Veridise

Auditing Report

Hardening Blockchain Security with Formal Methods

FOR

Leo Wallet

Aleo Staking

Veridise Inc.

June 7, 2024

► **Prepared For:**

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

► **Prepared By:**

Mark Anthony
Nicholas Brown
Bryan Tan

► **Contact Us:** contact@veridise.com

► **Version History:**

Jun. 7, 2024	V1.01 - update status of V-DEM-VUL-001
Apr. 26, 2024	V1

Contents

1 Executive Summary

From Apr. 15, 2024 to Apr. 19, 2024, Demox Labs engaged Veridise to review the security of their Aleo Staking project. Veridise conducted the assessment over 3 person-weeks, with 3 engineers reviewing code over 1 week. The auditing strategy involved extensive manual code review of the source code performed by Veridise engineers.

Project summary. The security assessment covered three parts of the Aleo Staking project:

- The `arc_0038.aleo` program, an Aleo instructions implementation of ARC-38. Conceptually, this Aleo program implements a liquidity pool that acts as a delegator to an Aleo validator node. Users can provide Aleo Credits to the pool to obtain "shares" in the pool. The pool will then automatically stake the credits and collect staking rewards on behalf of the users. Users can redeem their shares to reobtain Aleo credits. The administrator of the pool will collect a percentage of the earned rewards as commission.
- Pull request AleoHQ/snarkVM#2402, which updates the Aleo credits program to allow Aleo programs to transfer Aleo credits from the signer of the transaction.
- Pull request AleoHQ/snarkVM#2406, which makes some further changes to the Aleo credits program as well as snarkVM itself to facilitate the implementation of ARC-38.

Code assessment. The Aleo Staking developers provided the source code of the three parts mentioned above for review. The code appears to be original, written by the Aleo Staking developers. It contains some documentation in the form of READMEs and documentation comments on functions and storage variables. To facilitate the Veridise auditors' understanding of the code, the Aleo Staking developers met with the auditors and explained the intended behavior of the project.

The source code contained a test suite, which the Veridise auditors noted covered a variety of scenarios which may occur during the program execution.

Summary of issues detected. The audit uncovered 6 issues, 3 of which are assessed to be of low severity by the Veridise auditors. Specifically, it is possible for a malicious user to perform a temporary denial of service attack when the pool is switching validators (V-DEM-VUL-003). Additionally, there is an issue where inefficiencies and inconsistent bookkeeping can occur due to the way in which batches are determined (V-DEM-VUL-002), as well as a potential situation where the changes in AleoHQ/snarkVM#2406 could cause reverts that would not have occurred before the changes (V-DEM-VUL-001). The Veridise auditors also identified 2 warnings, including a potential revert due to subtraction overflow (V-DEM-VUL-005) and missing data validation (V-DEM-VUL-004), as well as 1 informational finding.

Demox Labs has addressed all of the identified issues.

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
arc_0038.aleo	`10c3b6a8`	Aleo	Aleo
AleoHQ/snarkVM#2402	`761d4932`	Rust	N/A
AleoHQ/snarkVM#2406	`bb9bf602`	Rust	N/A

Table 2.2: Engagement Summary.

Dates	Method	Consultants Engaged	Level of Effort
Apr. 15 - Apr. 19, 2024	Manual & Tools	3	3 person-weeks

Table 2.3: Vulnerability Summary.

Name	Number	Acknowledged	Fixed
Critical-Severity Issues	0	0	0
High-Severity Issues	0	0	0
Medium-Severity Issues	0	0	0
Low-Severity Issues	3	3	2
Warning-Severity Issues	2	2	2
Informational-Severity Issues	1	1	1
TOTAL	6	6	5

Table 2.4: Category Breakdown.

Name	Number
Denial of Service	2
Logic Error	1
Usability Issue	1
Data Validation	1
Maintainability	1

3 Audit Goals and Scope

3.1 Audit Goals

The engagement was scoped to provide a security assessment of the three components of the Aleo Staking project as described in section 1. In our audit, we sought to answer questions such as:

- Is it possible for user funds to be locked in the arc0038 program?
- Can an attacker steal funds from the arc0038 program?
- Is it possible for a malicious user to block deposits and/or withdrawals in the arc0038 program?
- Will the arc0038 program ever revert when passed valid inputs?
- Do the arc0038 program's mappings always accurately implement the bookkeeping for the pool?
- Do the changes to snarkVM include any other unintended changes?

3.2 Audit Methodology & Scope

Audit Methodology. To address the questions above, our audit involved a manual analysis by human experts.

Scope. The scope of this audit covers the following files and folders:

- The `arc_0038.aleo` file provided by the Aleo Staking developers, which contains the Aleo instructions implementation of the arc0038 program.
- Changes to `synthesizer/program/src/resources/credits.aleo` in AleoHQ/snarkVM#2402.
- Changes to `synthesizer/program/src/resources/credits.aleo` and `synthesizer/src/vm/finalize.rs` in AleoHQ/snarkVM#2406.

The Aleo Staking developers also provided accompanying test files, documentation, and other reference materials; these were all excluded from the scope of the audit. During the audit, the Veridise auditors referred to the excluded files but assumed that they have been implemented correctly.

Methodology. Veridise auditors reviewed the reports of previous audits for Aleo Staking*, inspected the provided tests, and read the Aleo Staking documentation. They then began an extensive manual review of the code. During the audit, the Veridise auditors regularly met with the Aleo Staking developers to ask questions about the code.

* Prior to this audit performed by Veridise, Demox Labs engaged another security firm to review the security of Aleo Staking.

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-DEM-VUL-001	Effect of unbond_public not simulated	Low	Acknowledged
V-DEM-VUL-002	Unexpected behavior when computing next batch	Low	Fixed
V-DEM-VUL-003	Denial of service attack on create_withdraw_claim	Low	Fixed
V-DEM-VUL-004	Next validator can be set to the pool itself	Warning	Fixed
V-DEM-VUL-005	Some situations cause subtraction overflow in b...	Warning	Fixed
V-DEM-VUL-006	Unnecessary terms included in calculation	Info	Fixed

4.1 Detailed Description of Issues

4.1.1 V-DEM-VUL-001: Effect of unbond_public not simulated

Severity	Low	**Commit**	bb9bf60
Type	Logic Error	**Status**	Acknowledged
File(s)	`synthesizer/src/vm/finalize.rs`		
Location(s)	prepare_for_execution()		
Confirmed Fix At	N/A		

Pull request AleoHQ/snarkVM#2406 moves the maximum committee size check in `credits.aleo` to a new function `prepare_for_execution()` in the snarkVM transaction execution logic. The `prepare_for_execution()` function will be used to determine whether a transaction should be continued to be processed before executing the finalize portion of a transaction.

The maximum committee size is now checked by simulating the effects of the `credits.aleo/bond_public` calls on committee membership. Specifically, a set of potentially new validator addresses will be collected from the first argument of each `credits.aleo/bond_public` call in the transaction. The number of new committee members will be determined by comparing against the existing set of validators in the `credits.aleo/committee` mapping (prior to execution of the finalize functions). If the resulting committee size is larger than the maximum committee size, then the transaction will be rejected.

However, the logic in `prepare_for_execution()` does not also simulate the effect of `credits.aleo/unbond_public`, which a committee member can use to remove themself from the committee.

Impact If the committee is already at its maximum size, then a transaction that contains a `credits.aleo/unbond_public` call (to allow a committee member to remove themself from the committee) followed by `credits.aleo/bond_public` call (to allow another validator to add themself to the now-vacant committee slot) will be rejected. Note that this sequence of calls would be allowable under the previous implementation of the committee size check in `credits.aleo`.

Recommendation Consider adding code to also simulate the effect of `credits.aleo/unbond_public`. Note that this is trickier to implement compared to simulating the effect of `credits.aleo/bond_public`, as the `credits.aleo/unbond_public` case involves checking whether the validator is attempting to partially or fully unbond. The maximum committee size must also be checked immediately after each `bond_public` call that occurs after an `unbond_public` call, rather than being performed only once at the end.

Developer Response The developers acknowledged the issue but will not fix it at this time. The related logic will be changed again in a future Aleo Request for Comments (ARC) proposal.

```
// Check if the execution has any ‘bond_public‘ transitions, and collect
// the unique validator addresses if so.
// Note: This does not dedup for existing and new validator addresses.
let bond_validator_addresses: HashSet<_> = execution
    .transitions()
    .filter_map(|transition| match transition.is_bond_public() {
        // Check the first input of the transition for the validator address.
        true => match transition.inputs().first() {
            Some(Input::Public(_, Some(Plaintext::Literal(Literal::Address(address),
    _)))) => Some(address),
            _ => None,
        },
        false => None,
    })
  .collect();

match bond_validator_addresses.is_empty() {
    false => {
        // Retrieve the committee members from storage.
        let committee_members = /* ... */;
        // Get the number of new validators being bonded to.
        let num_new_validators =
            bond_validator_addresses.into_iter().filter(|address| !committee_members.
    contains(address)).count();
        // Compute the next committee size.
        let next_committee_size = committee_members.len().saturating_add(
    num_new_validators);
        // Check that the number of new validators being bonded does not exceed the
    maximum number of validators.
        match next_committee_size > Committee::<N>::MAX_COMMITTEE_SIZE as usize {
            true => Err(anyhow!("Call to ’credits.aleo/bond_public’ exceeds the
    committee size")),
            false => Ok(()),
        }
    }
    true => Ok(()),
}
```

Snippet 4.1: Code in `prepare_for_execution()` that implements the new check.

4.1.2 V-DEM-VUL-002: Unexpected behavior when computing next batch height

Severity	Low	**Commit**	10c3b6a
Type	Usability Issue	**Status**	Fixed
File(s)	`main.leo`		
Location(s)	withdraw_public		
Confirmed Fix At	9b93643		

When the pool is currently bonded, a user can initiate a withdrawal of their credits in exchange for burning some of their shares using the `withdraw_public` transition. This will invoke `credits.aleo/unbond_public` to unbond the equivalent credit value of their shares and then create a "withdrawal claim" for the user. When `credits.aleo/unbond_public` is called, there is a waiting period of 360 blocks before the unbonded credits can be claimed by the pool. Subsequent calls to `credits.aleo/unbond_public` will also push back the waiting time for the original call.

To prevent malicious actors from pushing this deadline back indefinitely through repeated calls to `withdraw_public`, withdrawals are batched so that there is an enforced period of a minimum of 360 blocks at the end of a batch where no user can initiate a withdrawal. Specifically, the end time for the batch is calculated as the current block number rounded up to the next multiple of 1000. The intention is that at the end of the batch, the users will be able to successfully call `claim_unbond` to have the pool claim the unbonded credits, and that the pool will have sufficient liquidity to process all withdrawal claims created in the batch.

```
// Update withdrawals mappings
div block.height 1000u32 into r69;
mul r69 1000u32 into r70;
add r70 1000u32 into r71;
ternary r7 r71 r5 into r72;
set r72 into current_batch_height[0u8];
cast r2 r72 into r73 as withdrawal_state;
set r73 into withdrawals[r3];
```

Snippet 4.2: Lines at the end of `withdraw_public` that compute the end of a new batch. Note that `claim_unbond` will remove the `current_batch_height[0u8]` entry.

There are several problems with the way the end time of a batch is determined. If the batch is initiated in blocks 641-1000 of the current thousand block cycle, any future calls to `withdraw_public` in this new batch will fail because they will be within 360 blocks of the end of the batch. This means that only a single withdrawal will occur in this batch, and further withdrawals will be blocked for 360 blocks to accommodate the withdrawal. Additionally, the recorded end time of the batch will be earlier than the actual time when `claim_unbond` may be called to complete the batch.

For example, if the batch is initiated at block 642, then the end time of the batch will be recorded as 1000, even though users will need to wait until at least block 1002 for the unbonding to complete.

Impact When a withdrawal claim is created in `withdraw_public`, its claim time is set to the newly computed end time of the batch. If a batch is initiated between blocks 641-999 of a batch, then the withdrawal claim can be claimed early at block 1000, even though the unbonded

```
// Determine if the withdrawal can fit into the current batch
get.or_use current_batch_height[0u8] 0u32 into r5;
add block.height 360u32 into r6;
is.eq r5 0u32 into r7;
gte r5 r6 into r8;
or r7 r8 into r9;
assert.eq r9 true;
```

Snippet 4.3: Lines at the beginning of `withdraw_public`'s finalize function that check that the requested unbonding will be completed before the end of the current batch (if it exists).

credits have not been claimed by the pool yet. For example, if a malicious user creates a new batch at block 999, then they can call `claim_withdrawal_public` at block 1000 to complete their withdrawal claim and transfer credits out of the pool. This can potentially cause a liquidity problem in the pool, as the pool may not have enough credits to cover other active withdrawal claims until the unbonding period has passed and the pool's unbonded credits have been claimed.

```
function claim_withdrawal_public:
    input r0 as address.public;
    input r1 as u64.public;
    call credits.aleo/transfer_public r0 r1 into r2;
    async claim_withdrawal_public r2 r0 r1 into r3;
    output r3 as arc_0038.aleo/claim_withdrawal_public.future;

finalize claim_withdrawal_public:
    input r0 as credits.aleo/transfer_public.future;
    input r1 as address.public;
    input r2 as u64.public;
    await r0;
    get withdrawals[r1] into r3;
// Assert that the withdrawal amount matches the mapping, and the withdrawal is ready
     to be claimed
    gte block.height r3.claim_block into r4;
    assert.eq r4 true;
    assert.eq r3.microcredits r2;
```

Snippet 4.4: Relevant code in `claim_withdrawal_public`. The `r3.claim_block` in the finalize function will be equal to the end time of a batch. Note that the validity of a withdrawal claim is solely determined by whether the current block height is past the end time of a batch and whether the requested amount is correct.

Recommendation Ensure that the claim block stored in the `withdrawals` mapping is always consistent with the actual end time of a batch. Some ways to do this include:

- When computing the `current_batch_height`, add 1000 to the current block height instead of rounding up to the nearest multiple of 1000.
- Keep the same logic, but prevent a new batch from being created within 360 blocks of the next multiple of 1000

Developer Response The developers changed the calculation of the `current_batch_height` to add 1000 to the current block height.

4.1.3 V-DEM-VUL-003: Denial of service attack on create_withdraw_claim

Severity	Low	**Commit**	10c3b6a
Type	Denial of Service	**Status**	Fixed
File(s)	`arc_0038.aleo`		
Location(s)	create_withdraw_claim		
Confirmed Fix At	910dcab		

After the pool has become fully unbonded and claimed all unbonded credits, users that did not make any previous withdraw requests while the pool was bonded can now do so using the `create_withdraw_claim` transition. This can be immediately followed by a call to `claim_withdrawal_public` to have the requested credits transferred back to the user.

The auditors have identified a hypothetical attack that may be used to prevent legitimate users from calling `create_withdraw_claim`, forcing them to wait until the next bonding/unbonding period is complete until funds can be withdrawn. An attacker can carry out such an attack by repeatedly depositing credits with `deposit_public` and then withdrawing the same amount of credits with `create_withdraw_claim` and `claim_withdrawal_public`. The attack is described in more detail below.

Hypothetical Attack Scenario The attack makes the same assumptions as `create_withdraw_claim`, which is that:

- The pool is not bonded to any validator in `credits.aleo`.
- The pool does not have a pending unbonding request in `credits.aleo`.

Under these assumptions, suppose the pool has the following quantities:

- Let `W` be the initial value of `pending_withdrawal[1]`
- Let `D` be the initial value of `pending_deposits[0]`
- Let `C` be the initial credit balance of the pool address, as stored in `credits.aleo`.
- Let `B` be the initial value of `total_balance[0]`

The attack is carried out as follows:

1. An attacker calls `deposit_public` to deposit `x` credits and get `y` shares, so that the pool's state now contains the following values:

```
pending_deposits[0] = D + x
pool's credit balance = C + x
```

   Although `deposit_public` will also increase `total_balance[0]` by the bonding rewards, this attack scenario assumes that the pool is currently not bonded. Thus, the value of `total_balance[0]` will not change.
2. Then they call `create_withdraw_claim` to burn `y` shares and request a withdrawal of `x` credits, so that the pool's state now contains the following values:

```
pending_withdrawal[1] = W + x
total_balance[0] = B - x
```

   Note that the withdrawal request cannot be claimed until the next block.
3. After at least one block passes, they call `claim_withdrawal_public` to complete their withdrawal, so that the pool's state now contains the following values:


```
pool's credit balance = C
pending_withdrawal[1] = W
```

Altogether, the above calls will move x credits from the `total_balance[0]` to the `pending_deposits[0]`, with the final pool state being:

```
pending_deposits[0] = D + x
total_balance[0] = B - x
pool's credit balance = C
pending_withdrawal[1] = W
```

As long as the pool satisfies the initial assumptions for this scenario, the attacker can invoke the above steps repeatedly until all of the `total_balance[0]` has been moved to the `pending_deposits[0]`. Furthermore, the first two steps can be completed in a single transaction, so the attack is only limited by the total amount of credits that the attacker has.

Impact If an attacker is able to move all of the value of `total_balance[0]` to `pending_deposits[0]`, then `create_withdrawal_claim` will revert on subsequent calls. This will cause credits to be locked in the pool until (1) the pool admin sets the next validator and (2) a user invokes `bond_all` to bond the pool to a validator. In particular, if the admin never sets the next validator, then the affected credits will be locked in the pool permanently. Note that users that previously used `withdraw_public` to queue withdrawal of their funds will not be affected by this attack.

Furthermore, the attack above highlights several flaws in the pool workflows:

- As long as a user has deposited credits previously, the user can call `create_withdraw_claim` to withdraw staked credits even if their deposited credits have never been bonded before.
- There is no way to retrieve credits deposited through `deposit_public` outside of using `withdraw_public` or `create_withdraw_claim`, both of which can only be used under limited circumstances.
- After a full unbonding, the `total_balance[0]` does not separate credits that are to be treated as pending deposits for the next bond and credits that are meant to be withdrawable by the user. In the current implementation, attackers can force all other users' credits to be reclassified as pending deposits.

Recommendation Consider adding functionality to allow users to convert pending deposits into withdraw claims. This would prevent funds from being locked up by the attack described above.

Developer Response The developers changed `create_withdraw_claim` to first deduct the credit amount from the `pending_deposits[0]`, and then deduct any remaining requested credits from the `total_balance[0]`. This prevents the attack since the pending deposits would be considered as withdrawable.

4.1.4 V-DEM-VUL-004: Next validator can be set to the pool itself

Severity	Warning	**Commit**	10c3b6a
Type	Data Validation	**Status**	Fixed
File(s)	`arc_0038.aleo`		
Location(s)	set_next_validator		
Confirmed Fix At	b86e5c4		

The `set_next_validator` transition can be called to change the validator address/node that the arc0038 program (pool) is delegating its stake to. However, the `set_next_validator` transition does not prevent the next validator from being set to the pool's address.

```
1  // Update the validator address, to be applied automatically on the next bond_all
       call
2  function set_next_validator:
3      input r0 as address.public;
4      assert.eq self.caller
       aleo1kf3dgrz9lqyklz8kqfy0hpxxyt78qfuzshuhccl02a5x43x6nqpsaapqru;
5      async set_next_validator r0 into r1;
6      output r1 as arc_0038.aleo/set_next_validator.future;
7
8  finalize set_next_validator:
9      input r0 as address.public;
10     set r0 into validator[1u8];
```

Snippet 4.5: Definition of `set_next_validator`

Impact The pool will pass the validator as an argument to `credits.aleo/bond_public` and `credits.aleo/unbond_public` when users invoke functionality like bonding and withdrawing on the pool. If the caller of `credits.aleo/bond_public` or `credits.aleo/unbond_public` is equal to the validator argument, then the caller will be treated as a validator rather than as a delegator. This would be contrary to the intended behavior of the pool, which is to act as a delegator.

Recommendation Add an assertion to `set_next_validator` and `initialize` that prevents the next validator and the initial validator, respectively, from being equal to the pool's address.

Developer Response The developers applied the recommendation.

4.1.5 V-DEM-VUL-005: Some situations cause subtraction overflow in bond_all

Severity	Warning	**Commit**	10c3b6a
Type	Denial of Service	**Status**	Fixed
File(s)	`arc_0038.aleo`		
Location(s)	bond_all		
Confirmed Fix At	1b21d3a		

When the pool is switching to a new validator, a user can complete the switch by calling the `bond_all` transition to cause the pool to become a delegator for the new validator. The `bond_all` transitions works by invoking `credits.aleo/bond_public` with a user-specified amount of credits from the pool's balance, and then updating internal bookkeeping in the pool's state afterwards. In particular, `bond_all` will update `pending_deposits[0]` using the following calculation (pseudocode):

```
pending_deposits[0] =
  (pending_deposits[0] + total_balance[0])
  - credits.aleo/bonded[CORE_PROTOCOL].microcredits
```

where `pending_deposits[0]` and `total_balance[0]` come from the state before the call to `bond_all` and the `credits.aleo/bonded` come from the state after the call to `bond_public`.

```
cast aleo1q6qstg8q8shwqf5m6q5fcenuwsdqsvp4hhsgfnx5chzjm3secyzqt9mxm8 0u64 into r5 as
    bond_state;
get.or_use credits.aleo/bonded[
    aleo1j0zju7f0fpgv98gulyywtkxk6jca99l6425uqhnd5kccu4jc2grstjx0mt] r5 into r6;
get total_balance[0u8] into r7;
get pending_deposits[0u8] into r8;
add r8 r7 into r9;
sub r9 r6.microcredits into r10;
set r10 into pending_deposits[0u8];
set r6.microcredits into total_balance[0u8];
```

Snippet 4.6: Corresponding Aleo instructions in `bond_all` that compute the new value of `pending_deposits[0]`

One important point to note about the `bond_all` transition is that the user-specified credit amount is not provided as an argument to `bond_all`'s finalize function. This means that under specific conditions, it may be possible for the user to cause a subtraction integer overflow in the above calculation and thereby cause a revert.

Analysis of the Overflow Conditions We now identify the specific conditions required to trigger the subtraction overflow. Suppose that a next validator is set in the pool. Furthermore:

- Let `D` be the initial value of `pending_deposits[0]` .
- Let `C` be the pool's initial credit balance.
- Let `B` be the initial value of `total_balance[0]` .
- Let `A` amount bonded in `credits.aleo`. This can be nonzero if the current validator and the same validator are equal; if they are not the same, then `bond_all` will revert.

Now, suppose a user calls `bond_all` with `x` credits such that `10000 <= x <= C` (i.e., that the credit amount is above the minimum threshold of 10000 credits required for delegators and below the

total credit balance of the pool). Furthermore, as asserted by `bond_all`, `x` must be above `C - W` (i.e., the total credit balance minus the amount reserved for withdraw claims). Assuming such a call is successful, the final state of the pool will be:

```
pool's credit balance = C - x
amount bonded in credits.aleo = A + x
total_balance[0] = A + x
pending_deposits[0] = D + B - (A + x)
```

From the above, observe that subtraction overflow can occur if `D + B > A + x`. More concretely, `bond_all` will revert if the user specifies an amount of credits that is greater than

```
(pending_deposits[0] + total_balance[0]) - credits.aleo/bonded[CORE_PROTOCOL].
    microcredits
```

where all values are from the state of the pool before the call to `bond_all`.

Impact In general, the subtraction overflow can only occur when the caller of `bond_all` explicitly picks a credit amount that exceeds the amount of credits tracked by `pending_deposits[0]` plus `total_balance[0]`. If the caller is only bonding an amount of credits tracked by the bookkeeping of the pool, then the subtraction overflow is unlikely to trigger. However, if the caller decides to use the simpler formula "pool's credit balance minus amount of credits reserved for withdrawal" to calculate the amount to bond, then the overflow becomes possible.

We further note that when the preconditions of `bond_all` are satisfied, `pending_deposits[0]` represents the amount of credits deposited by users but not bonded yet, and `total_balance[0]` represents the amount of fully unbonded credits that have not been requested to be withdrawn yet. Thus, the subtraction overflow is more likely to occur when any of the following conditions hold:

- Some arbitrary user "gifts" credits to the pool using one of the `credits.aleo` transfer functions, increasing the pool's credit balance above `pending_deposits[0] + total_balance[0]`.
- Prior to a full unbonding of the pool, multiple users have called `withdraw_public` to decrease the `total_balance[0]`, but they have not completed their withdrawal claims yet.
- The next validator is the same as the current validator, so that there may be a nonzero amount of credits already bonded.

Recommendation It is not clear whether it is even intended for users to be able to bond more credits than is tracked by the pool's `pending_deposits[0] + total_balance[0]`. If not intended, the developers should add an additional assertion to enforce this restriction.

A simple change to avoid the overflow is to change the calculation of `pending_deposits[0]` to implement the following pseudocode:

```
pending_deposits[0] =
  credits.aleo/accounts[CORE_PROTOCOL].microcredits - pending_withdrawal[1]
```

The reasoning behind why this is correct:

- The credits corresponding to the `amount` specified by the user come from three sources: `pending_deposits[0]`, `total_balance[0]` left over from the previous unbonding period, and any untracked credit amounts in the pool's credit balance.
- Right before `bond_all` is called, `total_balance[0]` represents the credits unbonded in the previous cycle that have not been requested for withdrawal. Thus, the final value of `total_balance[0]` set after the call to `bond_all` will correctly be the final amount of bonded credits.
- After the call to `credits.aleo/bond_public`'s finalize function, the remaining credits in the pool must therefore consist of credits requested for withdrawal, unspent pending deposits, and other untracked credits. The excess untracked credits are automatically absorbed into the pending deposits.
- The proposed subtraction operation should not overflow, since an important invariant of the arc0038 program is that the pool should always have enough liquid credits to cover withdrawal requests.

Developer Response The developers applied the suggested change to the calculation of `pending_deposits[0]`.

4.1.6 V-DEM-VUL-006: Unnecessary terms included in calculation

Severity	Info	**Commit**	10c3b6a
Type	Maintainability	**Status**	Fixed
File(s)	`arc_0038.aleo`		
Location(s)	See description		
Confirmed Fix At	ba660cb		

The following sequence of instructions is used several times throughout several finalize functions in `arc_0038.aleo`:

```
    add r0 r1 into r4;
    mul r3 1_000u128 into r5;
    add r4 r2 into r6;
    mul r5 r6 into r7;
    mul r4 1_000u128 into r8;
    div r7 r8 into r9;
```

Snippet 4.7: Snippet from `calculate_new_shares_test()` in `arc_0038.aleo`

This sequence of instructions corresponds to the following calculation from the reference code in `main.leo`.

```
let new_total_shares: u128 = (shares * PRECISION_UNSIGNED) * (full_balance + deposit)
    / (full_balance * PRECISION_UNSIGNED);
```

Snippet 4.8: Snippet from `calculate_new_shares()` in `main.leo`

The term `PRECISION_UNSIGNED` is used in both the dividend and the divisor. It does not affect the precision of the calculation as all multiplications and additions occur before the division operation. Thus, the extra `PRECISION_UNSIGNED` terms have no impact on the final result of the equation and can be removed.

The finalize functions that use the pattern described above include:

- `calculate_new_shares_test`
- `set_commission_percent`
- `unbond_all`
- `claim_commission`
- `deposit_public`
- `withdraw_public`

Impact The redundant multiplications complicate the calculation, which may cause confusion for future developers. They also slightly increases the risk of a multiplication integer overflow happening in the calculation.

Recommendation Remove the `PRECISION_UNSIGNED` term from the numerator and the denominator.

Developer Response The developers applied the recommendation.

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

Aleo Credit A token paid by users and awarded to participants of the Aleo blockchain. More information can be found at https://aleo.org/aleo-credits/. . 1

liquidity pool Crowdsourced pools of digital assets used to facilitate trades between assets.. 1