Veridise

Auditing Report

Hardening Blockchain Security with Formal Methods

FOR

Arcane Finance AMM

Veridise Inc.

June 13, 2024

► **Prepared For:**

Arcane Finance
https://www.arcane.finance/

► **Prepared By:**

Benjamin Mariano
Mark Anthony

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

► **Version History:**

June 13, 2024 V2
May 30, 2024 V1

Contents

1 Executive Summary

From May 13, 2024 to May 30, 2024, Arcane Finance engaged Veridise to review the security of their Arcane Finance AMM. The review covered a Uniswap V2-style AMM written in Leo. Veridise conducted the assessment over 4 person-weeks, with 2 engineers reviewing code over 2 weeks on commit `0c84184`. The auditing strategy involved extensive manual code review performed by Veridise engineers.

Project summary. The security assessment covered the Leo programs implementing the AMM. These included the Arcane Token, which is used to wrap tokens for interacting with the system, multiple vaults which are used to transfer funds in and out of the system, as well as the pool and exchange programs which contain swapping and liquidity logic.

Code assessment. The Arcane Finance AMM developers provided the source code of the Arcane Finance AMM contracts for review. The source code is mostly original code written by the Arcane Finance AMM developers. However, it is heavily inspired by Uniswap V2 AMMs, as well as Curve stable coin pools. It contains some limited documentation in the form of READMEs and documentation comments on functions and storage variables, although auditors did note that some of the provided documentation was out of date.

The source code contained a test suite, which the Veridise auditors noted covered basic behaviors of the protocol. However, auditors did find that many behaviors and APIs are entirely untested.

Summary of issues detected. The audit uncovered 20 issues, 5 of which are assessed to be of high or critical severity by the Veridise auditors. Critical issues included price manipulation (V-ARC-VUL-004), stealing or overwriting existing funds (V-ARC-VUL-001,V-ARC-VUL-002,V-ARC-VUL-003), and blocking intended pool operations (V-ARC-VUL-007,V-ARC-VUL-005). The Veridise auditors also identified a few other medium and low issues, including logic errors that block certain pool functionalities (V-ARC-VUL-006) and centralization risks (V-ARC-VUL-009).1 informational finding and 8 warnings were also reported to the developers. The Arcane Finance AMM developers have indicated an intent to fix these issues.

Recommendations. After auditing the protocol, the auditors had a few suggestions to improve the Arcane Finance AMM.

Testing. Auditors noted that tests written by developers only seemed to test *expected* behaviors of the protocol, but did not test corner cases and less common behaviors. Many of the bugs found may have been caught with such testing (e.g., burning more than owned, using existing deposit/voucher IDs, etc.). We suggest the auditors add more tests to their suite to test corner case behavior of their protocol.

Code Organization. Auditors found a number of instances where there were typos, misleading comments, and unnecessary code. While these issues may seem inconsequential, they can lead

to code which is hard to understand and can mask serious security vulnerabilities. We suggest developers carefully read through all comments and code to fix typos and update outdated comments. Additionally, auditors noticed many areas where code appeared to be copied and pasted, resulting in duplicate and unnecessary code. While some of this is due to unavoidable restrictions of the Leo language, we encourage developers to carefully consider each location where they copy and paste to determine if shared logic can be factored out into a single location or simplified. When possible, we suggest avoiding duplicate or unnecessary code.

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
Arcane Finance AMM	`0c84184`	Leo	Aleo

Table 2.2: Engagement Summary.

Dates	Method	Consultants Engaged	Level of Effort
May 13 - May 30, 2024	Manual	2	4 person-weeks

Table 2.3: Vulnerability Summary.

Name	Number	Acknowledged	Fixed
Critical-Severity Issues	4	4	4
High-Severity Issues	1	1	1
Medium-Severity Issues	3	3	3
Low-Severity Issues	3	3	2
Warning-Severity Issues	8	8	4
Informational-Severity Issues	1	1	0
TOTAL	20	20	14

Table 2.4: Category Breakdown.

Name	Number
Logic Error	11
Data Validation	6
Authorization	1
Frontrunning	1
Maintainability	1

3 Audit Goals and Scope

3.1 Audit Goals

The engagement was scoped to provide a security assessment of Arcane Finance AMM's Leo programs. In our audit, we sought to answer questions such as:

- Is the AMM subject to price manipulation attacks?
- Can LP funds be stolen or lost?
- Is AMM math implemented correctly?
- Can funds be unexpectedly lost on swaps?
- Does the AMM have centralization risks?
- Can swaps be blocked by malicious users?
- Can funds be stolen from vaults?

3.2 Audit Methodology & Scope

Audit Methodology. To address the questions above, our audit involved a thorough manual review of the code.

Scope. The scope of this audit is limited to the Leo programs in the `programs/` folder, excluding `arc20_usdt.leo` and `arc20_usdt.leo` which implement mock tokens for testing, as well as `arcaneutil.leo` which contains an unused helper function.

Methodology. Veridise auditors started by reviewing Leo documentation as well as audit reports for similar protocols (i.e., V2-style AMMs). They then inspected the provided tests, and read the Arcane Finance AMM documentation. Finally, they performed an extensive manual review of the code. During the audit, the Veridise auditors met with the Arcane Finance AMM developers to ask questions about the code and share findings.

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

Vulnerability Report 4

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-ARC-VUL-001	Create funds on burn	Critical	Fixed
V-ARC-VUL-002	Overwrite existing LP deposits	Critical	Fixed
V-ARC-VUL-003	Overwrite existing vouchers	Critical	Fixed
V-ARC-VUL-004	Price manipulation on partial liquidity removal	Critical	Fixed
V-ARC-VUL-005	Banned users can still revoke	High	Fixed
V-ARC-VUL-006	Public access never granted	Medium	Fixed
V-ARC-VUL-007	Spoof predefined tokens	Medium	Fixed
V-ARC-VUL-008	Cannot withdraw all pool reserves	Medium	Fixed
V-ARC-VUL-009	Centralization Risks	Low	Acknowledged
V-ARC-VUL-010	Swap output sent to caller instead of receiver	Low	Fixed
V-ARC-VUL-011	AMM math can overflow	Low	Fixed
V-ARC-VUL-012	Failed swaps on bad protocol fee	Warning	Fixed
V-ARC-VUL-013	Inconsistent swap logic	Warning	Fixed
V-ARC-VUL-014	Limited use APIs	Warning	Acknowledged
V-ARC-VUL-015	Pool ID protocol fee key clash	Warning	Fixed
V-ARC-VUL-016	Discrepancies with UniswapV2	Warning	Acknowledged
V-ARC-VUL-017	Discrepancies with Curve	Warning	Acknowledged
V-ARC-VUL-018	User liquidity deposits can be DOSed	Warning	Acknowledged
V-ARC-VUL-019	Different approvers/receivers across APIs	Warning	Fixed
V-ARC-VUL-020	Typos, comments, and unnecessary code	Info	Acknowledged

4.1 Detailed Description of Issues

4.1.1 V-ARC-VUL-001: Create funds on burn

Severity	Critical	**Commit**	0c84184
Type	Logic Error	**Status**	Fixed
File(s)	`arcanetoken_v2_0.leo`		
Location(s)	burn_private()		
Confirmed Fix At	N/A		

The transition `burn_private` is used to burn some amount from an existing Arcane Token record. To do so, it takes in a record and constructs a new record with the new amount as follows.

```
transition burn_private(token: token, public amount: u128) -> token {
    return token {
        owner: token.owner,
        amount: amount,
        token_id: token.token_id,
    } then finalize(token.token_id, amount);
}
```

Snippet 4.1: Implementation of transition `burn_private`

The `amount` of the new token record is simply taken as an input to the transition; there is no check that this input amount is less than the amount of the current token record.

Impact An attacker could arbitrarily increase their token balance by passing an `amount` greater than `token.amount`. Furthermore, doing this would arbitrarily decrease the total supply (via the logic in the finalize function).

Recommendation Change the amount in the returned token record to be `token.amount - amount`.

4.1.2 V-ARC-VUL-002: Overwrite existing LP deposits

Severity	Critical	**Commit**	0c84184
Type	Data Validation	**Status**	Fixed
File(s)	`arcn_pool_v2_1_1.leo`		
Location(s)	create_pool()		
Confirmed Fix At	N/A		

When creating a pool via the `create_pool` API, the caller passes in a `deposit_id` which is used to store information about the LP who provided the initial liquidity to the pool in the `amm_deposits` mapping.

```
finalize create_pool(
        pool_key: field,
        token1_id: u64,
        token1_amount: u128,
        token2_id: u64,
        token2_amount: u128,
        swap_fee: u128,
        pool_hash: field,
        deposit_id: field,
        pool_type: u8,
        amm_lp: u128,
        stable_lp: u128,
        ampl_coef: u128
) {
    ...

    Mapping::set(amm_pools, pool_key, pool);
    Mapping::set(created_pools, pool_hash, true);
    Mapping::set(amm_deposits, deposit_id, lp);
}
```

Snippet 4.2: Snippet from finalize `create_pool()`

However, there is no check in `create_pool` that the `deposit_id` provided has not already been used for another deposit in the `amm_deposits` mapping.

Impact The `amm_deposits` mapping is used to track the liquidity provided by an LP and is necessary for an LP to be able to withdraw their deposit. Without this check, an attacker could overwrite another LP's deposit information, causing them to lose their funds.

Recommendation Add a check that the `deposit_id` is not already in use.

4.1.3 V-ARC-VUL-003: Overwrite existing vouchers

Severity	Critical	**Commit**	0c84184
Type	Data Validation	**Status**	Fixed
File(s)	`arcn_pool_v2_1_1.aleo`		
Location(s)	swap_amm		
Confirmed Fix At	N/A		

In `arcn_pool_v2_1_1`, the mapping `amm_extras` maps voucher IDs to the residual amounts which can then be redeemed by users. In finalize `swap_amm()`, this mapping is updated after a swap and the key `extra_change_voucher` is then updated to hold amount `extra_amount`. See snippet below.

```
// Here. No  check to ensure voucher is not already in use.
Mapping::set(amm_extras, extra_change_voucher, extra_amount);

Mapping::set(amm_pools, pool_id, updated_pool);
Mapping::set(
    accrued_protocol_fees,
    token_in_id,
    Mapping::get_or_use(accrued_protocol_fees, token_in_id, 0u128) + collected_fee.
    protocol_in_fee
);
```

Snippet 4.3: Snippet from finalize `swap_amm()`

However, it is never ensured that `extra_change_voucher` is not already in use.

Impact An attacker can reuse the same voucher ID to overwrite another users existing voucher causing loss of funds for the user.

Recommendation Add a check to ensure the `extra_change_voucher` is not already in use.

4.1.4 V-ARC-VUL-004: Price manipulation on partial liquidity removal

Severity	Critical	**Commit**	0c84184
Type	Data Validation	**Status**	Fixed
File(s)	`arcn_pool_v2_1_1.leo`		
Location(s)	remove_amm_liquidity_part		
Confirmed Fix At	N/A		

The function `remove_amm_liquidity_part` is used to allow an LP to withdraw part of their liquidity. The function allows the user to pass in their expected `token1_amount` and `token2_amount` they expect to get back — the remaining parts of the LPs stake will be reinvested in the pool on their behalf.

```
finalize remove_amm_liquidity_part(pool_id: field, deposit_id: field, token1_id: u64,
     token2_id: u64, token1_amount: u128, token2_amount: u128) {
    ...

    updated_pool = PoolInfo {
        id: pool.id,
        token1_id: pool.token1_id,
        token2_id: pool.token2_id,
        swap_fee: pool.swap_fee,
        reserve1: pool.reserve1 - token1_amount,
        reserve2: pool.reserve2 - token2_amount,
        lp_total_supply: pool.lp_total_supply - deposit_amount_lp +
    new_lp_deposit_amount,
        ampl_coef: pool.ampl_coef,
        pool_type: pool.pool_type
    };
    Mapping::set(amm_pools, pool_id, updated_pool);
    Mapping::set(amm_deposits, deposit_id, new_lp_deposit_amount);
}
```

Snippet 4.4: Snippet from `example()`

No validation is done to ensure that the amounts of each token (`token1_amount` and `token2_amount`) are removed in proportion to the current amounts in the pool.

Impact Because the price in the pool is determined by the ratio of the token amounts held in the reserves, if a user can remove liquidity disproportionately from one reserve or the other, they can manipulate the price for swaps.

Recommendation Ensure the token amounts are removed in proportion to the current pool assets. One way to achieve this is to have the removal amount specified in LP tokens — thus, the only math required here is to compute the LP's total deposit amount, compute the amount of token 1 and token 2, and subtract those amounts from the reserves. If desired, a frontend could calculate the expected outputs of each token to report to the user before they withdraw. This would avoid the confusing logic require to allow the user to directly request a specific amount of tokens back.

Developer Response The developers chose to fix the bug by having the `remove_amm_liquidity_part` essentially compute two transactions in one: first remove all the LP's liquidity then add back in whatever remains after the desired amount is removed. Though somewhat complex, this logic should work assuming all reserves can never be removed from the pool. In another PR, the developers have ensured reserves never reach zero by burning a small amount of the initial deposit from the pool.

4.1.5 V-ARC-VUL-005: Banned users can still revoke

Severity	High	**Commit**	0c84184
Type	Data Validation	**Status**	Fixed
File(s)	`arcn_access_manager.aleo`		
Location(s)	revoke_private()		
Confirmed Fix At	N/A		

The transition `revoke_private` allows persons with revoke permissions to revoke access of an `access_id`. The revoked `access_id` is added to the `ban_list` and has its `grant_ability` and `revoke_ability` revoked. See snippet below for implementation. However, the finalize for the transition does not ensure that the `access_id` of the access token used to initiate the revoke is not in the `ban_list`. Therefore, an already banned `access_id` can be used to revoke and ban valid access tokens.

```
transition revoke_private(access: Access, public access_id: field, public to: address
    ) -> (Access) {
  assert(access.is_able_to_revoke);

  return Access {
    owner: access.owner,
    access_id: access.access_id,
    is_able_to_grant: access.is_able_to_grant,
    is_able_to_revoke: access.is_able_to_revoke
    } then finalize(access_id, to);
}

finalize revoke_private(access_id: field, to: address) {
  Mapping::set(ban_list, access_id, true);
  Mapping::set(grant_ability, to, false);
  Mapping::set(revoke_ability, to, false);
}
```

Snippet 4.5: Snippet from `revoke_private()`

Impact An access ID in the `ban_list` can still be used to revoke others.

Recommendation Add a check to ensure the access ID invoking the revoke is not in the `ban_list`.

4.1.6 V-ARC-VUL-006: Public access never granted

Severity	Medium	**Commit**	0c84184
Type	Logic Error	**Status**	Fixed
File(s)	`arcn_access_manager.leo`		
Location(s)	N/A		
Confirmed Fix At	N/A		

In the `arcn_access_manager` program, the `public_access` mapping is intended to store information about addresses which have been granted public access. Both the function `grant_public` and `revoke_public` check that only addresses which are in this mapping (or are the `INITIAL_ACCESS` address) are able to execute these functions, as shown below in the implementation of `grant_public`.

```
finalize grant_public (granter: address, to: address, is_able_to_grant: bool,
    is_able_to_revoke: bool) {
    let isInitialAddress: bool = granter == INITIAL_ACCESS;
    assert(isInitialAddress || Mapping::get_or_use(public_access, granter, false));
    assert(isInitialAddress || Mapping::get_or_use(grant_ability, granter, false));

    Mapping::set(grant_ability, to, is_able_to_grant);
    Mapping::set(revoke_ability, to, is_able_to_revoke);
}
```

Snippet 4.6: Snippet from finalize `grant_public()`

The `public_access` mapping is never written to, meaning no `granter` will ever be in the mapping `public_access`.

Impact As a result, even after `grant_public` is called for some particular `granter`, that address will not be able to call `grant_public` nor `grant_revoke` (nor will they pass a `check_access_public` check). The only address which will be able to do these things is the `INITIAL_ACCESS` address.

Recommendation Record addresses in the `public_access` mapping when they are granted public access.

Developer Response The fix has been implemented as recommended. The fix does expose the `to` address for private functions (i.e., `grant_private` and `revoke_private`) — this is intended behavior of these private records is to operate over records, not to maintain privacy of admin addresses.

4.1.7 V-ARC-VUL-007: Spoof predefined tokens

Severity	Medium	**Commit**	0c84184
Type	Logic Error	**Status**	Fixed
File(s)	See issue description		
Location(s)	See description		
Confirmed Fix At	N/A		

The program `arc20_usdc_vault_v1` is meant to act as a vault for `USDC` which allows wrapping `USDC` in an Arcane Token. The vault initializes `USDC` for use with Arcane Tokens by calling `create_arcane_token()` in the `arcanetoken_v2_0` program. Once initialized, a token ID cannot be used again. Because the `USDC` Token ID is predefined, the creation of its Arcane Token can be blocked by frontrunning – i.e., someone can register the token id `USDC_TOKEN_ID` before the call to `create_wrapped_token` can be completed. See snippet below for context.

```
1 transition create_wrapped_token() {
2     arcanetoken_v2_0.aleo/create_arcane_token(USDC_TOKEN_ID, USDC_TOKEN_NAME,
      USDC_SYMBOL, 6u8, 0u128);
3 }
```

Snippet 4.7: Snippet from program `arc20_usdc_vault_v1.alea`

This same problem exists for the `USDT` and `credits` vault as well.

Impact Frontrunning here could block the program from registering the token as expected. Furthermore, an attacker could register their own token, meaning they could mint themselves their token, wrap it in Arcane Token, and then call `withdraw_private_arc20_usdc` to steal USDC from the vault contract.

Recommendation Avoid predefined token IDs *or* ensure to immediately create the predefined tokens upon creation.

Developer Response The developers have added a reservation system that allows a token ID to be reserved first. While this could also technically be frontrun, it can be completed before deploying the vault programs, avoiding any more serious consequences.

4.1.8 V-ARC-VUL-008: Cannot withdraw all pool reserves

Severity	Medium	**Commit**	0c84184
Type	Logic Error	**Status**	Fixed
File(s)	`arcn_v2_pool_1_1.leo`		
Location(s)	get_d()		
Confirmed Fix At	N/A		

In `get_d()`, the following calculation occurs in the Newton-Raphson loop.

```
D_P = D_P * D / (reserve1 * 2u128);
D_P = D_P * D / (reserve2 * 2u128);
```

Snippet 4.8: Snippet from inline `get_d()`

`reserve1` and `reserve2` refer to either the current pool reserves or the amount of reserves after a deposit/withdraw for token 1 and 2 respectively. In this case, if either reserve is withdrawn down to `0`, the call will revert due to a division-by-zero.

One might think this is prevented by the following check which occurs previously in the code.

```
if (S == 0u128) {
    return 0u128;
}
```

Snippet 4.9: Snippet from inline `get_d()`

However, the semantics of Leo are such that the division-by-zero will still happen, even if this early return is triggered.

Impact This will block an LP from withdrawing their full liquidity if they are the last to withdraw from a pool. Similarly, if a pool were ever to reach a reserve level of `0`, no liquidity can be added to the pool. This will happen for both stable and regular pools.

Recommendation Avoid division-by-zero issues.

4.1.9 V-ARC-VUL-009: Centralization Risks

Severity	Low	**Commit**	0c84184
Type	Authorization	**Status**	Acknowledged
File(s)	`arcn_access_manager.aleo`		
Location(s)	See description		
Confirmed Fix At	N/A		

The `arcn_access_manager` imparts grant and revoke abilities to addresses (for the public transitions) and to access records (for the private transitions).

In particular, these abilities allow the following actions:

- Grant other addresses or records `grant_ability` and `revoke_ability`.
- Revoke and ban addresses or `access_id` of access records. In particular this can be extremely dangerous as a malicious person with revoke privileges can revoke access for all addresses/records except the `INITIAL_ACCESS` address.
- Setting protocol fee config.
- Whitelisting a new fee tier.

Impact A malicious person with these privileges can cause serious issues to the protocol. For example, the protocol fees can be set to `0`. A malicious revoker can revoke access of others.

Recommendation As these are all particularly sensitive operations, we would encourage the developers to carefully vet the addresses they give privileges to and to keep the number as limited as possible. If a private key is lost or a malicious entity gets access they can cause damage to the protocol.

We would encourage using a decentralized governance or multi-sig contract as opposed to a single account to manage these privileges, which can introduce a single point of failure.

Developer Response The developers have been notified of the issue but have yet to respond with acknowledgement or fixes.

4.1.10 V-ARC-VUL-010: Swap output sent to caller instead of receiver

Severity	Low	**Commit**	0c84184
Type	Logic Error	**Status**	Fixed
File(s)	`arcn_x_priv_v1_1.aleo`		
Location(s)	swap_amm_a_usdt()		
Confirmed Fix At	N/A		

The transition `swap_amm_a_usdt()` is a helper function intended to simplify the process of swapping from any arcane token to `USDT`.

In the context of the function, `self.caller` is the address initiating the swap and providing the input tokens. The `receiver` address is the address which is supposed to receive the output tokens. At the end of the swap operation `token_in_change` is the amount of leftover input token and `token_out` is the amount of output token received after the swap.

As can be seen in the snippet below the `token_in_change` is incorrectly sent to the `receiver`. And the `token_out` is sent to `self.caller` which is reverse to the intended behaviour.

```
1 let token_in_change: arcanetoken_v2_0.aleo/token = arcanetoken_v2_0.aleo/
      transfer_private(swap_res.0, receiver, swap_res.0.amount).0;
2 let token_out: arc20_usdt.aleo/token = arc20_usdt_vault_v1.aleo/
      withdraw_private_arc20_usdt(swap_res.1, swap_res.1.amount, self.caller).0;
```

Snippet 4.10: Snippet from `swap_amm_a_usdt()`

Impact The output `USDT` tokens are be sent to `self.caller` instead of `receiver` and the input token change is sent to the `receiver` instead of `self.caller`.

Recommendation The swap output should be sent to the `receiver` and the input token change should be sent to `self.caller`.

4.1.11 V-ARC-VUL-011: AMM math can overflow

Severity	Low	**Commit**	0c84184
Type	Logic Error	**Status**	Fixed
File(s)	`arcn_pool_v2_1_1.aleo`		
Location(s)	create_pool(), calculate_lp(), get_d(), get_y()		
Confirmed Fix At	N/A		

The Arcane Finance AMM math uses `128` bit unsigned integers to represent token values. The use of `u128` can lead to mathematical operations on larger numbers overflowing. Particularly, tokens with `18` decimals can easily cause an overflow in the current AMM math.

Let's consider the initial liquidity allocation. If we want to initialize a pool for `WETH/DAI`, both of which have `18` decimals and we deposit just `1 WETH` with roughly `4000 DAI` (as per the current price ratio) then the initial liquidity allocation is $xy = 10^{18} * 4 * 10^{18} * 1000 = 4 * 10^{39}$ which is larger than $3 * 10^{38}$, that is approximately the maximum number which can be represented with a `u128`. See snippet below for the implementation.

```
let amm_pool_lp: u128 = initial_amount1 * initial_amount2;
```

Snippet 4.11: Snippet from `create_pool()`

If we consider a case with one token having `18` decimals and the other token having `6` decimals. Then, even in this case the values blow up very quickly. For example, if `tokenA` has `18` decimals and `tokenB` has `6` decimals then in `calculate_lp()` as shown in snippet below, `new_lp1` will have roughly $10^{18} * 10^{6} * 10^{18} = 10^{42}$ decimals in the numerator for the nominal amount of token deposits.

```
let new_lp1: u128 = (token1_amount * pool.lp_total_supply) / pool.reserve1;
```

Snippet 4.12: Snippet from `calculate_lp()`

The same concerns exist for the stableswap calculations where the sum of the reserves is multiplied repeatedly with itself in the calculation of `D_p` as shown in the snippet below. These calculations can easily overflow if one of the tokens has `18` decimals.

```
for i: u8 in 0u8..8u8 {
    let D_P: u128 = D;
    D_P = D_P * D / (reserve1 * 2u128);
    D_P = D_P * D / (reserve2 * 2u128);

    D_prev = D;
```

Snippet 4.13: Snippet from `get_d()`

Impact At the moment Arcane finance supports `USDT`, `USDC`, and `aleo/credits`. All of these have `6` decimals. Therefore, pools between these tokens are not likely to overflow.

However, while it doesn't impact any of the current tokens, it could severely limit tokens that can be supported in the future. Especially, any token with `18` decimals.

Recommendation At the moment, Aleo does not not support `u256`. The developers should implement a solution which can multiply larger numbers while handling overflows gracefully.

Though not directly applicable as it is designed for `u256` values, this implementation from Uniswap is used to avoid overflow in intermediate calculations for multiplication and then division and could serve as a guide for types of calculations that might be necessary to support higher decimal tokens.

Developer Response The developers opted to simply disallow tokens with more than `6` decimals to avoid this issue.

4.1.12 V-ARC-VUL-012: Failed swaps on bad protocol fee

Severity	Warning	**Commit**	0c84184
Type	Data Validation	**Status**	Fixed
File(s)	`arcn_pool_v2_1_1.leo`		
Location(s)	validate_swap()		
Confirmed Fix At	N/A		

In `validate_swap()`, when computing the total amount taken in fees, the following computation is performed.

```
...
let collected_in_fee: u128 = (amount_in_without_fee - amount_in_with_fee) / FEE_BASE;
let collected_protocol_in_fee: u128 = (collected_in_fee * fee_config.rate as u128) /
    FEE_BASE;
let pool_in_fee: u128 = collected_in_fee - collected_protocol_in_fee;
...
```

Snippet 4.14: Snippet from inline `validate_swap()`

If the `fee_config.rate` is greater than `FEE_BASE`, the subtraction when computing `pool_in_fee` will fail underflow constraints.

Impact If the protocol fee rate is ever set above `FEE_BASE`, swaps for standard pools will fail.

Recommendation Add checks when setting the protocol fee that the rate is less than the `FEE_BASE`.

4.1.13 V-ARC-VUL-013: Inconsistent swap logic

Severity	Warning	**Commit**	0c84184
Type	Logic Error	**Status**	Fixed
File(s)	`arcn_pool_v2_1_1.leo`		
Location(s)	swap_stable()		
Confirmed Fix At	N/A		

When swapping on the regular pool, output fees are calculated as a percentage of the amount out *after* fees are applied on the input. For swapping on the stable pool, output fees are calculated *before* fees are applied on the input.

```
...
let amount_in_with_fee: u128 = (amount_in * (FEE_BASE - fee)) / FEE_BASE;
let amount_out_with_fee: u128 = (amount_out * (FEE_BASE - fee)) / FEE_BASE;

let token_in_fee: u128 = amount_in - amount_in_with_fee;
let token_out_fee: u128 = amount_out - amount_out_with_fee;
...
```

Snippet 4.15: Snippet from transition `swap_stable()`

Impact Extracting the output fee *before* fees are applied or the swap is enacted (as in the regular pool) may mean users will not get the full `amount_out` they have requested.

Recommendation Apply fees for stable swaps in the same way as done for regular swaps.

4.1.14 V-ARC-VUL-014: Limited use APIs

Severity	Warning	**Commit**	0c84184
Type	Logic Error	**Status**	Acknowledged
File(s)	`arcn_x_priv_v1_1.leo`, `arcn_x_pub_v1_1.leo`		
Location(s)	See issue description		
Confirmed Fix At	N/A		

When creating pools, it is required that the first token ID in the pool is less than the second token ID in the pool, as shown in the following snippet in `create_pool`.

```
transition create_pool(
        public pool_id: field,
        public owner: address,
        token1: arcanetoken_v2_0.aleo/token,
        public initial_amount1: u128,
        token2: arcanetoken_v2_0.aleo/token,
        public initial_amount2: u128,
        public swap_fee: u128,
        public deposit_id: field,
        public pool_type: u8,
        public ampl_coef: u128
) -> (PoolAdmin, arcanetoken_v2_0.aleo/token, arcanetoken_v2_0.aleo/token,
    LpTokenReceipt) {
        ...

    assert(token1.amount >= initial_amount1 && token2.amount >= initial_amount2);
    assert(token1.token_id < token2.token_id);

        ...
}
```

Snippet 4.16: Snippet from transition `create_pool()`

Despite this, in the exchange contracts, many of the removing liquidity APIs ignore this for pools involving Aleo Credits and USDT, which have fixed token IDs of `0` and `1` respectively. In particular, multiple APIs have been suggested for removing liquidity from pools where either Aleo Credits or USDT are the *second* token in the pool. For Aleo Credits, this will never happen and for USDT the only pool for which this could happen is an Aleo Credits / USDT pool which already have other APIs. The full list of the APIs which have this issue are the following functions (in both `arcn_x_priv_v1_1.leo` and `arcn_x_pub_v1_1.leo`):

- `remove_liq_a_pric`
- `remove_amm_liq_part_a_pric`
- `remove_liq_a_usdt`
- `remove_amm_liq_part_a_usdt`

Impact These functions will not be useful for users and make the interface to the protocol unnecessarily confusing.

Recommendation Remove these APIs.

Developer Response The developers have been notified of the issue but have yet to respond with acknowledgement or fixes.

4.1.15 V-ARC-VUL-015: Pool ID protocol fee key clash

Severity	Warning	**Commit**	0c84184
Type	Data Validation	**Status**	Fixed
File(s)	`arcn_pool_v2_1_1.leo`		
Location(s)	create_pool()		
Confirmed Fix At	N/A		

For pools, the global protocol fee is fetched by checking if the `pool_id` has a registered fee configuration in the `protocol_fee` mapping and otherwise using the default fee configuration at the `GLOBAL_PROTOCOL_FEE_KEY` entry in `protocol_fee` (as shown in the snippet from `swap_amm` as follows).

```
let global_fee_config: ProtocolFeeConfig = Mapping::get_or_use(
    protocol_fee,
    GLOBAL_PROTOCOL_FEE_KEY,
    ProtocolFeeConfig {
        rate: 0u64,
        foundation: SELF_ADDRESS
});

let fee_config: ProtocolFeeConfig = Mapping::get_or_use(protocol_fee, pool_id,
    global_fee_config);
```

Snippet 4.17: Snippet from finalize `swap_amm()`

However, when creating a pool, there is no check that the pool ID is not equal to the `GLOBAL_PROTOCOL_FEE_KEY`.

Impact If a pool ID is equal to the `GLOBAL_PROTOCOL_FEE_KEY`, this pool will never be able to adjust its protocol fee without changing the default protocol fee for all pools.

Recommendation Add a check to `create_pool` to disallow the use of a pool ID equal to the `GLOBAL_PROTOCOL_FEE_KEY`.

4.1.16 V-ARC-VUL-016: Discrepancies with UniswapV2

Severity	Warning	Commit	0c84184
Type	Logic Error	Status	Acknowledged
File(s)	arcn_pool_v2_1_1.aleo		
Location(s)	create_pool()		
Confirmed Fix At	N/A		

The Arcane Finance AMM is based on the UniswapV2 implementation. The core logic is largely the same although there are some subtle differences. As the UniswapV2 model is well-tested, deviating from it is not recommended. Listed below are some differences which have been identified and which can be potentially dangerous.

Initial Liquidity Allocation In UniswapV2, the amount of LP tokens minted to the first depositor is $\sqrt{xy}$ where x and y are the deposit amounts for the two assets. This has been changed to xy as can be seen in the snippet below.

```
let amm_pool_lp: u128 = initial_amount1 * initial_amount2;
let stable_pool_lp: u128 = get_d(initial_amount1, initial_amount2, ampl_coef);
```

Snippet 4.18: Snippet from `create_pool()`

As per the UniswapV2 whitepaper the initial liquidity supply was chosen in such a way to ensure that the liquidity pool share at any time is independent of the ratio at which liquidity was initially deposited. It also ensures that a liquidity pool share will never be worth less than the geometric mean of the reserves in that pool.

This also reduces the likelihood of rounding errors, since the number of bits in the quantity of shares will be approximately the mean of the number of bits in the quantity of asset x in the reserves, and the number of bits in the quantity of asset y in the reserves. See the equation below for some better context.

$$\log_2 \sqrt{xy} = \frac{\log_2 x + log_2 y}{2}$$

By changing the formula for the initial liquidity supply to xy the same guarantees do not exist.

Minimum Liquidity Amount Additionally, UniswapV2 makes certain provisions to ensure that a price of a single liquidity pool share does not appreciate to the point that it is infeasible for small depositors to meet this minimum value for deposits. To mitigate the above, UniswapV2 burns the first 10^{-15} pool shares that are minted sending them to the zero address instead of sending it to the minter. This dramatically increases the cost of inflating the value of liquidity pool shares. The above consideration is missing in the Arcane Finance AMM implementation.

Impact The UniswapV2 standard is rigorously tested and any deviation from the standard is ill-advised.

Recommendation Use $\sqrt{xy}$ instead of xy as the initial LP token amount minted to pool creator. Additionally, burn the first 10^{-15} shares to increase cost of inflating the value of liquidity pool shares.

Developer Response The developers have been notified of the issue but have yet to respond with acknowledgement or fixes.

4.1.17 V-ARC-VUL-017: Discrepancies with Curve

Severity	Warning	Commit	0c84184
Type	Logic Error	Status	Acknowledged
File(s)	arcn_pool_v2_1_1.aleo		
Location(s)	get_d() and get_y()		
Confirmed Fix At	N/A		

The stable swap implementation of the Arcane Finance AMM is based on the Curve Finance implementation.

Newton's method, also known as Newton–Raphson, is a root-finding algorithm which produces successively better approximations to the roots (or zeroes) of a real-valued function. In `arcn_pool_v2_1_1.aleo` the inline functions `get_d()` and `get_y()` make use of Newton's method to calculate the new value of `D` and `y`. See snippet below for details.

```
for i: u8 in 0u8..8u8 {
  let D_P: u128 = D;
  D_P = D_P * D / (reserve1 * 2u128);
  D_P = D_P * D / (reserve2 * 2u128);

  D_prev = D;
  D = (Ann * S + D_P * 2u128) * D / ((Ann - 1u128) * D + 3u128 * D_P);

  let tmp_res: bool = (max_u128(D, D_prev) - min_u128(D, D_prev) <= 1u128) ? true :
    false;

  if (tmp_res) {
      return D;
  }
}
```

Snippet 4.19: Snippet from `get_d()`

The convergence of Newton's method depends mainly on two factors:

- The initial guess for the root.
- The number of iterations.

If the initial guess is not close to the zero then Newton's method may keep ping ponging between multiple values and may never converge. Likewise in the case that the initial guess is not close enough if there are not enough number of iterations then the method may not be reach a sufficient approximation of the root.

The Curve Finance implementation uses `256` iterations whereas the above implementation only uses `9` iterations. This is potentially dangerous as just `9` iterations may not be enough for Newton's method to converge.

Impact With a lower number of iterations, Newton's method may not converge. This can have a negative effect on the entire calculation of the pool state and may render it incorrect.

Recommendation It is recommended to ensure that there are enough iterations for Newton's method to converge with practically negligible error probability.

Developer Response The developers have been notified of the issue but have yet to respond with acknowledgement or fixes.

4.1.18 V-ARC-VUL-018: User liquidity deposits can be DOSed

Severity	Warning	**Commit**	0c84184
Type	Frontrunning	**Status**	Acknowledged
File(s)	arcn_pool_v2_1_1.aleo		
Location(s)	add_amm_liquidity.aleo		
Confirmed Fix At	N/A		

The `add_amm_liquidity` transition accepts the field `deposit_id` as an input parameter. This `deposit_id` is public and can be viewed by anyone.

An attacker can frontrun a user's transaction by using the same `deposit_id` and causing the user's transaction to revert. Theoretically, the attacker can continuously frontrun transactions effectively DOS'ing users. In practice, blocking *all* users would likely be prohibitively costly, however, it might be feasible for an attacker to DOS all high value deposits.

```
transition add_amm_liquidity(
    public pool_id: field,
    owner: address,
    token1: arcanetoken_v2_0.aleo/token,
    public token1_amount: u128,
    token2: arcanetoken_v2_0.aleo/token,
    public token2_amount: u128,
    public deposit_id: field,
    public voucher1_id: field,
    public voucher2_id: field
)
```

Snippet 4.20: Snippet from `add_amm_liquidity()`

Impact An attacker can frontrun a user by using the same `deposit_id`. It may be too costly to frontrun each and every transaction, but because the `token_amounts` are public, an attacker can choose to target high value liquidity providers and frontrun them, limiting pool growth.

The same frontrunning technique can be used to block `voucher_ids`.

Recommendation The developers should consider adding some protections against frontrunning.

Developer Response The developers have been notified of the issue but have yet to respond with acknowledgement or fixes.

4.1.19 V-ARC-VUL-019: Different approvers/receivers across APIs

Severity	Warning	**Commit**	0c84184
Type	Logic Error	**Status**	Fixed
File(s)	`arcn_x_pub_v1_1.aleo`, `arcn_pool_pub_v2_1_1.aleo`		
Location(s)	See Description.		
Confirmed Fix At	N/A		

A common patter used in the codebase in both the exchanges and pool is to transfer public funds to a private token that can be used in a pool operation. For example, in the transition `create_pool_a_a`, the following code snippet is used public funds from the "approver" `self.caller` to the "receiver" `self.signer`.

```
let token2: arcanetoken_v2_0.aleo/token = arcanetoken_v2_0.aleo/
    transfer_from_public_to_private(
    token2_id,
    self.signer,
    self.caller,
    token2_amount
);
```

Snippet 4.21: Snippet from transition `create_pool_a_a()`

However, in other places in the code, a similar but slightly different patter is used, where public tokens are transferred from the "approver" `self.signer` to the "receiver" `self.caller`, as in the following snipper from the transition `swap_amm_a_a`.

```
let token_in: arcanetoken_v2_0.aleo/token = arcanetoken_v2_0.aleo/
    transfer_from_public_to_private(
    token_in_id,
    self.caller,
    self.signer,
    amount_in
);
```

Snippet 4.22: Snippet from transition`swap_amm_a_a()`

The above pattern is used extensively, including in the following transitions:

- `arcn_pool_pub_v2_1_1.aleo/add_amm_liq_a_a()`
- `arcn_x_pub_v1_1.aleo/swap_route2_pric()`
- `arcn_x_pub_v1_1.aleo/create_pool_usdt_a()`
- `arcn_x_pub_v1_1.aleo/create_pool_a_usdt()`
- `arcn_x_pub_v1_1.aleo/swap_amm_a_usdt()`
- `arcn_x_pub_v1_1.aleo/add_amm_liq_usdt_a()`

Impact Inconsistent behaviors across these functions could yield a confusing and difficult-to-use interface for users.

Recommendation Make the pattern consistent across all functions.

4.1.20 V-ARC-VUL-020: Typos, comments, and unnecessary code

Severity	Info	**Commit**	0c84184
Type	Maintainability	**Status**	Acknowledged
File(s)	See issue description		
Location(s)	See issue description		
Confirmed Fix At	N/A		

Description In the following locations, the auditors identified minor typos and unnecessary/unintuitive code:

- `arcanetoken_v2_0.leo`:
  - `struct metadata`: The field `total_supply` actually indicates the *max* total supply and if it is set to `0` then there is no max. The name should be changed to `max_total_supply` and the `0` behavior should be explicitly documented.
  - `finalize burn_private`: The check `new_supply >= 0u128` is trivially satisfied as `new_supply` is a `u128` which is always greater than or equal to `0`.
  - `finalize burn_private`: There is no `burn_public` API.
  - `finalize transfer_from_public`: The check `allowance >= amount` is redundant as it is performed on the subtraction `allowance-amount` on the next line.
  - `transition transfer_private_2rec`: The fixed loop from `0u8..2u8` to compute the sum can just be turned into the single statement `let sum: u128 = amounts[0] + amounts[1]`. The same is true of the subsequent `2rec` and `3rec` variants.
  - `transition transfer_private_2_rec`: The check `sum >= amount` is redundant as it is performed on the subtraction `sum-amount` when computing the `amount` for the `change` token record.
- `arcn_pool_v2_1_1.leo`:
  - `const MAX_U64`: This constant is never used.
  - `struct DepositKey`: This struct is never used.
  - `inline safe_div`: This inline is never used.
  - `transition create_pool`: The check `token1.token_id != token2.token_id` is not necessary as it is later checked that `token1.token_id < token2.token_id`.
  - `finalize create_pool`: The logic to choose between the `amm_lp` and `stable_lp` depending on the pool type can be moved to the `transition` function to simplify the `finalize` interface with a single `lp` value and reduce gas costs.
  - `transition remove_liquidity`: The check that `extra_voucher1` and `extra_voucher2` are not equal is redundant as in `finalize` when they add each entry to `amm_extras` they first check for collisions, which would catch if the two were equal on the entry of `extra_voucher2`.
  - `struct CollectedFees`: The fields `pool_in_fee` and `pool_out_fee` are never used.
  - `inline validate_swap`: The check that `actual_amount_out >= amount_out` is redundant given the subtraction `actual_amount_out-amount_out` performed on line 732. The same is true in `validate_stable_swap`.
  - `invline validate_stable_swap`: The variable `fee` is unused.

- `inline validate_stable_swap`: Consider combining the two subtractions on lines 876-877 on `reserve_out` into the single subtraction: `reserve_out -= (actual_amount_out + protocol_out_fee)`.
- `transition swap_amm`: This only contains a non-zero check on `amount_out` but not `amount_in`. Similarly, `transition swap_stable` only contains a non-zero check on `amount_in` and not `amount_out`. Both should include both checks.
- `struct PoolInfo`: The comment for `id` indicates that the pool ID should be the hash of the two token IDs. This is not actually the expectation and so the comment should be removed.
- `record PoolAdmin`: The admin for a pool is set on the creation of a pool. However, the pool admin has no special rights. The admin record should be removed or the admin should be given some additional rights for the pool.
- `get_d` and `get_y`: The computation of `tmp_res` with a ternary operator is redundant — just use the boolean.

▶ `arc20_usdt_vault_v1.leo`:

- This vault does not support `deposit_public` and `withdraw_public` as does the USDC vault.
- `withdraw_public_arc20_usdt`: For lowest gas consumption in case of revert, the call to `burn()` should happen before the call to `transfer_public()`.

▶ `arcn_x_priv_v1_1.leo`:

- `transition swap_amm_pric_a`: The input argument `amount_in` is unused.

▶ `arcn_x_pub_v1_1.leo`:

- `transition swap_route2_pric`: `token_out_ids[1u8]` should just be `0` (the ID for Aleo Credits). It might make more sense to add an assertion to this effect or just ask the user to provide a single output token ID for the first leg of the swap.

▶ `credits_vault_v1.leo`:

- `withdraw_puc_from_signer`: This function is unused and is just a wrapper around functionality that already exists for Aleo credits.

Impact These minor errors may lead to future developer confusion.

Developer Response The developers have been notified of the issue but have yet to respond with acknowledgement or fixes.

