Veridise

Auditing Report

Hardening Blockchain Security with Formal Methods

FOR

Leo Wallet

Multi-Token Support Program

Veridise Inc.

Jul. 18, 2024

► **Prepared For:**

Demox Labs
https://www.demoxlabs.xyz/

► **Prepared By:**

Benjamin Mariano
Alberto Gonzalez
Mark Anthony

► **Contact Us:**

contact@veridise.com

► **Version History:**

Aug. 20, 2024 V2
Jul. 18, 2024 V1

Contents

1 Executive Summary

From Jul. 15, 2024 to Jul. 18, 2024, Demox Labs engaged Veridise to review the security of their Multi-Token Support Program (or MTSP). The review covered the implementation of their MTSP, which implements a token registry where token types are registered, minted, burned, and transferred between users. Veridise conducted the assessment over 12 person-days, with 3 engineers reviewing code over 4 days on commit `6d921f3`. The auditing strategy involved extensive manual code review performed by Veridise engineers.

Project summary. The security assessment covered the token registry implementation in Leo. This covered all major functionalities of the registry, including registering tokens, minting/burning tokens, maintaining privileged roles for tokens (i.e., minters/burners), private and public transfers, and special handling for interactions with Aleo credits. The scope of the audit was limited to the code in the `multi_token_support_program/` folder.

Code assessment. The Multi-Token Support Program developers provided the source code of the program for review. The source code appears to be mostly original code written by the Multi-Token Support Program developers. It contains some very limited documentation in the form of in-line comments in the code. No other documentation was provided to the Veridise auditors for the audit.

At the moment of the audit, the code contains no automated tests.

Summary of issues detected. The audit uncovered 13 issues, 3 of which are assessed to be of high or critical severity by the Veridise auditors. Specifically, auditors discovered that token metadata cannot be updated (V-MTSP-VUL-001), prehooks incorrectly authorize expired funds (V-MTSP-VUL-002), and authorization expiration is not checked on public transfers (V-MTSP-VUL-003). The Veridise auditors also identified 1 medium-severity issue , where the recipient of funds is improperly leaked on private mints (V-MTSP-VUL-004), as well as 3 low-severity issues, 5 warnings, and 1 informational finding . Demox Labs has indicated an intent to fix these issues.

Recommendations. After auditing the protocol, the auditors had a few suggestions to improve the Multi-Token Support Program implementation.

Authorization Expirations. At the moment, the code base is intended to support authorization expirations (i.e., after some amount of time, authorized funds can become unauthorized). However, the design of the code does not easily support this. For instance, there are a number of places in the code where it is assumed that all funds in the `authorized_balances` mapping are not expired, even though this is never enforced. Furthermore, two of the highest severity issues (and an additional low severity issue) stem from misuse of the authorization expiration (V-MTSP-VUL-002, V-MTSP-VUL-003, V-MTSP-VUL-005). We suggest the developers revisit how they are choosing to support authorization expiration and if this notion is even necessary.

If it is not necessary, the code could be substantially simplified to simply include authorized and unauthorized funds (with no worrying about whether or not authorized funds are expired). If authorization expiration is required, we believe the authors should spend some time carefully considering their data structures and overall code design to better handle this requirement. As an example, currently the authorization expiration time is tracked even for unauthorized balances in the `balances` mapping. This information is unnecessary and misleading as authorization expiration only applies to authorized balances in the `authorized_balances` mapping. Design decisions such as this may lead to bugs or issues in the future which could be avoided with a cleaner design.

self.signer. Currently, the program supports the use of the `self.signer` variable for public transfers as well as for aleo.credits wrapping. However, the use of this variable should be discouraged due to its susceptibility to phishing attacks. `self.signer` refers to the original account that triggered the initial transaction, regardless of how many contracts have been called in between. This makes it possible for malicious contracts to exploit it, tricking the original user into transferring unintended funds by chaining contract calls.

Testing. Currently, the code base contains no automated tests. Without tests, small or subtle bugs that occur during refactoring/updating of the code could be entirely missed. We strongly suggest the developers implement some form of testing to lower the probability of such an error.

2 Project Dashboard

Table 2.1: Application Summary.

Name	Version	Type	Platform
Multi-Token Support Program	`6d921f3`	Leo	Aleo

Table 2.2: Engagement Summary.

Dates	Method	Consultants Engaged	Level of Effort
Jul. 15–Jul. 18, 2024	Manual	3	12 person-days

Table 2.3: Vulnerability Summary.

Name	Number	Acknowledged	Fixed
Critical-Severity Issues	0	0	0
High-Severity Issues	3	3	3
Medium-Severity Issues	1	1	1
Low-Severity Issues	3	3	1
Warning-Severity Issues	5	5	3
Informational-Severity Issues	1	1	1
TOTAL	13	13	9

Table 2.4: Category Breakdown.

Name	Number
Logic Error	6
Information Leakage	2
Maintainability	2
Access Control	1
Phishing	1
Authorization	1

3 Audit Goals and Scope

3.1 Audit Goals

The engagement was scoped to provide a security assessment of the Multi-Token Support Program implementation in Leo. In our audit, we sought to answer questions such as:

- Can a malicious user mint their own funds?
- Can a malicious user burn other users' funds?
- Can a malicious user steal funds from another user?
- Can balances across token types be mixed unintentionally?
- Can interactions with a token be blocked?
- Is user privacy maintained as expected?
- Are approvals appropriately adjusted after spends?
- Are external authorizations and authorization expirations obeyed?
- Is the protocol susceptible to phishing attacks?

3.2 Audit Methodology & Scope

Audit Methodology. To address the questions above, our audit involved multiple auditors performing detailed manual code review.

Scope. The scope of this audit is limited to the `multi_token_support_program/` folder of the source code provided by the Multi-Token Support Program developers, which contains the implementation of the Multi-Token Support Program project in Leo.

Methodology. Veridise auditors reviewed audit reports of protocols similar to Multi-Token Support Program and discussed the intentions of the project with the developers. They then began a manual review of the code. During the audit, the Veridise auditors met with the Multi-Token Support Program developers to ask questions about the code, share issues found, and clarify developer intentions.

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.

In this case, we judge the likelihood of a vulnerability as follows in Table 3.2:

In addition, we judge the impact of a vulnerability as follows in Table 3.3:

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

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

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-MTSP-VUL-001	Token metadata cannot be updated	High	Fixed
V-MTSP-VUL-002	Prehook authorizes more funds than intended	High	Fixed
V-MTSP-VUL-003	No authorization expiration check on transfer	High	Fixed
V-MTSP-VUL-004	Privacy leakage on private mint	Medium	Fixed
V-MTSP-VUL-005	Incorrect authorization expiration on transfer	Low	Fixed
V-MTSP-VUL-006	Privacy leakage on private transfer	Low	Acknowledged
V-MTSP-VUL-007	Centralization Risk	Low	Acknowledged
V-MTSP-VUL-008	Possible phishing attacks	Warning	Fixed
V-MTSP-VUL-009	Error-prone manual changes to compiled code	Warning	Acknowledged
V-MTSP-VUL-010	Bad assumption on Aleo Credits total supply	Warning	Acknowledged
V-MTSP-VUL-011	Flooding the market with fake tokens	Warning	Fixed
V-MTSP-VUL-012	Unsafe typecast	Warning	Fixed
V-MTSP-VUL-013	Unnecessary code	Info	Fixed

4.1 Detailed Description of Issues

4.1.1 V-MTSP-VUL-001: Token metadata cannot be updated

Severity	High	Commit	6d921f3
Type	Logic Error	Status	Fixed
File(s)	`main.leo`		
Location(s)	finalize_update_token_management()		
Confirmed Fix At	48c7e7c		

The function `finalize_update_token_management()` is used to update the admin and external authorization party addresses for a given token. However, the changes to the state are never actually saved back to the `registered_tokens` mapping, meaning no changes are ever registered.

```
async function finalize_update_token_management(
  token_id: field,
  admin: address,
  external_authorization_party: address,
  caller: address
) {
  let token: TokenMetadata = registered_tokens.get(token_id);
  assert_eq(caller, token.admin);

  let new_metadata: TokenMetadata = TokenMetadata {
    token_id: token_id,
    name: token.name,
    symbol: token.symbol,
    decimals: token.decimals,
    supply: token.supply,
    max_supply: token.max_supply,
    admin: admin,
    external_authorization_required: token.external_authorization_required,
    external_authorization_party: external_authorization_party
  };
}
```

Snippet 4.1: Implementation of `finalize_update_token_management()`

Impact Neither the admin nor the external authorization party can ever be updated for a token.

Recommendation Save the changes to `registered_tokens`.

Developer Response Developers implemented the recommendation.

4.1.2 V-MTSP-VUL-002: Prehook authorizes more funds than intended

Severity	High	**Commit**	6d921f3
Type	Logic Error	**Status**	Fixed
File(s)	`main.leo`		
Location(s)	finalize_prehook_public()		
Confirmed Fix At	46ac7d1		

The prehook is run to authorize unauthorized funds. The authorizing party calls the function `finalize_prehook_public` to authorize a specific `amount` of funds. The snippet below shows where the balance of the user is updated to reflect this authorization.

```
let authorized_balance: Balance = authorized_balances.get_or_use(balance_key,
    default_balance);
let new_authorized_balance: Balance = Balance {
  token_id: owner.token_id,
  account: owner.account,
  balance: authorized_balance.balance + amount,
  authorized_until: authorized_until
};
```

Snippet 4.2: Snippet from `finalize_prehook_public()`

This rewrites the `authorized_until` value for *all* funds owned by a user (not just the `amount` specified by the authorizing party when calling the prehook).

Impact This could cause an external authority to accidentally authorize far more funds than intended if the existing funds in `authorized_balances` have expired.

Recommendation Ensure that the prehook only authorizes the intended amount of funds.

Developer Response The developers have fixed the issue.

4.1.3 V-MTSP-VUL-003: No authorization expiration check on transfer

Severity	High	**Commit**	6d921f3
Type	Logic Error	**Status**	Fixed
File(s)	`main.leo`		
Location(s)	See issue description.		
Confirmed Fix At	0cf9c43		

For any token where `external_authorization_required` is `true`, any transfer should only use "authorized" funds. This means that all funds should be pulled from the `authorized_funds` mapping *and* the `authorized_until` value should be checked to ensure that fund authorization has not yet expired. None of the public transfer functions contain the required check on the `authorized_until` value.

Impact This will allow illegal transfers of expired funds.

Recommendation Add checks on transfers to ensure funds are not expired.

Developer Response The developers have fixed the issue.

4.1.4 V-MTSP-VUL-004: Privacy leakage on private mint

Severity	Medium	**Commit**	6d921f3
Type	Information Leakage	**Status**	Fixed
File(s)	`main.leo`		
Location(s)	finalize_mint_private()		
Confirmed Fix At	610ed84		

When minting private tokens, both the `recipient` and `authorize_until` values are needlessly passed to the `finalize` function, which leaks this information publicly on chain.

```
async function finalize_mint_private(
  token_id: field,
  recipient: address,
  amount: u128,
  external_authorization_required: bool,
  authorized_until: u32,
  caller: address
)
```

Snippet 4.3: Function signature for `finalize_mint_private()`

Impact Both the recipient and authorization expiration time for minted tokens are leaked.

Recommendation Remove these as arguments to the `finalize` function.

Developer Response The developers have removed the `recipient` parameter and have added a check on the `authorized_until` parameter, making it a necessary parameter.

4.1.5 V-MTSP-VUL-005: Incorrect authorization expiration on transfer

Severity	Low	**Commit**	6d921f3
Type	Logic Error	**Status**	Fixed
File(s)	`main.leo`		
Location(s)	See issue description.		
Confirmed Fix At	0cf9c43		

In this protocol, users can have both "authorized" and "unauthorized" funds (which are stored in `authorized_balances` and `balances` respectively). Any time a transfer of funds happens, those transferred funds should be marked as "unauthorized" (unless the token does not require authorization at all). Below is an example of where the recipient's balance is adjusted after a transfer.

```
let recipient_balance: Balance = authorization_required ? balances.get_or_use(
    recipient_balance_key, default_balance) : authorized_balances.get_or_use(
    recipient_balance_key, default_balance);
let new_recipient_balance: Balance = Balance {
  token_id: token_id,
  account: recipient,
  balance: recipient_balance.balance + amount,
  authorized_until: balance.authorized_until
};
```

Snippet 4.4: Snippet from `finalize_transfer_public()`

As shown, the `authorized_until` value receives the value `balance.authorized_until` (as opposed to `recipient_balance.authorized_until`). Note that this happens on most of the transfer functions in the file that write to the public balances. The same happens on minting of public balances.

It should also be noted that, as a result of this, the value `default_expiration` is never actually used. As explained below, it is never really necessary to compute a `default_expiration` (any constant will do, or even better a refactor that avoids the need for tracking expiration timeline for unauthorized balances).

Impact Because the `authorized_until` value is only checked on values in `authorized_balances`, and transfers will only write directly to `authorized_balances` if the token does *not* require authorization, this should not have safety impacts on the code. However, the logic is convoluted and error-prone.

Recommendation Write `recipient_balance.authorized_until` as the `authorized_until` value instead of `balance.authorized_until`.

Please note that even with this change, there is still a weird behavior where on any transfer *all* funds will require re-authorization (even if they were previously authorized). This may be cumbersome and unnecessary. A larger refactor/rethinking of the authorization logic may be warranted here.

Developer Response The developers have made the suggested change to `recipient_balance.authorized_until` and have acknowledged the additional recommendation.

4.1.6 V-MTSP-VUL-006: Privacy leakage on private transfer

Severity	Low	Commit	6d921f3
Type	Information Leakage	Status	Acknowledged
File(s)	main.leo		
Location(s)	finalize_transfer_private()		
Confirmed Fix At	N/A		

On private transfers, the `input_token_authorized_until` value is passed to the `finalize` function for verification.

```
async function finalize_transfer_private(
  external_authorization_required: bool,
  input_token_authorized_until: u32
)
```

Snippet 4.5: Function signature for `finalize_transfer_private()`

Impact If a token uses a recognizable authorization expiration time, this could leak information about the token type being transferred.

Recommendation Create documentation to make it clear to users that this is a possible source of leakage, likely with best practices to avoid identifiable authorization expiration times if possible.

Developer Response The developers intend to make a note of this in their documentation as recommended.

4.1.7 V-MTSP-VUL-007: Centralization Risk

Severity	Low	**Commit**	6d921f3
Type	Access Control	**Status**	Acknowledged
File(s)	`main.leo`		
Location(s)	See issue description.		
Confirmed Fix At	N/A		

1. Similar to many projects, each token type is assigned an administrator who is given special permissions. The administrator for a token has the ability to mint and burn tokens (including burning tokens for *any* user). They can also update the metadata for a token, including changing the administrator and updating the external authorization party. The administrator for a token can also grant other users minting/burning privileges for that token.
2. The code also includes a set of roles that can be granted to arbitrary accounts. These roles have the authority to mint or burn tokens but they should not be able to bypass the authorization external contract. However, an issue in `mint_private` allows that by allowing these roles to pass an arbitrary `authorized_until` value.

Impact

1. If a private key were stolen, a hacker would have access to sensitive functionality that could compromise the project. For example, a malicious administrator could simply burn the funds of users they did not like.
2. Roles can bypass the authorization external contract which should be outside of their privileges.

Recommendation

1. As these are all particularly sensitive operations, we would encourage the developers to utilize a decentralized governance or multi-sig contract as opposed to a single account, which introduces a single point of failure.
2. If the `token_id` requires authorization, then validate that `authorized_until` is zero in `mint_private`.

Developer Response The developers note that the administrator address can be a contract, which would allow token developers to implement their own multi-sig within that contract. They have also removed the ability of an administrator to set an arbitrary `authorized_until` value in commit `97aed54b239668f550c479c4461b688e93f0efe2`.

4.1.8 V-MTSP-VUL-008: Possible phishing attacks

Severity	Warning	**Commit**	6d921f3
Type	Phishing	**Status**	Fixed
File(s)	`main.leo`		
Location(s)	deposit_credits_public(), transfer_public_as_signer()		
Confirmed Fix At	b494569		

The function `deposit_credits_public()` calls `credits.aleo/transfer_public_as_signer` to transfer credits from the user to this contract. Then the `self.caller` is given the wrapped funds on this contract. This is prone to phishing attacks, where other contracts gain the trust of a sender and then those malicious contracts call this contract to steal funds from them. A similar concern exists for `transfer_public_as_signer()`, where user funds are transferred away from the *signer* (as opposed to the caller).

Impact This type of phishing attack could lead to lost funds for users.

Recommendation For `deposit_credits_public`, deposit the credits for `self.signer` instead of `self.caller`. For `transfer_public_as_signer`, avoid allowing transfers on behalf of signers. Approvals should be able to achieve this in a safer way.

Developer Response The developers have deposited the credits for `self.signer` instead of `self.caller` in `deposit_credits_public`. The developers have opted to keep `transfer_public_as_signer` as is for the time being.

4.1.9 V-MTSP-VUL-009: Error-prone manual changes to compiled code

Severity	Warning	Commit	6d921f3
Type	Maintainability	Status	Acknowledged
File(s)	`main.leo`		
Location(s)	finalize_withdraw_credits_private(), finalize_deposit_credits_private()		
Confirmed Fix At	N/A		

For two finalize functions (`finalize_deposit_credits_private()` and `finalize_withdraw_credits_private()`), instead of passing in the `Future` from the transition and calling `await` on it, the body of the `finalize` is left as `assert(true)` as shown below.

```
async function finalize_deposit_credits_private() {
  assert(true);
}
```

Snippet 4.6: Implementation of `finalize_deposit_credits_private()`

Then, to make sure the `await` happens as expected, the developers manually adjust the compiled code to include the `await`.

Impact This could lead to issues in the future if this manual step is ever forgotten.

Recommendation Include the `await` logic in the code itself.

Developer Response At the moment, the developers believe this is unnecessary due to a known issue with the Leo compiler. They have filed a bug ticket and intend to avoid this once the bug is fixed.

4.1.10 V-MTSP-VUL-010: Bad assumption on Aleo Credits total supply

Severity	Warning	**Commit**	6d921f3
Type	Logic Error	**Status**	Acknowledged
File(s)	`main.leo`		
Location(s)	finalize_initialize()		
Confirmed Fix At	46ac7d1		

On initializing the program, a special token is created for Aleo credits with the metadata shown below.

```
let credits_reserved_token: TokenMetadata = TokenMetadata {
  token_id: CREDITS_RESERVED_TOKEN_ID,
  name: 1095517519u128,
  symbol: 1095517519u128,
  decimals: 6u8,
  supply: 1_500_000_000_000_000u128,
  max_supply: 1_500_000_000_000_000u128,
  admin: multi_token_support_program_v1.aleo,
  external_authorization_required: false,
  external_authorization_party: multi_token_support_program_v1.aleo
};
```

Snippet 4.7: Snippet from `finalize_initialize()`

As shown, the total supply is hard-coded to `1.5` billion (accounting for the `6` decimals). This is consistent with the current total supply of Aleo credits. However, as indicated here, Aleo intends to create more Aleo credits over the next `10` years, meaning the supply may grow in coming years. There is no way to adjust this supply in the current code.

Impact The total supply for Aleo credits may come out-of-sync with the actual total supply.

Recommendation Add some mechanism to update the supply for Aleo credits. As an important note, if this change is made, it should be noted that the function `initialize` can currently be called multiple times (which would reset the supply). As a result, if this change is made, it should include some check that prevents initializing the Aleo credits metadata multiple times.

Developer Response The developers have split the credits token wrapper into a separate program and set the max supply as 10 billion. They believe this is a reasonable upper limit given the total Aleo supply and the fact that the MTSP will likely never hold the entirety of the Aleo credits supply.

4.1.11 V-MTSP-VUL-011: Flooding the market with fake tokens

Severity	Warning	**Commit**	6d921f3
Type	Authorization	**Status**	Fixed
File(s)	`main.leo`		
Location(s)	finalize_register_token()		
Confirmed Fix At	N/A		

In this protocol, anyone can register a token. While they cannot register an existing ID, they are free to register as many tokens as they want.

Impact A malicious user could flood the market with spam tokens, or even tokens intended to trick users into thinking they are legitimate.

Recommendation It might be worthwhile restricting token creation to some set of users (or require some voting process to accept tokens to be registered). There are tradeoffs here in terms of centralization risks that should be considered.

Developer Response The developers have the following response: "We prioritized decentralization and flexibility when designing the program, and recognize this is an unfortunate tradeoff. We expect the community - wallets, dApps, exchanges - to validate certain tokens before supporting them and to warn users before transacting with an unverified token."

4.1.12 V-MTSP-VUL-012: Unsafe typecast

Severity	Warning	**Commit**	6d921f3
Type	Logic Error	**Status**	Fixed
File(s)	`main.leo`		
Location(s)	finalize_burn_public()		
Confirmed Fix At	cd07ef7		

The following line is used to compute the amount of "locked" balance remaining after a burn.

```
1 let remaining_after_burn: i128 = balance.balance as i128 - amount as i128;
```

Snippet 4.8: Snippet from `finalize_burn_public()`

The balance is cast to an `i128` (from `u128`) because it is possible to burn more than the balance (in that case, the remaining funds are burned from "authorized" balances). However, both `amount` and `balance.balance` are `u128`, which will cause the cast to revert if either is larger than the max `i128`.

Impact If an account has more than the max `i128` in balance, none of its balance can be burned, even by accounts with the "burning" role who should be able to do so. In theory, this could be worked around by the account choosing to split their assets via a transfer, but that would require the buy-in of the account themselves which may not be desired.

Recommendation Calculate the amounts to burn from the "locked" and "authorized" balances without casting to `i128`.

Developer Response The developers have updated the code to avoid the revert.

4.1.13 V-MTSP-VUL-013: Unnecessary code

Severity	Info	**Commit**	6d921f3
Type	Maintainability	**Status**	Fixed
File(s)	See issue description		
Location(s)	See issue description		
Confirmed Fix At	N/A		

Description The following pieces of code are unnecessary:

- `finalize_transfer_from_public_to_private`:
  - Assertion `current_allowance >= amount`.
  - Assertion `balance.balance >= amount`.
  - Assertion that `owner == balance.account`.
- `finalize_unapprove_public`:
  - Assertion `current_allowance >= amount`.
- `finalize_transfer_from_public`:
  - Assertion `current_allowance >= amount`.
  - Assertion `balance.balance >= amount`.
  - Assertion that `owner == balance.account`.
- `finalize_mint_public`:
  - Assignment to local variable `is_admin` (it is only ever used in the `if` statement which directly follows it).
  - Creation of `TokenOwner` struct in `finalize` instead of in `transition` (this is inconsistent with other places in the code which construct this in `transition`).
- `finalize_mint_private`:
  - Assignment to local variable `authorization_required` (it is only ever used in the assertion directly following it)

Impact These assertions add unnecessary complexity to the code.

Developer Response The developers have implemented the suggested fixes.

