxNumLeaves = Math.floor(2 ** (MAX_TREE_SIZE - 1))`, expecting support for 2^15 leaves. However, the Leo implementation's loop constraint prevents utilizing the full sibling array capacity.

Impact Users attempting to create Merkle proofs for trees with more than 16,384 leaves will experience verification failures. The protocol cannot achieve its design goal of supporting freeze lists with up to 32,768 addresses. This limitation reduces the scalability of the freeze list mechanism by 50%.

Recommendation Increase the `MAX_TREE_DEPTH` constant from 15 to 16 to allow the loop to access all elements of the 16-element siblings array.

Developer Response The developers implemented a different solution, but it still resolves the issue. Instead of changing the value of the `MAX_TREE_DEPTH` constant, they increased the for-loop range by one and adjusted the rest of the code as necessary.

5.1.5 V-SLC-VUL-005: Missing leaf index bounds validation allows users to use indexes beyond the tree size

Severity	Warning	**Commit**	9bc54de
Type	Data Validation	**Status**	Fixed
Location(s)	`programs/merkle_tree.leo:75`		
Confirmed Fix At	https://github.com/sealance-io/compliant-transfer-aleo/pull/95, ed9a3e7		

Description The `verify_non_inclusion()` function in `programs/merkle_tree.leo` proves that an address is absent from a sorted Merkle tree by demonstrating the address would fall between two consecutive leaves. Non-inclusion proofs work by providing Merkle proofs for two adjacent positions in the sorted tree where the target address should appear if it were included. The function verifies that the target address value falls between these two adjacent leaf values, confirming the address is not in the tree since all leaves are sorted and no gaps exist between consecutive entries.

The function calculates the maximum valid leaf index as `last_index_leaf = 2^(depth1 - 1) - 1` but only enforces this bound when both proofs reference the same leaf index. In the adjacent leaf case, the function only verifies that `merkle_proofs[0].leaf_index + 1 == merkle_proofs[1].leaf_index` without checking whether either index exceeds `last_index_leaf`.

This allows proofs with leaf indices beyond the actual tree bounds to pass validation. For example, in a tree with depth 4 supporting indices 0-15, an attacker could submit proofs with indices 24 (binary `11000`) and 25 (binary `11001`). While these indices exceed the maximum valid index of 15, the 4-bit slices `1000` and `1001` (indices 8 and 9) remain adjacent and thus the proof remains legitimate as long as the target address sits between leafs 8 and 9.

Impact Attackers can submit Merkle proofs with artificially inflated leaf indices that exceed the actual tree size. While current analysis suggests this may not be directly exploitable, the missing validation increases the protocol attack surface and could enable future attack vectors if the implementation changes.

Recommendation Add explicit bounds checking to ensure both leaf indices are within the valid range before processing adjacent leaf proofs. Validate that `merkle_proofs[0].leaf_index <= last_index_leaf` and `merkle_proofs[1].leaf_index <= last_index_leaf` in the adjacent leaf branch.

Developer Response The code now validates that the most right index of the proof is smaller or equal than the most right index of the tree, as suggested.

5.1.6 V-SLC-VUL-006: Inefficient Freeze List Management Leads to False Capacity Exhaustion

Severity	Warning	**Commit**	9bc54de
Type	Usability Issue	**Status**	Fixed
Location(s)	lib/FreezeList.ts:30		
Confirmed Fix At	https://github.com/sealance-io/compliant-transfer-aleo/pull/87, e8e6d8b		

Description The protocol implements an account freeze list using a fixed-size Merkle tree. The `f_update_freeze_list()` function in the Leo contracts manages frozen accounts by maintaining a mapping of addresses to indices and a `last_index` tracker. When an account is unfrozen, its slot is freed and marked with `ZERO_ADDRESS`.

The client-side `calculateFreezeListUpdate()` function in `FreezeList.ts` scans the list to find the next insertion point. However, it always returns `lastIndex` (the highest observed index + 1) rather than searching for and reusing available `ZERO_ADDRESS` slots. This creates a strictly ascending index pattern even when freed slots exist at lower indices.

```
// Example state progression:
Initial:    [A, B, C] lastIndex=2
Unfreeze B: [A, ZERO, C] lastIndex=2
New freeze: [A, ZERO, C, D] lastIndex=3  // Should reuse index 1
```

Impact The function `calculateFreezeListUpdate` will incorrectly report that the "Merkle tree is full" when free slots exist.

Recommendation It is advisable to modify `calculateFreezeListUpdate()` to scan for and return the first available `ZERO_ADDRESS` slot.

Developer Response The developers implemented the suggested fix.

5.1.7 V-SLC-VUL-007: Maintainability concerns

Severity	Info	**Commit**	9bc54de
Type	Maintainability	**Status**	Fixed
Location(s)	▶ `programs/sealance_freezelist_registry.leo:63-74`		
Confirmed Fix At	https://github.com/sealance-io/compliant-transfer-aleo/pull/81, https://github.com/sealance-io/compliant-transfer-aleo/pull/94, d711ab3, eafac4b		

Description The project contains minor maintainability concerns in the form of incorrect comments and duplicate code.

1. **Incorrect comments:** The comments on `update_role()` in the freezelist registry program do not correspond to the function's logic
   -
2. **Duplicate code**: In `merkle_tree.test.ts`, lines `20-24` and `26-30` implement the same test case.

Impact While not functionally incorrect, the instances mentioned above may reduce clarity and readability.

Recommendation Address the incorrect comments and remove the duplicate code.

Developer Response The developers have updated the incorrect comments and removed the duplicate code.

5.1.8 V-SLC-VUL-008: Unoptimized code

Severity	Info	**Commit**	9bc54de
Type	Maintainability	**Status**	Fixed
Location(s)	programs/ ▶ sealed_report_policy.leo:272-301 ▶ sealed_threshold_report_policy.leo:396-425		
Confirmed Fix At	https://github.com/sealance-io/compliant-transfer-aleo/pull/83/, 2a8c815		

Description In `transfer_public_to_priv()`, tokens are moved from a public sender to a private recipient by first transferring them into the policy program with `transfer_from_public()`, and then forwarding them to the recipient with `transfer_public_to_private()`. Both steps also trigger pre-hook calls for program authorization. See snippet below for the implementation.

```
    let transfer_to_program: Future = token_registry.aleo/transfer_from_public(
        TOKEN_ID,
        self.caller,
        PROGRAM_ADDRESS,
        amount,
    );

    let program_owner: TokenOwner = TokenOwner {
        account: PROGRAM_ADDRESS,
        token_id: TOKEN_ID
    };

    let authorization_call_for_program: Future = token_registry.aleo/prehook_public(
        program_owner,
        amount,
        AUTHORIZED_UNTIL
    );

    let transfer_to_recipient: (token_registry.aleo/Token, Future) = token_registry.
    aleo/transfer_public_to_private(
        TOKEN_ID,
        recipient,
        amount,
        true,
    );
```

This two-step process is redundant, since the same outcome can be achieved more efficiently by directly calling`transfer_public_from_private` from the token registry.

Impact The current implementation introduces unnecessary steps to achieve the same outcome, resulting in reduced efficiency.

Recommendation Use `transfer_public_from_private()` from the token registry.

Developer Response The developers now make use of `transfer_public_from_private()`, as recommended.

A Appendix

In addition to the confirmed findings, this report includes an appendix with issues that were ultimately classified as intended behavior or invalid after discussions with the development team. We chose to include these items in the report for two main reasons:

- **Transparency.** To document the scope of our review and ensure that all potential concerns identified during the audit are visible to stakeholders.
- **Context.** To provide the development team with a clear record of which findings were analyzed, discussed, and intentionally dismissed, avoiding future confusion if similar questions arise.

A.1 Intended Behavior

A.1.1 V-SLC-APP-VUL-001: Private keys stored and transmitted in plaintext

Severity	Low	**Commit**	9bc54de
Type	Authorization	**Status**	Intended Behavior
Location(s)	`scripts/upgrade.ts:8`		
Confirmed Fix At	N/A		

Description The `contract/base-contract.ts` file implements a `BaseContract` class that manages network configurations and account credentials. The system stores private keys in environment variables via `aleo-config.js` and retrieves them through the `getPrivateKey()` method for various operations including contract deployment and program upgrades.

The `getPrivateKey()` method at `contract/base-contract.ts` directly returns private key strings from the network configuration without any encryption or protection. These plaintext private keys are then passed to various operations such as the `upgradeProgram()` function in `lib/Upgrade.ts`, which executes shell commands containing the private key as a command-line argument. The upgrade script `scripts/upgrade.ts` retrieves admin private keys using `getPrivateKey()` and passes them directly to the Leo CLI tool via command line execution.

This pattern is replicated across deployment scripts where private keys are extracted and used in plaintext.

Impact Private keys exposed in plaintext create multiple attack vectors that could compromise the entire protocol. Keys can be intercepted through process monitoring, shell history, or log files. Command-line arguments containing private keys are visible to other processes and system administrators through process lists. If compromised, these keys provide full control over protocol accounts, enabling unauthorized contract upgrades, token minting, freeze list manipulation, and theft of protocol funds.

Recommendation Implement secure key management services. Ensure private keys are never exposed in command-line arguments, log outputs, or process memory in plaintext form.

Developer Response These keys are used in testing framework (dokojs) and the mentioned scripts will not be used in the actual production deployments. The testing framework relies on using leo cli that requires key being passed as the argument.

A.1.2 V-SLC-APP-VUL-002: Constant epoch length fails to capture variable block rates

Severity	Warning	**Commit**	9bc54de
Type	Usability Issue	**Status**	Intended Behavior
Location(s)	programs/sealed_threshold_report_policy.leo:19		
Confirmed Fix At	N/A		

Description The `sealed_threshold_report_policy` emits a compliance record for an investigator, whenever a user's token transfers exceed a defined threshold within a single epoch. The epoch length is implemented as a constant, approximated from the expected block production rate of 10 seconds per block, which is taken as `8640` for 24 hours.

```
const EPOCH: u32 = 8640u32; // ~24 hours, 3600/10 * 24 = 8640
```

However, in practice the block rate may not be fixed. For example, the reference code assumes that 360 blocks (one epoch) corresponds to one hour. Yet, having a look at Aleoscan shows that the block production rate is faster than the chosen approximation and an epoch may complete in a significantly lesser time. This discrepancy highlights that the epoch definition is tightly associated to assumptions about block production speed, which can vary considerably in practice.

Impact Because the epoch duration is shorter than the expected (1 day), the users can exceed the intended daily limits. For example, they can transfer the `THRESHOLD` value (`1000000000u128`) 2-3 times in a day without triggering the compliance check.

Recommendation Avoid hardcoding epoch length as a fixed number of blocks based on assumed block times. Instead, introduce a mechanism that accounts for variability in block production rates such as making the `EPOCH` configurable and allowing an admin to update its values regularly based on current estimates.

If a constant `EPOCH` must be used, the documentation should explicitly state this limitation. Because these policy programs are intended to be customized before deployment, users should be advised to configure the `EPOCH` value to reflect a realistic projection of block production rates in the real world.

Developer Response We will keep it as intended behavior since it was a part of the product requirements, though we will make a note in the readme explaining the technical limitations.