Veridise

Auditing Report

Hardening Blockchain Security with Formal Methods

FOR

Verulend

Veridise Inc.

Aug. 19, 2025

► **Prepared For:**

Venture23
https://www.venture23.io/

► **Prepared By:**

Alberto Gonzalez
Mark Anthony

► **Contact Us:**

contact@veridise.com

► **Version History:**

Nov. 17, 2025 V2
Sep. 10, 2025 V1

Contents

1 Executive Summary

From Aug. 19, 2025 to Sep. 2, 2025, Venture23 engaged Veridise to conduct a security assessment of their Verulend protocol. The engagement focused on reviewing the Leo smart contracts associated with the Verulend project. Veridise conducted the assessment over 22 person-days, with 2 security analysts reviewing the project over 11 days on commit `04a6133`. The review strategy involved a thorough, manual review of the program source code performed by Veridise security analysts.

Project Summary. The security assessment covered Verulend, an overcollateralized lending protocol implemented on Aleo. It enables users to supply assets as collateral, borrow against them, and earn yield from the interest generated on deposits. The protocol enforces over-collateralization to ensure solvency, with each user's borrowing power determined by their collateral holdings that are then validated against the configured loan-to-value (LTV) ratios and liquidation thresholds.

A central design element of the protocol is its system of **cumulative interest rate indexes**, which track both the lending yields and borrowing costs. Each time a user interacts with the protocol, their balances are updated against these indexes, ensuring that accrued interest is accurately reflected. Asset prices used to value collateral balances and debt positions are sourced via the Aleo Oracle, which maintains asset prices verified by the oracle notarization backend.

The Verulend protocol is implemented as a collection of modular Leo programs:

- **Admin program.** The admin program acts as a custodian of all the pooled funds. Its operations are governed by the council, ensuring that only privileged programs or governance actions can transfer out protocol assets.
- **Oracle program.** Integrates with the Aleo Oracle to source and normalize price data to the protocol supported token decimals. It also computes the live price of Paleo tokens each time the price of Aleo credits is updated.
- **Treasury program.** Collects protocol fees, such as a share of interest payments and borrow fees, and holds them separately from user funds. Treasury assets can be redistributed by the council.
- **Reserve State program.** The reserve state program manages the state of all asset reserves within the protocol and tracks the protocol's financial state, including cumulative interest indexes and users position balances.
- **User State Program.** The user state program tracks each user's deposits, borrowings, and accrued interest. It manages state updates to user positions and ensures that user balances are consistently updated against the latest reserve indexes.

Governance and administrative actions are executed through the **council**, a privileged body responsible for protocol configuration, treasury management, and upgrades. Proposals can only be enacted once a majority of council members have approved them. All Verulend programs follow the Aleo upgradability pattern, enabling governance-controlled upgrades while preserving the program state.

Code Assessment. The Verulend developers provided the source code of the protocol contracts for review. The codebase appears to be largely original. Documentation was provided in the form of READMEs, along with supplementary material on interest rate simulations. To aid the review process, the Verulend developers also met with the Veridise security analysts to provide a walkthrough of the codebase and answer questions about its design.

The source code included a test suite; however, the Veridise security analysts noted that it could not be executed due to recent modifications.

Summary of Issues Detected. The security assessment uncovered 24 issues, 5 of which are assessed to be of a critical or high severity by the Veridise analysts. Specifically, V-VER-VUL-001 highlights how a flawed uniqueness check allows duplicate assets to be processed multiple times when computing balances, enabling users to overstate collateral and manipulate health factors, V-VER-VUL-002 shows how missing validation between the reserve asset and the provided token record during liquidation lets liquidators use mismatched assets, enabling them to liquidate high value debt while transferring cheaper tokens, V-VER-VUL-003 demonstrates how misuse of the `overflow_subtract()` function can be exploited by users to steal funds, V-VER-VUL-004 highlights how inconsistent use of global indexes in interest calculations leads to outdated values and double counting, and V-VER-VUL-005 shows how incorrect index resets can let attackers restore balances with stale indexes, enabling them to steal unearned interest.

The Veridise analysts also identified 4 medium-severity issues, including V-VER-VUL-006 where the user key to reference user balances is incorrectly derived from the caller instead of the signer, V-VER-VUL-007 that shows how failing to handle cases where collateral and reserve assets are the same can cause double accumulation of interest and overwriting of user indexes, V-VER-VUL-008 which explains why the withdrawal of frozen assets should not affect the accounting of collateral balance, as well as 9 low-severity issues, 5 warnings, and 1 informational finding.

Recommendations. After conducting the assessment of the protocol, the security analysts had a few suggestions to improve the security of the Verulend protocol before deployment.

Improve Testing and Documentation. During the audit, the Veridise team was unable to run the test suite. To ensure reproducibility and accessibility, the project should maintain a clear and detailed guide describing the steps required to set up the environment and run the tests after cloning the repository. This guide should also include instructions for adding new tests. The ability to write tests is important for security assessments of DeFi protocols, as proof-of-concept tests can help evaluate the precise impact of potential issues and eliminate false positives.

From the issues identified during the review, none were related to standard user flows; most originated from explicitly adversarial behavior. This indicates that the current test suite provides adequate coverage for expected usage patterns. Additional adversarial tests are recommended to strengthen the project's defenses against malicious inputs.

A specific gap was observed in the unit testing of interest rate calculations and cumulative interest index usage. The issue V-VER-VUL-004 highlighted the need for targeted tests that confirm correct application of interest rates and indexes throughout the codebase. The Veridise team recommends including multiple scenarios that verify expected values for user indexes, accrued interest, and related calculations.

Liquidations. The Verulend protocol implements a lending cluster pool of three assets, where any asset can be used as collateral to borrow any of the others. This structure exposes the system to correlated risk, as the stability of the entire protocol depends on the asset with the weakest collateral performance. When a position becomes undercollateralized and generates bad debt, the effect can propagate to other positions, especially in cases where collateral has been rehypothecated. To mitigate these risks, it is essential to validate that the liquidation mechanism functions reliably under adverse conditions. The Veridise team recommends conducting extensive stress testing of the liquidation engine and to consider the following measures to increase the security of the liquidation system in case of volatile scenarios:

- **Risk Parameter Configuration.** The protocol already defines borrow caps, maximum loan-to-value ratios, and liquidation thresholds. When configuring these parameters, it is important to account for the actual liquidity available for each supported token in the Aleo ecosystem. Large positions may fail to liquidate if liquidators cannot close them profitably due to high slippage or insufficient liquidity. Proper calibration of these parameters reduces the likelihood of unresolvable positions and systemic instability.
- **Same-Asset Collateral and Debt.** In Verulend, the same token used as collateral can also be borrowed. The liquidation flow must account for scenarios where both the debt and collateral involve the same token. Although inconsistencies with this case were identified and reported in issue V-VER-VUL-007, it is important to ensure such scenarios are covered in the test suite to validate correct behavior.
- **Position Manipulation Prevention.** As discussed in issue V-VER-VUL-009, make sure a safety buffer exists between the position a user can open and the liquidation threshold. Without such a buffer, users can create positions that are immediately close to liquidation, which increases pressure on the liquidation system during volatile conditions. This can overload liquidator infrastructure, including automated bots, and raise the likelihood of failed or delayed liquidations.
- **Liquidation Amount Optimization.** Increase the portion of a user's position that can be liquidated in a single transaction, as discussed in issue V-VER-VUL-016. The purpose of liquidation is to restore the protocol's health by improving the affected position's health factor. If multiple transactions are required to liquidate a single account, the probability of subsequent liquidations failing increases, which can leave the protocol exposed to bad debt.

Post-Deployment Review. During this engagement, the Veridise team identified several issues across various components of the protocol. The subsequent fix-review process was conducted incrementally and often in isolation, with different components being reviewed at different points in time. In addition to addressing previously reported findings, the fix-review phase included architecture changes and modifications that originated from a separate review process.

Moreover, during the fix review, new issues were identified that stemmed from recent code modifications. This highlights the potential for additional vulnerabilities to arise as a consequence of ongoing development and code iteration. Taken together, these observations indicate that a further code review will be necessary to ensure the robustness of the final deployed version.

For these reasons, the Veridise analysts recommend that the development team conduct an additional security review after the protocol has been deployed and has accumulated some operational history. This review should focus on the live system's state.

Until such a review is completed, we also recommend limiting the total value locked (e.g., via whitelisting or controlled access) during the initial launch while the amount of funds remains small. This will help reduce potential exposure in the event that undiscovered issues persist and will provide an additional safeguard for user funds as the system's post-deployment security posture is reassessed.

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
Verulend	`04a6133`	Leo	Aleo

Table 2.2: Engagement Summary.

Dates	Method	Consultants Engaged	Level of Effort
Aug. 19–Sep. 2, 2025	Manual	2	22 person-days

Table 2.3: Vulnerability Summary.

Name	Number	Acknowledged	Fixed
Critical-Severity Issues	3	3	3
High-Severity Issues	2	2	2
Medium-Severity Issues	4	4	3
Low-Severity Issues	9	9	7
Warning-Severity Issues	5	5	3
Informational-Severity Issues	1	1	1
TOTAL	24	24	19

Table 2.4: Category Breakdown.

Name	Number
Logic Error	9
Data Validation	7
Usability Issue	4
Maintainability	4

3 Security Assessment Goals and Scope

3.1 Security Assessment Goals

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

- Is access control implemented correctly across the protocol?
- Are the cumulative interest rate calculations implemented correctly?
- Is the interest gain applied in each workflow that computes the total collateral and total debt of a user?
- Is the treasury fee calculation implemented correctly? And are the fees handled properly?
- Is the health of user positions evaluated correctly before borrows, withdrawals, and liquidations?
- Does the protocol implement freshness guarantees for fetched oracle prices?
- Is the protocol susceptible to any common vulnerabilities specific to Leo programs?
- Does the liquidation model protect the protocol from ending up with bad debt?
- Does the protocol correctly account for tokens with different decimals and prices while maintaining sufficient precision?
- Does the oracle validate manage different token prices correctly, and does it perform the necessary validations on the attested data?
- Can borrows or withdrawals leave user positions in an under-collateralized state?
- Can user funds get locked within the protocol?
- Are frozen assets accounted for properly?
- Is market configuration handled correctly? And does the protocol make proper provisions to handle removal of market support?
- Has the protocol set up Aleo program upgradability in a secure and correct manner?
- Can an attacker exploit any of the deposit, borrow, redeem, repay, or liquidation workflows to steal funds from the protocol?
- Can an attacker cause denial of service for other users, or for protocol administrators?

3.2 Security Assessment Methodology & Scope

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

Scope. The scope of this security assessment is limited to the following files within the `(aleo/programs)` folder of the source code provided by the Verulend developers, which contains the smart contract implementation of Verulend.

- `amm_admin_v006.leo`
- `amm_council_impl_v001.leo`
- `amm_instant_cr_v004.leo`

- `amm_interface_liquidation_v002.leo`
- `amm_interface_v007.leo`
- `amm_oracle_interface_v002.leo`
- `amm_oracle_v004.leo`
- `amm_program_checksums.leo`
- `amm_reserve_state_v004.leo`
- `amm_treasury_v002.leo`
- `amm_user_state_v004.leo`

Methodology. Veridise security analysts read the Verulend documentation and then began a manual review of the code.

During the security assessment, the Veridise security analysts regularly met with the Verulend developers to ask questions about the code and to update the developers about the progress of the review.

3.3 Classification of Vulnerabilities

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

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

Table 3.1: Severity Breakdown.

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

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

Table 3.2: Likelihood Breakdown

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

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

Table 3.3: Impact Breakdown

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

4 Trust Model

4.1 Operational Assumptions

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

- **Council Members.** Council members are trusted not to collude, act maliciously, or otherwise act against the interests of the protocol and its users.
- **Program Ownership.** The council implementation is intended to serve as the *owner* of all contracts within the protocol. Analysts assume that ownership of the individual programs is correctly assigned to the council as intended.
- **Aleo Oracle Prices.** The protocol relies on the Aleo oracle to provide accurate asset prices. Analysts assume that the oracle-reported values are correct and not derived from compromised feeds, but make no assumptions regarding the freshness of the data.
- **Block Height Approximation.** The protocol computes time differences using a fixed average block height instead of on-chain timestamps. Analysts assume that the chosen average block height is a sufficiently accurate approximation of the actual network average.
- **Asset Configuration Parameters.** The asset configuration parameters (e.g., borrow caps, maximum loan-to-value ratios, liquidation thresholds) are assumed to have been carefully set, reflecting realistic liquidity conditions in the ecosystem.
- **Third-Party Integrations.** It is assumed that third-party programs will not intent to directly integrate with the Money Market beyond serving as call aggregators. Since all fund keys are derived using `self.signer` rather than `self.caller`, external programs cannot independently open or manage positions within the protocol.
- **Off-Chain Liquidation Infrastructure.** The protocol relies on the presence of off-chain infrastructure responsible for monitoring unhealthy positions and initiating liquidations as needed. This infrastructure is also assumed to periodically clean up locked liquidation mappings to ensure that the protocol's internal state remains consistent over time.

4.2 Privileged Roles

Roles. This section describes in detail the specific roles present in the system, and the actions each role is trusted to perform. During the review, Veridise analysts assumed that these role operators perform their responsibilities as intended. Protocol exploits relying on the below roles acting outside of their privileged scope are considered outside of scope.

- The program `amm_council_impl_v001.leo` is intended to serve as the owner of all contracts within the protocol. It is governed by `amm_council_v001.leo`, where members vote on the proposed actions. A proposal passes only if it meets the required threshold of votes, after which it can be executed. In effect, this program functions as the administrator for the following programs:
  - `amm_admin_v006.leo`

  - `amm_instant_cr_v004.leo`
  - `amm_interface_liquidation_v002.leo`
  - `amm_interface_v007.leo`
  - `amm_oracle_v004.leo`
  - `amm_program_checksums.leo`
  - `amm_reserve_state_v004.leo`
  - `amm_treasury_v002.leo`
  - `amm_user_state_v004.leo`
- The following privileged actions can be performed through `amm_council_impl_v001.leo`.
  - Pause or unpause the admin program, which also halts or resumes the outflow of funds from the protocol.
  - Update the allowlist of programs maintained by each of the programs listed above. The allowlist defines which callers are permitted to call the functionality defined by the programs.
  - Upgrade any of the programs listed above.
  - Update the set of allowed tokens.
  - Transfer ownership of the programs.
  - Redistribute treasury funds.
  - Associate oracle request hashes with their corresponding tokens. Essentially configuring the allowed price feeds for different assets.
  - Withdraw tokens from the AMM pool.
  - Add, remove, or update reserve configurations for tokens.

Operational Recommendations. All privileged operations within the protocol are governed by the AMM council, making it essential that members act promptly when required. Procedures should be set up to actively monitor proposal activity and voting participation, ensuring that quorum thresholds are met in a timely manner for critical administrative actions. Time-sensitive operations (such as pausing the protocol or updating critical parameters) should be thoroughly tested to demonstrate the council's ability to respond without undue delay.

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

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

- Regularly drilling responses to situations requiring emergency response such as pausing/unpausing.

5 Vulnerability Report

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

Table 5.1: Summary of Discovered Vulnerabilities.

ID	Description	Severity	Status
V-VER-VUL-001	User data updates can process duplicate . . .	Critical	Fixed
V-VER-VUL-002	Reserve asset in liquidation not checked . . .	Critical	Fixed
V-VER-VUL-003	Users can steal funds due to misuse of the . . .	Critical	Fixed
V-VER-VUL-004	Inconsistent Cumulative Index Updates . . .	High	Fixed
V-VER-VUL-005	Reset of cumulative interest indexes allows . . .	High	Fixed
V-VER-VUL-006	User key derived from caller instead of . . .	Medium	Fixed
V-VER-VUL-007	Identical tokens in liquidation causes . . .	Medium	Fixed
V-VER-VUL-008	Frozen token withdrawals should not . . .	Medium	Fixed
V-VER-VUL-009	Withdrawal checks use liquidation . . .	Medium	Acknowledged
V-VER-VUL-010	Initializer can be called multiple times to . . .	Low	Fixed
V-VER-VUL-011	Council program cannot withdraw . . .	Low	Fixed
V-VER-VUL-012	Lack of freshness guarantees in oracle price . . .	Low	Fixed
V-VER-VUL-013	Multiple request hashes can correspond to . . .	Low	Fixed
V-VER-VUL-014	Hardcoded block time may diverge from . . .	Low	Fixed
V-VER-VUL-015	Treasury fees not deducted from available . . .	Low	Fixed
V-VER-VUL-016	Small liquidation cap creates inefficient . . .	Low	Acknowledged
V-VER-VUL-017	Global debt and collateral temporarily . . .	Low	Acknowledged
V-VER-VUL-018	Utilization rate calculated with outdated . . .	Low	Fixed
V-VER-VUL-019	Missing mechanism to disable deprecated . . .	Warning	Fixed
V-VER-VUL-020	Missing validation of token decimals when . . .	Warning	Fixed
V-VER-VUL-021	Minor Inconsistencies	Warning	Fixed
V-VER-VUL-022	Division by zero in exa_div() returns 0 . . .	Warning	Acknowledged
V-VER-VUL-023	Price updates rely only on whitelisted . . .	Warning	Acknowledged
V-VER-VUL-024	Unused Code	Info	Fixed

5.1 Detailed Description of Issues

5.1.1 V-VER-VUL-001: User data updates can process duplicate assets

Severity	Critical	**Commit**	04a6133
Type	Data Validation	**Status**	Fixed
Location(s)	`aleo/programs/amm_user_state_v004.leo:124-168, 407-440`		
Confirmed Fix At	4e107ab		

Description The `update_user_data` transition relies on the `asset_token_id` array to calculate a user's total deposits, collateral, and borrow balances. The expectation is that this array contains unique assets, but the validation mechanism is flawed. The `get_unique_count()` function determines uniqueness by only considering the last occurrence of any duplicate asset. As a result, the function can return the expected `unique_count` even if duplicates exist earlier in the array. See snippet below for reference.

```
1    function get_unique_count(token_ids:[field;10]) -> u8 {
2        let unique_count:u8 = 0u8;
3        if(token_ids[0] !=  ZERO_TOKEN_ID) {
4            unique_count=token_ids[0]!= token_ids[1] && token_ids[0]!=  token_ids[2]
    && token_ids[0]!= token_ids[3] && token_ids[0]!= token_ids[4] && token_ids[0]!=
    token_ids[5] && token_ids[0]!= token_ids[6] && token_ids[0]!= token_ids[7] &&
    token_ids[0]!= token_ids[8] && token_ids[0]!= token_ids[9] ? unique_count+ 1u8 :
    unique_count+0u8;
5        }
6        if(token_ids[1] !=  ZERO_TOKEN_ID) {
7            unique_count=token_ids[1]!= token_ids[2] && token_ids[1]!= token_ids[3]
    && token_ids[1]!= token_ids[4] && token_ids[1]!= token_ids[5] && token_ids[1]!=
    token_ids[6] && token_ids[1]!= token_ids[7] && token_ids[1]!= token_ids[8] &&
    token_ids[1]!= token_ids[9] ? unique_count+ 1u8 : unique_count+0u8;
8        }
9        if(token_ids[2] !=  ZERO_TOKEN_ID) {
10           unique_count=token_ids[2]!= token_ids[3] && token_ids[2]!=  token_ids[4]
    && token_ids[2]!= token_ids[5] && token_ids[2]!= token_ids[6] && token_ids[2]!=
    token_ids[7] && token_ids[2]!= token_ids[8] && token_ids[2]!= token_ids[9] ?
    unique_count+ 1u8 : unique_count+0u8;
11       }
12       if(token_ids[3] !=  ZERO_TOKEN_ID) {
13           unique_count=token_ids[3]!= token_ids[4] && token_ids[3]!= token_ids[5]
    && token_ids[3]!= token_ids[6] && token_ids[3]!= token_ids[7] && token_ids[3]!=
    token_ids[8] && token_ids[3]!= token_ids[9] ? unique_count+ 1u8 : unique_count+0
    u8;
14       }
15       if(token_ids[4] !=  ZERO_TOKEN_ID) {
16           unique_count=token_ids[4]!= token_ids[5] && token_ids[4]!= token_ids[6]
    && token_ids[4]!= token_ids[7] && token_ids[4]!= token_ids[8] && token_ids[4]!=
    token_ids[9] ? unique_count+ 1u8 : unique_count+0u8;
17       }
18       if(token_ids[5] !=  ZERO_TOKEN_ID) {
```

```
            unique_count=token_ids[5]!= token_ids[6] && token_ids[5]!= token_ids[7]
    && token_ids[5]!= token_ids[8] && token_ids[5]!= token_ids[9] ? unique_count+ 1u8
     : unique_count+0u8;
        }
        if(token_ids[6] !=  ZERO_TOKEN_ID) {
            unique_count=token_ids[6]!= token_ids[7] && token_ids[6]!= token_ids[8]
    && token_ids[6]!= token_ids[9] ? unique_count+ 1u8 : unique_count+0u8;
        }
        if(token_ids[7] !=  ZERO_TOKEN_ID) {
            unique_count=token_ids[7]!= token_ids[8] && token_ids[7]!= token_ids[9] ?
     unique_count+ 1u8 : unique_count+0u8;
        }
        if(token_ids[8] !=  ZERO_TOKEN_ID) {
            unique_count=token_ids[8]!= token_ids[9] ? unique_count+ 1u8 :
    unique_count+0u8;
        }
        if(token_ids[9] !=  ZERO_TOKEN_ID) {
            unique_count += 1u8;
        }
        return unique_count;
    }
```

This becomes critical because the loop that processes balances iterates over the entire `[0..10]` range of the provided asset array. Consequently, repeated assets can be processed multiple times, allowing a user to inflate deposits or manipulate their health factor for withdrawals, borrows and the like. For example, by duplicating a deposited asset, a user can artificially boost their collateral value and health factor, thereby bypassing liquidation risk.

Impact Users can inflate their collateral balances by crafting duplicate asset arrays, leading to incorrect health factor calculations and under-collateralized positions which can cause protocol insolvency.

Recommendation Within `get_unique_count()`, only count unique occurrences of assets and enforce that there are no duplicates within the asset array.

For reference, the amm council implements a `get_valid_unique_address_count()` function which correctly enforces this when counting unique addresses.

Developer Response The developers now enforce that there are no duplicates when counting assets with `get_unique_count()`.

5.1.2 V-VER-VUL-002: Reserve asset in liquidation not checked against transferred token

Severity	Critical	**Commit**	04a6133
Type	Data Validation	**Status**	Fixed
Location(s)	`aleo/programs/amm_interface_liquidation_v002.leo:60-198`		
Confirmed Fix At	9c6e903		

Description The `liquidation()` transition accepts both a `reserve` field and an `input_record`, but it never verifies that these two values refer to the same asset. The reserve is meant to represent the asset whose debt is being repaid, while the input record contains the token actually transferred to the pool. Because the function proceeds with liquidation checks using the `reserve`, yet executes the transfer based on the `input_record`, a malicious liquidator can deliberately pass mismatched values.

This mismatch allows the attacker to pass a valid reserve for the purpose of validating the liquidation, while simultaneously providing an unrelated input record for the transfer. An attacker can exploit this mismatch to steal funds by liquidating a large debt position denominated in a high-value token while only transferring tokens from a market with a significantly lower value.

Impact A liquidator can use a mismatched reserve and input record to drain funds from the protocol.

Recommendation Remove the separate `reserve` input, and instead derive the repaid asset directly from the `input_record`.

Developer Response The developers now derive the token id of the repaid asset directly from the input record, as recommended.

5.1.3 V-VER-VUL-003: Users can steal funds due to misuse of the overflow_subtract() function

Severity	Critical	**Commit**	04a6133
Type	Logic Error	**Status**	Fixed
Location(s)	`aleo/programs/amm_reserve_state_v004.leo:299, 491`		
Confirmed Fix At	66fc36d		

Description The protocol includes functions `finalize_update_user_amt_borrow()` for handling borrow/repay operations and `finalize_update_data_redeem()` for withdrawal operations. Both functions support an `all` parameter that allows users to repay their full debt or withdraw their complete balance.

The protocol uses a custom `overflow_subtract()` function that returns 0 when the first operand is smaller than the second, rather than reverting. This causes an issue in two of the aforementioned workflows:

In repayments, when `all` is true and `ramount < current_borrowed_amount`, the calculation becomes:

```
1 pending_repay_balance = overflow_subtract(ramount, current_borrowed_amount) = 0;
2 ramount = current_borrowed_amount;
```

The above has the consequence that the user will transfer to the protocol an amount of funds smaller than `current_borrowed_amount` but its debt will be reset to zero, essentially draining the protocol funds.

In withdrawal operations (line 479), when `all` is true and `balance < r_amount` the calculation becomes:

```
1 pending_balance = overflow_subtract(balance, r_amount) = 0;
2 r_amount = balance;
```

In the above snippet, `r_amount` represents the amount to withdraw. In this way, it can be observed that users can withdraw more than their full balance.

Impact Users can exploit this vulnerability to steal protocol funds through two attack vectors. In repayment operations, attackers can fully clear their debt by repaying any amount less than owed while setting `all=true`. In withdrawal operations, attackers can extract more than their maximum available balance by requesting withdrawal amounts greater than their balance while setting `all=true`.

Recommendation For repayment flows, verify that `ramount >= current_borrowed_amount` when `all=true`, and for withdrawal flows, verify that `r_amount <= balance` when `all=true`. Implement clear conditional logic with appropriate error handling rather than relying on the `overflow_subtract` function.

Developer Response The developers now verify that `ramount >= current_borrowed_amount` when `all=true` in the repayment flow and `r_amount <= balance` during the withdrawal flow.

5.1.4 V-VER-VUL-004: Inconsistent Cumulative Index Updates Lead to Double-Counting and Stale Interest Calculations

Severity	High	**Commit**	04a6133
Type	Logic Error	**Status**	Fixed
Location(s)	`aleo/programs/` ▶ `amm_reserve_state_v004.leo:224, 292` ▶ `amm_user_state_v004.leo:137, 144`		
Confirmed Fix At	b28cd85, d14a04d, f3d031c, 0fde540		

Description The Verulend protocol tracks interest through global cumulative indexes for supply and borrow that accumulate over time (`liquidity_cumulative_index` and `borrow_cumulative_index`), plus per-user indexes recorded at deposit/borrow time. The `instant_CR()` function calculates accumulated interest rates (linear interest for deposits, compounded interest for borrows) and `update_CI()` updates the global indexes. Then, up to date user balances can computed as `(amount * global_index) / user_index` to account for interest accrued since the previous update. However, as presented in the following instances, the code sometimes uses outdated global index values or double-counts accrued interest:

- In both `finalize_update_user_amt_borrow` and `finalize_update_user_liq` from the `amm_reserve_state_v004` program, the code computes normalized_debt as `exa_mul(interest_for_debt, reserve.borrow_cumulative_index)`. However, `reserve.borrow_cumulative_index` was already updated via `update_CI` previously before calling this function, resulting in double-counting of accrued interest.
- In `finalize_update_user_data` from the `amm_user_state_v004` program:
  - The code uses outdated `reserve.liquidity_cumulative_index` values for tokens in the `asset_token_id` array that are not the current `token_id`.
  - The code unnecessarily computes normalized_debt (as in point 1) for the `token_id`, even though `update_CI` has already been called for this token, again resulting in double-counting of borrow interest.
- It is important to consider that point (2) becomes more complex when `finalize_update_user_data` is invoked by the liquidation transition from the `amm_interface_liquidation_v002` program, where two tokens from the `asset_token_id` array have `update_CI` called instead of just one (the reserve and the collateral), potentially affecting how the fix is introduced for `finalize_update_user_data`. This is of particular importance when considering that both the tokens may be in the same market, in which case calling `update_CI()` twice on the `collateral` and `reserve` will result in counting the interest twice.

Impact Using non-updated or double counted indexes will lead to a miscalculation of collateral and debt for the users, potentially causing liquidation decisions based on incorrect data.

Recommendation (1) Remove the redundant `normalized_debt` calculations from `finalize_update_user_amt_borrow` and `finalize_update_user_liq` since `update_CI` has already updated the `reserve.borrow_cumulative_index` index,

(2) Modify `finalize_update_user_data` to differentiate between the borrowed or withdrawn token for which `update_CI` has been called and the other tokens in the `asset_token_id` array which have not.

(3) Consider that in the `liquidation` flow, two different tokens from the `asset_token_id` will have `update_CI` called (the reserve and the collateral) when dealing with the (2) fix.

Developer Response Point 1 was addressed by removing the mentioned redundant calculations.

Point 2 was addressed by adding a conditional which only simulates cumulative index increase for tokens (normalized deposit, and debt) other than the asset involved in the borrow/withdrawal flow.

Point 3 was addressed by calling `update_CI(collateral)` after `update_user_data()`.

5.1.5 V-VER-VUL-005: Reset of cumulative interest indexes allows free interest extraction

Severity	High	**Commit**	04a6133
Type	Logic Error	**Status**	Fixed
Location(s)	`aleo/[...]/amm_reserve_state_v004.leo:301, 375-376, 495`		
Confirmed Fix At	b772bb6		

Description The protocol implements a two-step liquidation process where the liquidated user debt and collateral that were not part of the final liquidation are temporarily locked in the `liquidatee_locked_collateral` and `liquidatee_locked_borrow` mappings before being transferred back via `claim_liquidation()`. However, the `finalize_claim_liquidation()` function in `amm_reserve_state_v004.leo` fails to update the user's cumulative indexes when restoring locked balances to the main `deposited_amount` and `borrowed_amount` mappings.

On the other side, the protocol tracks interest accrual through cumulative indexes that compound over time. When users fully repay debt or withdraw all collateral, the `finalize_update_user_amt_borrow()` and `finalize_update_data_redeem()` functions incorrectly reset these indexes to `EXA` (the base value). This reset logic, combined with the `finalize_claim_liquidation()` function that restores balances without updating cumulative indexes, creates an exploitable vulnerability.

An attacker can exploit this by: (1) Creating a position near liquidation threshold, (2) Self-liquidating to lock balances in the `liquidatee` mappings, (3) Fully repaying remaining debt and withdrawing the remaining collateral to reset cumulative indexes to `EXA`, (4) Calling `claim_liquidation()` to restore locked balances with stale indexes, causing the protocol to calculate interest from `EXA` to the current cumulative index values.

Impact Attackers can extract unearned interest from the protocol by manipulating the timing of liquidation completion and exploiting the cumulative index resets.

Recommendation Remove the cumulative index reset logic from the `finalize_update_user_amt_borrow()` and `finalize_update_data_redeem()` functions. Instead of resetting indexes to `EXA`, whenever debt is fully repaid or collateral fully withdrawn, preserve the current global liquidity and borrow cumulative indexes.

Developer Response The cumulative indexes are not reset after complete repayment or complete withdrawal, as recommended.

5.1.6 V-VER-VUL-006: User key derived from caller instead of signer during claims

Severity	Medium	**Commit**	04a6133
Type	Logic Error	**Status**	Fixed
Location(s)	`aleo/programs/amm_interface_v007.leo:373-376, 400-403`		
Confirmed Fix At	fc067e4		

Description In the `claim_pending_repay()` and `claim_pending_withdraw()` transitions, the `user_key` is computed using `self.caller` instead of `self.signer`. This diverges from the rest of the codebase, where `self.signer` is consistently used to derive the user key. If a user interacts with these functions through an intermediary contract, the `user_key` will be tied to the contract address rather than the actual signer of the transaction. See snippet below for context.

```
1    async transition claim_pending_withdraw(user_salt: amm_user_state_v004.aleo/
     UserSalt, token_id: field, amount: u128) -> (token_registry.aleo/Token, Future) {
2        assert(amount > 0u128);
3        let user_key: field = BHP256::hash_to_field(
4            UserKey {
5                holder: self.caller,
6                salt: user_salt.salt
7            }
```

A user attempting to claim pending funds through an intermediary contract may fail to claim their funds entirely, since their signer-derived `user_key` will not match the one derived through `self.caller` within the claim transitions. This represents a deviation from expected behavior, as accounting for pending claims should always be tied to the transaction signer, and not the intermediary contract.

Impact Users who interact through intermediary contracts will be unable to claim their pending funds.

Recommendation The `user_key` computation should use `self.signer` rather than `self.caller` in both `claim_pending_repay()` and `claim_pending_withdraw()`.

Developer Response The `user_key` is now derived from `self.signer` in the mentioned locations.

5.1.7 V-VER-VUL-007: Identical tokens in liquidation causes double accounting of interest and incorrect index updates

Severity	Medium	**Commit**	04a6133
Type	Logic Error	**Status**	Fixed
Location(s)	`aleo/programs/` ▶ `amm_interface_liquidation_v002.leo:142-144` ▶ `amm_reserve_state_v004.leo:241-248`		
Confirmed Fix At	afd08a2, 8f01bbe		

Description The `liquidation` flow from the `amm_interface_liquidation_v002` program assumes that the collateral and reserve tokens are always different, but fails to handle cases where they are the same token. This creates two issues when liquidating positions where the borrowed asset and collateral are identical.

1. The liquidation flow calls `update_CI()` for both the reserve token and collateral token to update cumulative interest indexes. When these tokens are the same, `update_CI()` executes twice on the same token, causing double accumulation of liquidity and borrow interest that should only be applied once.
2. Additionally, `finalize_update_user_liq()` writes user cumulative indexes sequentially for both collateral and reserve positions. When the tokens are identical, these writes target the same user-token mapping, causing the second write to overwrite the first. The collateral index gets set to the current global liquidity index, but then immediately gets overwritten with the original user deposit index, reverting any updates:

```
user_cumulative_index.set(liquidating_collateral,UserCumulativeIndex {
     deposit: reserve_data_collateral.liquidity_cumulative_index,
     borrow: user_index_collateral.borrow
});

user_cumulative_index.set(liquidating_reserve,UserCumulativeIndex {
     deposit: user_index_reserve.deposit,
     borrow:  reserve_data_reserve.borrow_cumulative_index
});
```

Impact When liquidating positions with the same token as both collateral and debt, the protocol incorrectly calculates excessive interest accumulation and corrupts user cumulative indexes.

Recommendation Implement token identity checks in the liquidation flow to detect when collateral and reserve tokens are identical. When the same token is used, call `update_CI()` only once and merge the cumulative index updates into a single write operation that properly handles both deposit and borrow index updates for the same token mapping.

Developer Response The developers now make sure that when the same tokens are used as the collateral and reserve during liquidation, their cumulative indexes are only updated once throughout the liquidation flow.

5.1.8 V-VER-VUL-008: Frozen token withdrawals should not affect collateral balance

Severity	Medium	**Commit**	04a6133
Type	Logic Error	**Status**	Fixed
Location(s)	`aleo/programs/amm_user_state_v004.leo:196`		
Confirmed Fix At	f89dfa9		

Description In the withdrawal flow, the program subtracts the USD value of the withdrawn asset from `total_collateral_balance_USD`:

```
let requested_withdraw_USD:u128 = exa_mul(borrow_token_unit_price_exa,
    withdraw_amount_exa);
let collateral_after_withdraw_USD:u128 = total_collateral_balance_USD -
    requested_withdraw_USD;
```

The issue arises when the token being withdrawn is frozen. Frozen tokens do not contribute to `total_collateral_balance_USD`, yet their withdrawal still reduces it. This lowers the user's collateral balance even though the frozen asset was never included in the total. And it becomes more problematic if the frozen collateral is the only balance a user holds, since the program may end up locking the users frozen assets.

For example, if a user only deposits token A, and token A is later frozen, `total_collateral_balance_USD` becomes `0`. If the user then withdraws token A, the program attempts `0 - X` where `X > 0`, causing a runtime panic. Even when multiple assets are present, subtracting frozen tokens will incorrectly reduce the collateral balance and distort subsequent health factor and LTV calculations.

Impact This behaviour will misrepresent the health factor and LTV ratios by subtracting amounts that never contributed to the collateral in the first place. It can also lock user funds when they attempt to withdraw frozen tokens.

Recommendation If the token being withdrawn is frozen, do not subtract its USD value from `total_collateral_balance_USD`.

Developer Response The developers implemented a different fix where assets always contribute to the users collateral position even if they are in the frozen status.

5.1.9 V-VER-VUL-009: Withdrawal checks use liquidation threshold instead of max LTV

Severity	Medium	**Commit**	04a6133
Type	Data Validation	**Status**	Acknowledged
Location(s)	`aleo/programs/amm_interface_v007.leo:316`		
Confirmed Fix At	N/A		

Description The money market protocol implements different safety thresholds for user positions: `max_LTV` (maximum loan-to-value ratio for borrowing) and `liquidation_threshold` (threshold below which positions become liquidatable). The `max_LTV` is typically lower than the `liquidation_threshold` to provide a safety buffer and prevent users from creating positions that are immediately at risk of liquidation.

In the borrow flow within `finalize_borrow_token()` in `amm_interface_v007.leo`, the protocol correctly validates that new borrowing respects the max LTV by checking `user_data.collateral_needed_in_USD <= user_data.total_collateral_balance_USD` and `user_data.requested_borrow_USD <= user_data.available_borrows_USD`. These checks ensure users cannot borrow beyond the `max_LTV` threshold.

However, in the withdrawal flow within `finalize_withdraw_token()`, the protocol only validates against the liquidation threshold by checking `user_data.hf_withdraw_below_threshold == false`. The `hf_withdraw_below_threshold` field is calculated in `update_user_data()` using the `liquidation_threshold` rather than the `max_LTV`. This allows users to withdraw collateral until their position reaches just above the liquidation threshold, bypassing the safer max LTV constraint that applies to borrowing operations.

For example, if `max_LTV = 0.7` and `liquidation_threshold = 0.8`, a user can withdraw collateral until their LTV reaches 0.79, creating a position that is dangerously close to liquidation despite violating the intended max LTV safety margin.

Impact This inconsistency creates systematic risk by allowing users to open positions dangerously close to the liquidation threshold. In lending protocols where collateral tokens can also be borrowed, the risk compounds significantly. A single position going underwater can trigger cascading liquidations as the same assets serve as both collateral and debt across multiple positions. The domino effect from one failing position can propagate throughout the protocol, potentially creating bad debt that undermines the entire system's stability. By allowing withdrawals up to the liquidation threshold rather than the safer max LTV, the protocol increases the likelihood of such systematic failures.

Recommendation Modify the withdrawal validation logic to enforce the max LTV threshold instead of only checking the liquidation threshold.

Developer Response The developers acknowledged the issue but decided not to introduce any code changes.

5.1.10 V-VER-VUL-010: Initializer can be called multiple times to overwrite admin

Severity	Low	**Commit**	04a6133
Type	Data Validation	**Status**	Fixed
Location(s)	`aleo/programs/` ▶ `amm_interface_liquidation_v002.leo:25-32` ▶ `amm_interface_v007.leo:26-33` ▶ `amm_reserve_state_v004.leo:79-86` ▶ `amm_user_state_v004.leo:82-89`		
Confirmed Fix At	4008673, 45df077		

Description The `initialize()` transition in many programs only checks that the caller equals `INITIALIZER_ADDRESS` and then unconditionally sets the admin. There is no check to ensure that the initializer cannot be called repeatedly to overwrite the admin address. See snippet below for reference.

```
    async transition initialize(admin_address: address) -> Future {
        assert_eq(self.caller, INITIALIZER_ADDRESS);
        return(finalize_initialize(admin_address));
    }

    async function finalize_initialize(admin_address: address) {
        admin.set(true, admin_address);
    }
```

This protection exists in `amm_instant_cr_v004.leo` but is missing in several other related programs that are listed below.

- `amm_interface_v007.leo`
- `amm_interface_liquidation_v002.leo`
- `amm_reserve_state_v004.leo`
- `amm_user_state_v004.leo`

Impact The `INITIALIZER_ADDRESS` can overwrite the admin at any time with an address of its choice, undermining trust in the initialization process.

Recommendation Ensure that the `initialize()` transition can only be called once. This needs to be implemented in all the locations mentioned above.

Developer Response The developers implemented the suggested fix.

5.1.11 V-VER-VUL-011: Council program cannot withdraw received funds

Severity	Low	**Commit**	04a6133
Type	Logic Error	**Status**	Fixed
Location(s)	`aleo/programs/amm_council_impl_v001.leo`		
Confirmed Fix At	fa8eeaa		

Description Both `treasury.claim()` and `admin.withdraw()` send token registry funds to the council implementation program. However, the council program has no functionality to release or transfer these funds, leaving them locked unless the program itself is upgraded.

Impact Any tokens sent to the council program become inaccessible, potentially resulting in permanent loss of funds until an upgrade is performed.

Recommendation Modify the logic so that funds are transferred directly to the intended recipient address instead of being held by the council program.

Alternatively, add a mechanism to the council program to enable withdrawing received funds.

Developer Response On claim or withdraw tokens, the tokens are now sent to a specified recipient instead of the caller. Changes are made to support this change in the council for the proposal hash as well.

5.1.12 V-VER-VUL-012: Lack of freshness guarantees in oracle price updates

Severity	Low	**Commit**	04a6133
Type	Data Validation	**Status**	Fixed
Location(s)	`aleo/programs/amm_oracle_v004.leo`		
Confirmed Fix At	01526a9		

Description The oracle currently enforces that each new price update must have a timestamp greater than or equal to the previous one. See snippet below for context.

```
    async function finalize_set_price(from: address, request_hash: u128, token_price: u128, timestamp: u128) {
        assert_eq(allowed_programs.get(from), true);
        let token_id: field = request_hash_token_id.get(request_hash);
        let old_price_data: PriceData = price.get_or_use(token_id, PriceData { price: 0u128, timestamp: 0u128 });
        assert(old_price_data.timestamp <= timestamp);
        let decimals: u8 = token_registry.aleo/registered_tokens.get(token_id).decimals;
        let token_price_exa: u128 = convert_to_exa(token_price, decimals);

        // Veridise - elided
```

While this ensures monotonicity, it does not guarantee that the data is fresh. If the notarization backend stalls or goes offline, the timestamp of the last update remains valid indefinitely, and consumers have no way to determine that the data is stale.

Since Aleo does not expose a block timestamp on-chain, consumers of the oracle price cannot use it to implement freshness checks on-chain. As a result, they remain vulnerable in situations where the oracle backend experiences a denial-of-service or stops pushing new data, potentially causing users to unknowingly rely on outdated prices for critical functions such as collateral valuation, liquidations, and borrow limit calculations.

Impact If the backend halts, stale prices can remain "valid" indefinitely. Consumers may accept outdated data, misrepresent collateral/borrow states, and take unsafe actions using such stale prices.

Recommendation Implement a global state variable that consumers can consult to check the notarization backend status.

Developer Response Developers now verify the oracle's status before setting token prices. In addition, the Paleo token price is set independently via `set_price_paleo()`, while `set_price()` is reserved for other assets, including Aleo credits.

5.1.13 V-VER-VUL-013: Multiple request hashes can correspond to the same token ID

Severity	Low	**Commit**	04a6133
Type	Usability Issue	**Status**	Fixed
Location(s)	`aleo/programs/amm_oracle_v004.leo`		
Confirmed Fix At	ff5a68a		

Description The amm oracle associates a `request_hash` with a `token_id` through the `set_token_id` transition. This mapping enforces that a given request hash, which corresponds to a notarized backend API call, is tied to a specific token id. However, the reverse mapping is not unique i.e multiple distinct request hashes can be linked to the same `token_id`. See snippet below for reference.

```
    async transition set_token_id(request_hash: u128, token_id: field)-> Future {
        return (finalize_set_token_id(self.caller, request_hash, token_id));
    }

    async function finalize_set_token_id(from: address, request_hash: u128, token_id:
     field) {
        let owner: address = admin.get(true);
        assert_eq(from, owner);
        let allowed: bool = amm_reserve_state_v004.aleo/allowed_tokens.get(token_id);
        assert_eq(allowed, true);
        request_hash_token_id.set(request_hash, token_id);
    }
```

When prices are retrieved, the lookup is performed solely by `token_id` without considering the `request_hash`. This creates a mismatch in expectations, as consumers may believe the returned price corresponds to a specific trusted request hash. In reality, because multiple request hashes can update the same token ID, consumers may unknowingly rely on data from an untrusted or unintended feed. The risk is compounded by the absence of a mechanism to revoke deprecated request hashes, as noted in Missing mechanism to disable deprecated request hashes. An attacker could exploit this by reusing an alternate request hash for the same token ID, causing the system to resolve prices from a manipulated or stale data source.

Impact Consumers may rely on untrusted price feeds, leading to incorrect pricing for assets.

Recommendation Ensure that the token prices fetched are associated with both the `token_id` and `request_hash`, or enforce a one-to-one mapping so that a token cannot be linked to multiple request hashes.

Developer Response The developers now ensure that if a token ID is already mapped to a request hash, a new price feed cannot be linked to that token ID until the previous mapping is cleared.

5.1.14 V-VER-VUL-014: Hardcoded block time may diverge from actual network average

Severity	Low	**Commit**	04a6133
Type	Usability Issue	**Status**	Fixed
Location(s)	`aleo/programs/` ▶ `amm_reserve_state_v004.leo` ▶ `amm_user_state_v004.leo`		
Confirmed Fix At	980b901, 86f664e		

Description The protocol uses a hardcoded block time to estimate the time difference between interest updates. This value was originally set to `2` seconds per block, based on current mainnet observations. However, in the current implementation it has been set to `3`, which is not the intended value. See snippet for reference.

```
const BLOCK_TIME:u32=3u32;
const SECONDS_IN_YEAR:u128=31536000u128;
```

Moreover block production on Aleo is not constant, and the average block time can shift over time or vary due to differing network conditions. Depending on a fixed constant creates introduces a risk if the block time ever drifts from the network average. Even small differences in block intervals can compound over many blocks, leading to significant errors in calculations that depend on accurate time measurements. This can affect key protocol workflows such as interest accrual, borrowing, and liquidation.

Impact Relying on a hardcoded block time can cause time based calculations to become inaccurate, if the average block production time of the network ever diverges. This can majorly impact interest calculations, as well as borrowing and liquidation outcomes.

Recommendation Make the block time a configurable parameter rather than a fixed constant, so it can be updated if the network average changes. Furthermore, it should be regularly configured to accurately reflect the network average.

Developer Response The developers implemented the suggested fix.

5.1.15 V-VER-VUL-015: Treasury fees not deducted from available liquidity

Severity	Low	**Commit**	04a6133
Type	Logic Error	**Status**	Fixed
Location(s)	`aleo/programs/amm_reserve_state_v004.leo:250, 311`		
Confirmed Fix At	8bc7e83		

Description Whenever the treasury fees is computed, it is added to `treasury_claims` but not deducted from `total_available_liquidity` for the corresponding reserve market. This means the system continues to report liquidity that is, in practice, no longer available to users to borrow or withdraw since it can be claimed by the treasury at any time. As a result, the reported liquidity may overstate the true funds accessible within the pool.

```
1    //Veridise-elided
2
3        total_available_liquidity.set(collateral, new_available_liquidity);
4    treasury_claims.set(reserve, treasury_claims.get_or_use(reserve, 0u128) +
     treasury_interest);
```

The suggested adjustment should occur whenever treasury interest is computed, in functions such as `finalize_update_user_amt_borrow()` and `finalize_update_user_liq()`. Without making this change, the total available liquidity may continue to be tracked incorrectly which will result in incorrect protocol accounting for core protocol actions such as interest calculations, borrows, withdrawals and liquidations.

Impact Since the treasury fees is never deducted, the `total_available_liquidity` will be tracked incorrectly. Overtime the fees claimed by the treasury may grow to a substantially large value causing significant misrepresentation in protocol accounting, since the total liquidity is also used to compute the utilization rate.

Recommendation Each time the treasury fee is computed, reduce the fee amount from the `total_available_liquidity` for that asset to reflect that.

Developer Response The treasury fee is now accurately deducted from the total available liquidity in the highlighted locations.

5.1.16 V-VER-VUL-016: Small liquidation cap creates inefficient liquidation engine

Severity	Low	**Commit**	04a6133
Type	Usability Issue	**Status**	Acknowledged
Location(s)	`aleo/programs/amm_user_state_v004.leo:361-362`		
Confirmed Fix At	N/A		

Description: The money market protocol implements a liquidation mechanism where users can be liquidated when their health factor falls below the liquidation threshold. The protocol calculates a user's health factor based on their `total_collateral_balance_USD`, `total_borrow_balance_USD`, and `current_liquidation_threshold`. When liquidation occurs, the `check_on_liquidation()` function in `amm_user_state_v004.leo` caps the maximum liquidatable debt amount to only the excess above the maximum LTV ratio. The liquidation cap is calculated as:

```
let max_p_amount_to_Liquidate_USD: u128 = overflow_subtract(user_data.
    total_borrow_balance_USD, exa_mul((user_data.total_collateral_balance_USD),
    user_data.max_LTV)) as u128;
```

This calculation limits liquidation to only the debt amount that exceeds what would be allowed at the maximum LTV ratio. The `actual_amount_to_liquidate` is then capped to this maximum value, preventing liquidators from clearing larger portions of liquidatable positions in a single transaction. Note that although the excess of debt to the desired LTV ratio is liquidated, the position will not reach this LTV after the liquidation because the code still needs to consider the collateral to be liquidated.

This approach creates a suboptimal liquidation mechanism. Consider a scenario where `max_LTV = 0.7`, `liquidation_threshold = 0.8`, `user_debt = 900 USD`, `user_collateral = 1000 USD`, and `user_LTV = 0.9`. The maximum liquidatable amount would be calculated as `900 - (1000 * 0.7) = 200 USD`. After liquidating 200 USD of debt, the user would have `new_user_debt = 700 USD`, `new_user_collateral = 800 USD`, and `new_user_LTV = 0.875`, which still exceeds the liquidation threshold and requires additional liquidation rounds.

Impact The liquidation cap significantly reduces the efficiency of the liquidation mechanism during periods of high volatility. Multiple small liquidations are required to restore underwater positions to health, creating several negative consequences: increased gas costs for liquidators due to multiple transactions, slower position recovery during market stress, and potential accumulation of bad debt if liquidations cannot keep pace with declining collateral values.

Recommendation Consider adopting the AAVE approach to allow liquidations of up to 50% of a position's debt.

Reference: `https://aave.com/docs/developers/liquidations`

Developer Response The developers acknowledged the issue but decided not to introduce any code changes.

5.1.17 V-VER-VUL-017: Global debt and collateral temporarily inaccurate between liquidation steps

Severity	Low	**Commit**	04a6133
Type	Usability Issue	**Status**	Acknowledged
Location(s)	`aleo/programs/amm_reserve_state_v004.leo:238`		
Confirmed Fix At	N/A		

Description The protocol maintains global accounting for total borrowed and deposited amounts across all reserves through `total_borrowed_amount` and `total_deposited_amount` mappings in `amm_reserve_state_v004.leo`. During liquidations, the protocol follows a two-phase process: initial locking and final settlement during `finalize_claim_liquidation()`.

In the liquidation check function `check_on_liquidation()` in `amm_user_state_v004.leo`, the protocol calculates the actual liquidatable amounts (`actual_amount_to_liquidate` and `principal_amount_needed`) based on position health and liquidation rules. However, liquidators must provide estimated amounts (`purchase_amount` and `collateral_to_liquidate`) that can be up to 5% higher than the calculated amounts to account for price movements.

During the first phase in `finalize_update_user_liq()`, the global totals are updated using the liquidator's provided amounts rather than the actual liquidated amounts:

```
1 total_borrowed_amount.set(reserve, total_borrowed_amount.get(reserve) +
    borrow_interest - purchase_amount); total_deposited_amount.set(collateral,
    total_deposited_amount.get(collateral) + interest - collateral_to_liquidate);
```

The difference between provided and actual amounts is tracked in individual user mappings but not immediately reflected in global totals. While the accounting gets rebalanced during the second phase (`finalize_claim_liquidation()`), the global totals remain incorrect during the interim period between liquidation initiation and claim.

Impact The temporary inaccuracy in global totals creates a window where the protocol operates with incorrect state data. This affects the computation of the utilization rate which in turns affects the computation of the interest rate. The protocol assumes liquidation claims will be executed immediately, but network congestion, liquidator failures, or malicious intentional behavior could extend this vulnerable period indefinitely.

Recommendation Just as the `liquidatee_locked_collatera` and `liquidatee_locked_borrow` mappings maintain the difference tracked by user and token, it is advisable to introduce mappings that tracks the accumulated values from all the liquidations. In this way, these mappings can be queried and added to `total_borrows` and `total_deposited` when needed.

Developer Response The developers acknowledged the issue but decided not to introduce any code changes.

5.1.18 V-VER-VUL-018: Utilization rate calculated with outdated total borrowed amount

Severity	Low	**Commit**	04a6133
Type	Logic Error	**Status**	Fixed
Location(s)	`aleo/programs/amm_reserve_state_v004.leo:405`		
Confirmed Fix At	cd9a282, 9f10279, e6273a0		

Description The protocol calculates interest rates based on utilization rates, which depend on the ratio of total borrowed amount to total deposits. In `finalize_update_RR_timestamp()` within `amm_reserve_state_v004.leo`, the utilization rate is computed as:

```
let utilization_rate_val: u128 = exa_div(total_borrowed, total_borrowed +
    new_available_liquidity);
```

The protocol uses compound interest where individual user debt grows over time through the global borrow cumulative index. However, `total_borrowed_amount` only gets updated when users interact with their positions through `finalize_update_user_amt_borrow` and `finalize_update_user_liq`. These functions calculate each user's accrued interest and adds it to the global total:

```
total_borrowed_amount.set(token_id, previous_total_borrowed + bamount +
    borrow_interest - ramount);
```

This creates a fundamental issue where the global `total_borrowed_amount` represents the sum of principal amounts plus interest only for users who have recently interacted with the protocol, not the true real-time total debt including all accrued interest across all positions.

For example, if Alice borrows 50 tokens at time 0 and Bob borrows 50 tokens after 10% interest has accrued, the `total_borrowed_amount` will show 100 tokens instead of the actual 105 tokens (50 + 5 accrued interest + 50 new borrowing). The utilization rate calculation will use 100 instead of 105, leading to lower rates.

Impact Understated utilization rates result in artificially low interest rates, reducing protocol revenue and lenders receive inadequate compensation for their capital, while borrowers benefit from subsidized rates that don't reflect true market demand.

Recommendation Implement a scaled balance approach where user positions are stored as normalized amounts divided by the current global index. Update the global total calculation to multiply the sum of normalized balances by the current global cumulative index, providing real-time accurate totals without requiring individual user position updates. For example, in the Alice and Bob case, we will have:

- `alice_borrowed_amount = 50`
- `bob_borrowed_amount = 50 / 1.10 ~= 45.45`
- `total_borrowed_amount ~= 95.45`

When reading `total_borrowed_amount` you can multiply it by the current global index `1.10` to obtain `105` without Alice needing to update her position.

Developer Response The developers implemented the suggested fix.

5.1.19 V-VER-VUL-019: Missing mechanism to disable deprecated request hashes

Severity	Warning	**Commit**	04a6133
Type	Maintainability	**Status**	Fixed
Location(s)	`aleo/programs/amm_oracle_v004.leo`		
Confirmed Fix At	ff5a68a		

Description The AMM oracle maps a request hash to a token ID, where the request hash is a hash of the underlying encoded oracle data request, that includes the API query, asset data, precision decimals etc. This request hash is then used to query and verify the attested price for the particular token from the oracle. See snippet below for context.

```
    async transition set_token_id(request_hash: u128, token_id: field)-> Future {
        return (finalize_set_token_id(self.caller, request_hash, token_id));
    }

    async function finalize_set_token_id(from: address, request_hash: u128, token_id:
     field) {
        let owner: address = admin.get(true);
        assert_eq(from, owner);
        let allowed: bool = amm_reserve_state_v008.aleo/allowed_tokens.get(token_id);
        assert_eq(allowed, true);
        request_hash_token_id.set(request_hash, token_id);
    }
```

However, there is no way to disable a request hash once set, even if it becomes obsolete. If the particular details of the underlying API request have updated, the request hash related to it may now be stale or unsupported but since there is no mechanism to disable support for a request hash it will still remain active within the oracle.

Impact Deprecated feeds will remain active indefinitely, and consumers may knowingly or unknowingly rely on stale or invalid data.

Recommendation Add a mechanism to disable support for a deprecated request hash.

Developer Response The developers added `remove_token_id()` which can be used to remove support for a deprecated request hash.

5.1.20 V-VER-VUL-020: Missing validation of token decimals when adding a reserve

Severity	Warning	Commit	04a6133
Type	Data Validation	Status	Fixed
Location(s)	`aleo/programs/amm_reserve_state_v004.leo:128-134`		
Confirmed Fix At	39f49a3		

Description In `add_reserve()`, when adding a new asset reserve, the `decimals` parameter is taken as input and stored in the `ReserveConfig`. However, this value is not validated against the registered decimals in the `token_registry`. See snippet below for context.

```
1    async function finalize_add_reserve(from:address, token_id: field,
     token_reserve_data:ReserveData, token_reserve_config: ReserveConfig) {
2        let owner:address=admin.get(true);
3        assert_eq(from, owner);
4        assert(reserve_data.contains(token_id) == false);
5        reserve_data.set(token_id, token_reserve_data);
6        reserve_config.set(token_id, token_reserve_config);
7        allowed_tokens.set(token_id, true);
8        let unique_count:u8=unique_token_count.get_or_use(true, 0u8);
9        assert(unique_count + 1u8 <= MAX_TOKEN_COUNT);
10       unique_token_count.set(true, unique_count+ 1u8);
11   }
```

Impact If an asset is configured with an incorrect decimals value, it will cause deposits, borrows, and liquidations to be misrepresented, leading to inaccurate protocol accounting.

Recommendation When adding a new asset reserve, validate that the provided `decimals` matches the value recorded in the `token_registry`.

Developer Response When adding a new asset reserve, the provided decimals are now validated against the decimals recorded in the token registry.

5.1.21 V-VER-VUL-021: Minor Inconsistencies

Severity	Warning	**Commit**	04a6133
Type	Maintainability	**Status**	Fixed
Location(s)	`aleo/programs/` ▶ `amm_oracle_interface_v002.leo:70-89, 92-111` ▶ `amm_reserve_state_v004.leo:272-273`		
Confirmed Fix At	5910e53		

Description At the following locations in the project, the analysts identified minor inconsistencies with the intended behaviour.

- `aleo/programs/amm_reserve_state_v008.leo`, `finalize_checks_on_borrow()`, line 272.
  - Both the asserts use a strict inequality, when comparing the `borrow_amount` with the `available_borrow` and `reserve_available_liquidity`. This will cause an issue in edge cases where the borrow amount is exactly the same as the specified amounts. The assert statements should be updated to reflect this edge case.
- `aleo/programs/amm_oracle_interface_v008.leo`.
  - In both transitions `claim_all_withdraw()` and `claim_all_repay()`, the specified `asset_token_id` contains 10 fields but the functions only call claims for the first five. This will cause the remaining assets in the list to remain unclaimed, and may cause confusion and usability concerns for a user.

Impact The present implementation is inconsistent with the desired behaviour. If left unaddressed, it can introduce minor issues in the user workflow.

Recommendations Address the concerns as mentioned above.

Developer Response The developers implemented the suggested fix.

5.1.22 V-VER-VUL-022: Division by zero in exa_div() returns 0 instead of panicking

Severity	Warning	**Commit**	04a6133
Type	Maintainability	**Status**	Acknowledged
Location(s)	`aleo/programs/amm_reserve_state_v004.leo:646-659`		
Confirmed Fix At	N/A		

Description The `exa_div` function silently handles division-by-zero by assigning the denominator `b` to `EXA` and returning `0`. This behavior deviates from standard arithmetic expectations, where a division by zero should result in a panic or explicit error. See snippet below for context.

```
inline exa_div(a:u128,b:u128)->u128{
    if b == 0u128 {
        b = EXA;
        return 0u128;
    }

    let dividend_multiplier:u128=a * HALFEXA;
    let q1:u128=dividend_multiplier / b;
    let r1:u128=dividend_multiplier % b;
    let inter1:u128=r1 * HALFEXA;
    let q2:u128=inter1 / b;
    let r2:u128=inter1 % b;
    return q1*HALFEXA + q2;
}
```

The likely reason for this design is to avoid panics in `finalize_update_user_data()` when calculating `liquidation_threshold_withdraw()` in cases where a user has fully withdrawn their collateral. However, handling this special case logic directly in a low level math helper function introduces significant risk. Any other part of the code that uses `exa_div()` will inherit this non-standard behavior, making it easy to misuse or overlook. This can silently lead to incorrect calculations in scenarios where a denominator unexpectedly evaluates to zero, if used incorrectly.

Impact Handling division-by-zero in this manner can cause incorrect calculations if `exa_div()` is unknowingly misused. This can lead to inconsistent or incorrect protocol accounting that can be exploited.

Recommendation Handle division by zero within `exa_div()` with proper error handling. Any special edge-case should be handled outside this function wherever the special case applies.

Developer Response The developers acknowledged the issue but decided not to introduce any code changes.

Actually, the issue with the ternary expression is that, although it appears that only one branch should execute, during compilation all cases are evaluated. So, when the second value in exa_div is 0, it still

causes a division-by-zero error. That's why we decided to handle the zero case directly within the inline implementation of exa_div.

5.1.23 V-VER-VUL-023: Price updates rely only on whitelisted callers without oracle validation

Severity	Warning	**Commit**	04a6133
Type	Data Validation	**Status**	Acknowledged
Location(s)	`aleo/programs/amm_oracle_v004.leo:84-153`		
Confirmed Fix At	N/A		

Description The `finalize_set_price` function accepts a `request_hash`, `token_price`, and `timestamp` directly from the caller. The only protection applied is verifying that the caller is an allowed program. Once this passes, the function updates the price for the associated `token_id` without checking whether the provided values are valid or correspond to attested data within the oracle.

It is unsafe to assume that all whitelisted callers will always provide correct data. If a whitelisted program is compromised or misconfigured, it can set arbitrary prices. To prevent this, the request parameters should be validated directly against the `vlink_oracle_v0002` program, which maintains notarized attestations and guarantees that prices are backed by the oracle verification backend. This provides stronger security guarantees.

Impact Compromised or buggy whitelisted callers can set arbitrary prices, leading to manipulated asset valuations.

Recommendation Validate the `token_price` and `timestamp` against the attested data linked to the given `request_hash` in the `vlink_oracle_v0002` program, instead of relying solely on whitelisted callers.

Developer Response The developers acknowledged the issue but decided not to introduce any code changes.

5.1.24 V-VER-VUL-024: Unused Code

Severity	Info	**Commit**	04a6133
Type	Maintainability	**Status**	Fixed
Location(s)	`aleo/programs/` ► `amm_admin_v006.leo:48-57` ► `amm_interface_liquidation_v002.leo:67-78` ► `amm_user_state_v004.leo:34`		
Confirmed Fix At	61d5f68		

Description The following program constructs are unused:

1. `aleo/programs/amm_admin_v008.leo`
   - ► In `give_token()` and `finalize_give_token_private()`, the parameter `token_id` is never used.
2. `aleo/programs/amm_user_state_v008.leo`
   a) The constant `BLOCK_TIME` is currently unused.
3. `aleo/programs/amm_interface_liquidation_v008.leo`
   a) In `liquidation()`, the `liquidator_user` and `liquidating_user` are unused.

Impact These constructs may become out of sync with the rest of the project, leading to errors if used in the future.

Recommendation Remove the unused code.

Developer Response The developers have removed the unused code.