Veridise

Auditing Report

Hardening Blockchain Security with Formal Methods

FOR

VENTURE 23

Verulink

Veridise Inc.

Sep. 09, 2024

► **Prepared For:**

Venture23, Inc
https://venture23.xyz/

► **Prepared By:**

Bryan Tan
Jacob Van Geffen

► **Contact Us:**

contact@veridise.com

► **Version History:**

Sep. 09, 2024 V2 - updated issue statuses after receiving fixes
Aug. 23, 2024 V1

Contents

1 Executive Summary

From Aug. 21, 2024 to Aug. 26, 2024, Venture23, Inc engaged Veridise to review the security of the Aleo programs in the Verulink protocol. Compared to the previous version, which Veridise has audited previously*, the new version switches the internal bookkeeping to use the Multi-Token Standard Program. Veridise conducted the assessment over 8 person-days, with 2 engineers reviewing code over 4 days on commit `a627fe58`. The auditing strategy involved an extensive manual review of the source code performed by Veridise engineers.

Project summary. The security assessment covered the Leo programs used in the Verulink protocol. Verulink is a decentralized bridging protocol between Aleo and Ethereum, allowing users to send and receive crypto tokens between Aleo and EVM-compatible blockchains. To integrate a blockchain into the protocol, three smart contracts must be deployed:

- A *token service* contract implements the business logic for user workflows such as sending and receiving tokens.
- A *bridge* contract implements the on-chain logic required for cross-chain messaging. Messages are transmitted across chains by a set of *attestors*†.
- A *holding* contract is used to hold (and later release) funds that are disputed or flagged by the protocol governance.

The Aleo version of Verulink implements the three above programs as well as an additional *council* program that acts as a multi-signature wallet that configures and controls the Aleo side of the protocol. Privileged operations on the token service and bridge contracts are performed through *token service council* and *bridge council* programs, respectively.

Code assessment. The Verulink developers provided the source code of the Verulink contracts for review. The source code appears to be mostly original code written by the Verulink developers. It contains some documentation in the form of READMEs and documentation comments on functions and mappings, as well as detailed diagrams outlining the workflow for both the attestors and the implemented bridge as a whole. To facilitate the Veridise auditors' understanding of the code, the Verulink developers gave a detailed presentation of these included diagrams and other pieces of documentation.

The source code contained a test suite, which the Veridise auditors noted covered basic functionality of the bridge.

Summary of issues detected. The audit uncovered 9 issues, 1 of which are assessed to be of high or critical severity by the Veridise auditors. Specifically, access controls on the `token_service_council.aleo/holding_release` transition can be bypassed by calling the corresponding transition on `token_service` directly (V-VVAM-VUL-001). The Veridise auditors

* The previous audit report, if it is publicly available, can be found on Veridise's website at `https://veridise.com/audits/`

† The off-chain logic was audited previously and is not in the scope of this security assessment.

also identified 3 medium-severity issues, including a location affected by a miscompilation bug in an older version of Leo (V-VVAM-VUL-002), bugs in the withdrawal limit threshold (V-VVAM-VUL-004), and missing parameter validation when executing "add token" proposals (V-VVAM-VUL-003); as well as 2 low-severity issues, 2 warnings, and 1 informational finding .

Among the 9 issues, 8 issues have been acknowledged by the Venture23, Inc, and 1 issue has been determined to be intended behavior after discussions with Venture23, Inc. Of the 8 acknowledged issues, Venture23, Inc has fixed 7 issues. This includes the 1 critical issue. Venture23, Inc does not plan to fix the other 1 acknowledged issue at this time. The 1 issue that was determined to be intended behavior has also been included in the report by the Veridise auditors, so that readers are aware of behavior which may at first be unexpected.

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
Verulink	`a627fe58`	Leo	Aleo

Table 2.2: Engagement Summary.

Dates	Method	Consultants Engaged	Level of Effort
Aug. 21–Aug. 26, 2024	Manual & Tools	2	8 person-days

Table 2.3: Vulnerability Summary.

Name	Number	Acknowledged	Fixed
Critical-Severity Issues	1	1	1
High-Severity Issues	0	0	0
Medium-Severity Issues	3	3	3
Low-Severity Issues	2	2	2
Warning-Severity Issues	2	1	0
Informational-Severity Issues	1	1	1
TOTAL	9	8	7

Table 2.4: Category Breakdown.

Name	Number
Access Control	2
Maintainability	2
Logic Error	2
Data Validation	2
Information Leakage	1

3 Audit Goals and Scope

3.1 Audit Goals

The engagement was scoped to provide a security assessment of the changes to Verulink's Leo smart contracts. In our audit, we sought to answer questions such as:

- Can attestor signatures be forged or otherwise circumvented?
- Can arbitrary users release funds from the holding contract?
- Can users mint tokens arbitrarily?
- Does the token service maintain valid state through each transition?
- Are desired withdrawal limits applied correctly?
- Can a single attestor extract arbitrary funds from the bridge?
- Can a single attestor perform a denial-of-service attack on the bridge?
- Do any transitions leak private information?

3.2 Audit Methodology & Scope

Audit Methodology. To address the questions above, our audit involved an extensive manual audit performed by human experts.

Scope.The scope of this audit is limited to the updates to Leo programs within the `aleo/programs` folder of the source code provided by the Verulink developers, which contains the Leo program implementation of the Verulink.

Methodology. Veridise auditors reviewed the reports of previous audits for Verulink, inspected the provided tests, and read the Verulink documentation. They then began a manual review of the code assisted by both static analyzers and automated testing. At the start of the audit, the Veridise auditors met with the Verulink developers to ask questions about the code. Auditors also communicated with Verulink developers during the audit to ask further code questions.

3.3 Classification of Vulnerabilities

When Veridise auditors discover a possible security vulnerability, they must estimate its severity by weighing its potential impact against the likelihood that a problem will arise. Table 3.1 shows how our auditors weigh this information to estimate the severity of a given issue.

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

In this case, we judge the likelihood of a vulnerability as follows in 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

In addition, we judge the impact of a vulnerability as follows in 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 Vulnerability Report

In this section, we describe the vulnerabilities found during our audit. For each issue found, we log the type of the issue, its severity, location in the code base, and its current status (i.e., acknowledged, fixed, etc.). Table 4.1 summarizes the issues discovered:

Table 4.1: Summary of Discovered Vulnerabilities.

ID	Description	Severity	Status
V-VVAM-VUL-001	No access controls on holding_release	Critical	Fixed
V-VVAM-VUL-002	Control flow miscompilation due to . . .	Medium	Fixed
V-VVAM-VUL-003	Unauthenticated TsAddToken . . .	Medium	Fixed
V-VVAM-VUL-004	Potential bugs in withdrawal limit . . .	Medium	Fixed
V-VVAM-VUL-005	Missing asserts in . . .	Low	Fixed
V-VVAM-VUL-006	remove_token_ts does not remove . . .	Low	Fixed
V-VVAM-VUL-007	Incorrect sequence numbers for . . .	Warning	Acknowledged
V-VVAM-VUL-008	Most consume() parameters can be private	Warning	Intended Behavior
V-VVAM-VUL-009	Minor issues	Info	Fixed

4.1 Detailed Description of Issues

4.1.1 V-VVAM-VUL-001: No access controls on holding_release

Severity	Critical	Commit	a627fe5
Type	Access Control	Status	Fixed
File(s)	aleo/programs/token_service.leo		
Location(s)	holding_release()		
Confirmed Fix At	5adb04e		

In the documentation provided by the developers, the holding contract has the following intended behavior:

> This contract is responsible for holding disputed funds and transfers. In event of transfer being initiated to an address that is blacklisted on target chain, the token service contract on target chain will lock the funds in holding contract. The funds can be released by council multisig once the dispute has been settled for the blacklisted address.

To execute this workflow, a user must call the `token_service_council.aleo/holding_release` transition. This will call the `token_service.aleo/holding_release` transition and validate that sufficient votes have been acquired from the multisig.

```
let proposal: HoldingRelease  = HoldingRelease {
    id,
    token_id,
    connector: self.caller,
    receiver,
    amount
};
let proposal_hash: field = BHP256::hash_to_field(proposal);
token_service.aleo/holding_release(token_id,receiver,amount);
council.aleo/external_execute(id, proposal_hash, voters);
```

Snippet 4.1: Relevant code in `token_service_council.aleo/holding_release`

The `token_service.aleo/holding_release` transition forwards the parameters to the `holding.aleo/release_fund` transition.

```
transition holding_release(public token_id: field, public receiver: address, public
    amount: u128) {
    holding.aleo/release_fund(receiver, token_id, amount);
}
```

Snippet 4.2: Definition of `token_service.aleo/holding_release`

Note that there are no restrictions on the callers of `token_service.aleo/holding_release`.

The `holding.aleo/release_fund` transition itself has a precondition that the caller (of `release_fund`) is the `owner` of the `holding.aleo` program. However, this implies that the `owner` of the `holding.aleo` program must be `token_service` in order for the transition call to succeed.

```
transition release_fund(public user: address, public token_id :field, public amount:
    u128) {
    multi_token_support_program.aleo/transfer_public(token_id, user, amount);

    let token_holder: Holder = Holder{
        account: user,
        token_id: token_id
    };

    return then finalize(self.caller, token_holder, amount);
}

finalize release_fund(from: address, token_holder: Holder, amount: u128) {
    // Assert only owner can release fund
    let owner: address = Mapping::get(owner_holding, OWNER_INDEX);
    assert_eq(from, owner);

    let held_amount: u128 = Mapping::get(holdings, token_holder);

    // Decrease the held amount for the token_user
    Mapping::set(holdings, token_holder, held_amount - amount);
}
```

Snippet 4.3: Definition of `holding.aleo/release_fund`

Consequently, there are no access controls on `token_service.aleo/holding_release`, since nothing prevents an arbitrary account from calling `holding_release`.

Impact Attackers can bypass the access controls on `token_service_council.aleo/holding_release` by calling `token_service.aleo/holding_release` directly. This means that funds that are held by the protocol can be released to the receiver at any time by anyone.

Recommendation Add a `finalize` function to `holding_release` that checks that the caller is the `owner`, similar to the other privileged transitions in `token_service`.

Developer Response The developers have applied the recommendation.

4.1.2 V-VVAM-VUL-002: Control flow miscompilation due to outdated Leo version

Severity	Medium	**Commit**	a627fe5
Type	Maintainability	**Status**	Fixed
File(s)	See description		
Location(s)	See description		
Confirmed Fix At	N/A		

The current implementation of the project uses Leo v1.11, which contains a bug that causes branching control flow in finalize functions to be compiled incorrectly. Specifically, some branches are compiled to straight-line code, which can result in business logic bugs.

Impact The current implementation has several locations with a `if (...) { assert (...) }` pattern, which could be compiled to `assert(...)`:

- In the finalize function for `token_service.aleo/token_send`, an `assert` is used to conditionally enforce the withdrawal limit if the total supply is above the "no-limit" threshold. This may be compiled to straight-line code, so that the withdrawal limit is *always* enforced.
- The `get_majority_count()` function in `token_bridge` and `council` is used in transition logic, and it contains an assert that each provided signature is valid on the "yay" vote or the "nay" vote. The assert is only invoked if the corresponding signer is not the zero address.
  Currently, `get_majority_count()` is not used in finalize functions and should not be affected by the bug; however, a future change may introduce a call to `get_majority_count()` in a finalize function. If the assert is compiled to straight-line code, then the assertion will fail if any of the signers are set to the zero address, as it is very improbable for the signature to be valid.

Recommendation Update to Leo v1.12 or newer, where the bug has been fixed. Note that with the bug fix, any code that reassigns variables inside of a conditional block will need to be refactored to avoid reassignment.

Developer Response The developers have switched to using Leo v1.12.

4.1.3 V-VVAM-VUL-003: Unauthenticated TsAddToken proposal parameters

Severity	Medium	**Commit**	a627fe5
Type	Access Control	**Status**	Fixed
File(s)	`aleo/programs/token_service_council.sol`		
Location(s)	ts_add_token()		
Confirmed Fix At	cf68acf		

The members of the `council.aleo` program that manages the protocol can add tokens by making `TsAddToken` proposals. After the proposal has been approved, any arbitrary account can execute the proposal by calling the `token_service_council.aleo/ts_add_token` transition.

```
let proposal: TsAddToken  = TsAddToken {
    id,
    token_id,
    min_transfer,
    max_transfer,
    outgoing_percentage,
    time,
    max_no_cap
};
let proposal_hash: field = BHP256::hash_to_field(proposal);

// Execute the proposal
token_service.aleo/add_token_ts(token_id, min_transfer, max_transfer,
    outgoing_percentage, time, max_no_cap, token_address, token_service, chain_id);
council.aleo/external_execute(id, proposal_hash, voters);
```

Snippet 4.4: Relevant code in `ts_add_token`

This transition will call the `token_service.aleo/add_token_ts` with the proposal parameters and then check that the council has approved the proposal. However, the `proposal_hash` that is used to check approval does not exactly correspond to the actual transition call that is made: the parameters `token_address`, `token_service`, and `chain_id` are not used to compute the `proposal_hash`.

Impact If a `TsAddToken` proposal has been approved but not yet executed, then an attacker that knows the parameters used to construct the `TsAddToken` struct could actually execute the proposal with a different `token_address`, `token_service`, and `chain_id` than may have been intended to be used by the council. If such an attack is performed, it would be difficult to identify the attacker directly, as anyone can call `token_service_council.aleo/ts_add_token`.

This vulnerability has a higher chance of being exploited if the attacker is a council member that proposes a malicious `TsAddToken` themself or is able to learn the proposal parameters from a council member.

Recommendation Add `token_address`, `token_service`, and `chain_id` as fields on `proposal`, so that they become involved in the computation of the `proposal_hash`.

Developer Response The developers have applied the recommendation.

4.1.4 V-VVAM-VUL-004: Potential bugs in withdrawal limit threshold logic

Severity	Medium	Commit	a627fe5
Type	Logic Error	Status	Fixed
File(s)	aleo/programs/token_service.leo		
Location(s)	token_send()		
Confirmed Fix At	c6cf363		

A user that wishes to send their tokens on the Aleo network to a different blockchain can do so by calling the `token_service.aleo/token_send` transition. To mitigate against liquidity issues, the protocol implements a withdrawal period mechanism that, for each token, limits the total amount of that token can be withdrawn within a time interval. When a token has no active withdrawal period, the next call to `token_send()` will begin a new withdrawal period lasting for a protocol-configured duration with a maximum withdrawal limit. During the withdrawal period, users will be unable to withdraw an amount of token that causes the total amount withdrawn for that token to exceed the limit. After the period has elapsed, the next `token_send()` will reset the withdrawal period.

The protocol also implements a mechanism that disables the maximum withdrawal limit if the total supply is below some protocol-configured threshold. A documentation comment motivates this threshold as "help[ing] to address the increased difficulty in withdrawing as the available liquidity decreases."

```
let withdrawal_limit: WithdrawalLimit = Mapping::get(token_withdrawal_limits,
    wrapped_token_id);

let current_supply: u128 = Mapping::get(total_supply, wrapped_token_id);
let current_height: u32 = block.height;

let max_withdrawal: u128 = get_x_percent_of_y(withdrawal_limit.percentage,
    current_supply);

let allowed_withdrawal: u128 = Mapping::get_or_use(token_snapshot_withdrawal,
    wrapped_token_id, max_withdrawal);
let snapshot_height: u32 = Mapping::get_or_use(token_snapshot_height,
    wrapped_token_id, 0u32);
let amount_withdrawn: u128 = Mapping::get_or_use(token_amount_withdrawn,
    wrapped_token_id, 0u128);

if ( current_height - snapshot_height > withdrawal_limit.duration) {
    Mapping::set(token_snapshot_withdrawal, wrapped_token_id, max_withdrawal);
    Mapping::set(token_snapshot_height, wrapped_token_id, current_height);
    Mapping::set(token_amount_withdrawn, wrapped_token_id, amount);

} else {
    Mapping::set(token_amount_withdrawn, wrapped_token_id, amount_withdrawn+amount);

}

if (current_supply >= withdrawal_limit.threshold_no_limit) {
    let withdraw_amount:u128  = Mapping::get(token_amount_withdrawn, wrapped_token_id
    );
```

```
24      let withdrawal_allowed:u128 = Mapping::get(token_snapshot_withdrawal,
        wrapped_token_id);
25      assert(withdraw_amount <= withdrawal_allowed);
26  }
27
28  // Decrease the total supply
29  Mapping::set(total_supply, wrapped_token_id, current_supply - amount);
```

Snippet 4.5: Code in `token_send()` that computes the new total supply and checks the withdrawal limit.

There are several problems with the way the "no limit" threshold logic is implemented:

1. The compared total supply value is the *instantaneous* value at the time of the `token_send()` call, which effectively raises the "no limit" threshold. If the total supply value is only above the threshold by the withdrawal limit percentage, then users can nullify the withdrawal limit by withdrawing a small amount to drop below the threshold, and then withdrawing a large amount.
   For example, if the threshold is 100, the percentage is 10%, and the total supply is 105, then the user can supposedly only withdraw up to 10 tokens with a single `token_send()` call. However, the user can bypass the withdrawal limit with two `token_send()` calls. They first withdraw 6 tokens to drop the total supply to 99. Since the new total supply is below the threshold, the user can call `token_send()` again to withdraw the remaining 99 tokens.
2. The comparison does not account for the amount of held tokens, which are counted as part of total supply. Since held tokens are not liquid, the withdrawal limits may be enforced even when liquidity is low. For example, suppose there are 1000 tokens in the total supply, of which 100 are owned by users and 900 are held. If the threshold is 500, then the withdrawal limits will be enforced; however, users can realistically only access 100 tokens, which is far below the threshold.

Impact The behaviors described above can be confusing to users. Furthermore, attacks may be able to abuse the withdrawal limit enforcement to perform denial-of-service attacks on the protocol. For example, an attacker may intentionally send over a large amount of tokens that they know will be held so that the withdrawal limits will be enforced. This may prevent users from being unable to withdraw funds in a timely manner.

Recommendation

- The developers should clarify the intended behavior in the case where the total supply drops below the threshold. If the current behavior is not intended, then the developers may want to consider comparing the total supply value at the *start* of the withdrawal period rather than the instantaneous amount.
- To reflect the liquid amount of tokens more accurately, the developers should consider implementing one of the following mutually-exclusive mitigations:
  - Do not mint any held tokens in `token_receive()`. Instead, mint them for the user when the held funds are released.
  - Deduct the total amount of held tokens from `total_supply` when comparing against `withdrawal_limit.threshold_no_limit`.

Developer Response The developers noted that the current behavior is not intended, and they have applied the recommendation. For the held tokens bug, they have opted to deduct the held amount.

4.1.5 V-VVAM-VUL-005: Missing asserts in update_other_chain_ta,ts

Severity	Low	**Commit**	a627fe5
Type	Data Validation	**Status**	Fixed
File(s)	`aleo/programs/token_service.leo`		
Location(s)	update_other_chain_ta(), update_other_chain_ts()		
Confirmed Fix At	5ac8976		

The transitions `update_other_chain_ta()` and `update_other_chain_ts` updates the cross-chain token address and token service address, respectively, for a particular `chain_id` and `token_id`. Before updating the mapping, these transitions must check that the mapping already contains the relevant chain token info. However, the check is never actually performed; there is only a statement that evaluates a `Mapping::contains` expression, which only returns a boolean value indicating whether or not the mapping contains the token info. It does not revert the transaction if the expression evaluates to false.

```
let chaintokeninfo:ChainToken = ChainToken{
    chain_id: chain_id,
    token_id: token_id
};
Mapping::contains(other_chain_token_address, chaintokeninfo);
Mapping::set(other_chain_token_address, chaintokeninfo, pad_20_to_32(token_address));
```

Snippet 4.6: Snippet from `finalize` for `update_other_chain_ta`; `update_other_chain_ts` is similar.

Impact If the council erroneously calls `update_other_chain_ta` on a non-existent `chain_id` and `token_id`, the `other_chain_token_address` mapping will be updated without a corresponding update to `other_chain_token_service`. Since other functions correctly check that both mappings contain values for relevant tokens, this missing assert will not lead to any concrete attacks by itself. However, it could leave the mappings in an unexpected state.

Recommendation Wrap `Mapping::contains(other_chain_token_address, chaintokeninfo)` in an assert statement so that `update_other_chain_ta` and `update_other_chain_ts` revert when the chain token is not in the mapping.

Developer Response The developers have applied the recommendation.

4.1.6 V-VVAM-VUL-006: remove_token_ts does not remove other chain info

Severity	Low	**Commit**	a627fe5
Type	Logic Error	**Status**	Fixed
File(s)	`token_service.leo`		
Location(s)	remove_token_ts		
Confirmed Fix At	56e5d82		

The purpose of `remove_token_ts` is to remove a token from the token service by deleting all information corresponding to that token. However, information stored in the maps `other_chain_token_address` and `other_chain_token_service` is not deleted.

Impact If a token is removed and later added back, it will maintain all information corresponding to token services for other chains that was added before the removal. This means that after the token is added back, when users send those tokens, incorrect token addresses and services may be used in the `token_send()` transition.

There is also no way to remove entries from `other_chain_token_address` and `other_chain_token_service`.

Recommendation Remove all token service information from `other_chain_token_address` and `other_chain_token_service` corresponding to the given `token_id`. Consider also adding a method to remove chain ID, token pairs.

Developer Response The developers have added a function that can be used to remove a chain ID, token pair from the two mappings. They note that the council will need to call the function separately to clean up the mappings after a token has been removed.

4.1.7 V-VVAM-VUL-007: Incorrect sequence numbers for ALEO_CHAIN_ID

Severity	Warning	**Commit**	a627fe5
Type	Data Validation	**Status**	Acknowledged
File(s)	`aleo/program/token_bridge.aleo`		
Location(s)	add_chain_tb(), publish()		
Confirmed Fix At	N/A		

The `token_bridge.aleo/publish` transition is used to queue outgoing cross-chain packets for delivery to a blockchain supported by the protocol. As part of this process, the program will assign a monotonically increasing "sequence number" to the packet, and it will also store a "bridge sequence number" that counts the total number of packets that have been queued for delivery.

```
1
2  // Assert that the packet is being sent to supported chain.
3  assert(Mapping::contains(supported_chains, destination_chain_id));
4
5  // Assert that the packet is being sent from one of the supported services
6  assert(Mapping::contains(supported_services, source_service_program));
7
8  // Get Sequence number for the destination chain
9  let target_sequence_no: u64 = Mapping::get_or_use(sequences, destination_chain_id, 1
       u64);
10
11 // Get Sequence number of this bridge
12 // This can be used to check the total outgoing messages from this bridge
       irrespective of destination chain
13 let bridge_sequence_no: u64 = Mapping::get_or_use(sequences, ALEO_CHAIN_ID, 1u64);
14
15 // ... code to build the packet that uses the target_sequence_no...
16
17 // Update sequence
18 Mapping::set(sequences, destination_chain_id, target_sequence_no + 1u64);
19 Mapping::set(sequences, ALEO_CHAIN_ID, bridge_sequence_no + 1u64);
```

Snippet 4.7: Relevant lines in `publish()`

The bridge sequence number is stored in the special `ALEO_CHAIN_ID` index of the `sequences` mapping. However, it is theoretically possible for the caller to successfully call `publish()` with the destination set to the `ALEO_CHAIN_ID`. If this happens, then the sequence number for `ALEO_CHAIN_ID` will not be updated correctly. For example, if the sequence number for `ALEO_CHAIN_ID` is currently 5, and a call to `publish()` is made to a different to a different chain, then the sequence number for `ALEO_CHAIN_ID` will be updated to 6—even though this `publish()` call did not send the packet to the Aleo chain.

Impact If the owner of the program calls the `add_chain_tb()` transition to enable `ALEO_CHAIN_ID` as a supported chain, then it is possible to call `publish()` with `ALEO_CHAIN_ID` as a destination. Off-chain applications that read `sequences[ALEO_CHAIN_ID]` as the number of messages sent to the Aleo blockchain part of the protocol will not be reading the correct number.

Recommendation Add an assertion to `add_chain_tb()` that does not allow the chain ID to be `ALEO_CHAIN_ID`.

Developer Response The developers note that the protocol assumes that `ALEO_CHAIN_ID` will not be added as a supported chain:

> No user is allowed to transfer from the aleo to aleo. There are many checks that fail when you try to `token_send` to the destination as the ALEO_CHAIN. For example: supported_chain won't have a ALEO_CHAIN_ID as a key as this is something only council can do.

4.1.8 V-VVAM-VUL-008: Most consume() parameters can be private

Severity	Warning	Commit	a627fe5
Type	Information Leakage	Status	Intended Behavior
File(s)	`aleo/programs/token_bridge.leo`		
Location(s)	consume()		
Confirmed Fix At	N/A		

The `token_bridge.aleo/consume` transition is called when an attestor calls `token_service.aleo/token_receive` transition to process a cross-chain token transfer. In both of theses transitions, the `sender`, `height`, and `sign` parameters are marked as `public`. However, it is possible to mark them as `private` to reduce the risk of information leakage.

These parameters are used to validate that the message has been signed by the attestors. The `sender` and `height` parameters are used to compute a packet hash in `consume()`. The `sign` parameter is an array of signatures signed by attestors, where each signature is used to validate a `(packet_hash, screening_passed)` message.

```
let message: InTokenMessage = InTokenMessage {
    sender_address,
    dest_token_id,
    amount,
    receiver_address
};

let in_packet: InPacket = InPacket {
    version: VERSION,
    source,
    destination,
    sequence,
    message,
    height
};

let packet_hash: field = BHP256::hash_to_field(in_packet);
```

Snippet 4.8: Relevant lines in `consume()` that compute the packet hash.

The `screening_passed` field indicates the attestor's vote on whether the transfer should be allowed or held, where majority vote wins. To count the votes, each provided signature will be verified against a `(packet_hash, true)` message and a `(packet_hash, false)` message. Because the list of signatures is provided as a public parameter, it is possible to determine what way each attestor voted.

```
let packet_hash_with_yay: field = BHP256::hash_to_field(InPacketWithScreening {
    packet_hash,
    screening_passed: true
});

let packet_hash_with_nay: field = BHP256::hash_to_field(InPacketWithScreening {
    packet_hash,
    screening_passed: false
});
```



```
10
11 for i: u8 in 0u8..SUPPORTED_THRESHOLD {
12     if (signers[i] != ZERO_ADDRESS) {
13         let yay: bool = signature::verify(signs[i], signers[i], packet_hash_with_yay)
       ;
14         let nay: bool = signature::verify(signs[i], signers[i], packet_hash_with_nay)
       ;
15         assert(yay | nay);
16         if (yay) { yay_count = yay_count + 1u8; }
17         if (nay) { nay_count = nay_count + 1u8; }
18     }
19 }
```

Snippet 4.9: Snippet in `get_majority_count()` that validates the signatures and counts the votes.

Impact It may not be intended behavior to publicize information about how each attestor voted.

The risk of information leakage can be reduced by making the `sender`, `height`, and `sign` parameters private. If `sender` and `height` were to be marked as private, attackers would more difficulty computing the original packet hash. By making `sign` private, attackers would need to first retrieve the signatures in some other way before they can check the votes.

Recommendation The developers should clarify whether the intended behavior of the protocol is to publicize information about how each attestor voted. Note that voting protocols typically publicize such information for transparency reasons. If the intention is to hide such information, then the developers should mark the `sender`, `height`, and `sign` parameters (or their corresponding parameters) in `token_receive` and `consume` as `private`.

Developer Response The developers noted that it is intended behavior to make the parameters public:

> It is the requirement from Aleo that everything be public in the bridge for security reasons.

4.1.9 V-VVAM-VUL-009: Minor issues

Severity	Info	**Commit**	a627fe5
Type	Maintainability	**Status**	Fixed
File(s)	See description		
Location(s)	See description		
Confirmed Fix At	9531a6f		

The following is a list of minor issues found throughout the codebase.

1. The variable `allowed_withdrawal` in the transition `token_service.aleo/token_receive` is defined but never used.
2. Within `token_service.aleo/token_receive`, the variable `authorize_until` is set to the magic number `4294967295u32` before being passed into `multi_token_support_program.aleo/mint_public`. The significance of this constant is not documented.
3. In the `token_service` program, the transitions `update_other_chain_ts()` and `update_other_chain_ta()` are named similarly, differing only in the last character. Furthermore, `s` and `a` are next to each other on a standard QWERTY keyboard. This makes the two functions prone to typo errors.
4. The finalize function for `token_service.aleo/token_send` has a `from` parameter that is not used.

Impact The above issues are maintainability issues, which may increase the likelihood of future bugs and increase the difficulty of extending the code base.

Recommendation Address the minor issues reported above.

Developer Response The developers have fixed the described issues.

Glossary

Aleo A layer-1 blockchain that supports privacy-aware applications based upon zero-knowledge proof technology. See the official website at https://aleo.org/. . 1

Leo A high level programming language that allows developers to create private applications based on zero-knowledge proof technology. Programs written in this language are meant to be run on the Aleo blockchain. . 1

Multi-Token Standard Program A standard Aleo program that provides logic for creating and using multiple tokens. This is similar to the Ethereum multi-token smart contracts such as ERC-1155 tokens. For more information, see https://vote.aleo.org/p/21.. 1

smart contract A self-executing contract with the terms directly written into code. Hosted on a blockchain, it automatically enforces and executes the terms of an agreement between buyer and seller. Smart contracts are transparent, tamper-proof, and eliminate the need for intermediaries, making transactions more efficient and secure. 1