# Algofi AMM Security Assessment

AlgoFi Inc
March 31, 2022

**Prepared By**
Kevin Henry
Aleksandar Kircanski

**Prepared For**
John Clarke

# 1 Executive Summary

## Synopsis

During February and March 2022, Algofi engaged NCC Group to perform an implementation review of the Algofi AMM smart contract collection. The AMM collection includes a pool manager functionality, a basic Uniswap V2-like AMM implementation, together with an iterative method which computes token prices according to the StableSwap invariant. The AMM is written in PyTEAL, a Python-like language that gets compiled to Algorand's native smart contract language, TEAL.

The assessment was delivered over the course of 20 person-days by two consultants. Full source code access was provided and the Algofi Team also readily answered consultant's questions in a dedicated Slack channel. Four meetings were held in total during the project, including code walkthroughs by the Algofi Team and status updates on work progress by NCC Group consultants.

## Scope

Commit `daaaf4efa6` of the https://github.com/algofiorg/algofi-amm/ repository was in scope. It contains three components:

- **Pool manager:** Pool registration, global cross-pool variables, fees and limits, etc.
- **AMM pool implementation:** Pooling, burning, swapping and flash lending, etc.
- **StableSwap invariant**: Iterative price calculation according to StableSwap [1].

## Key Findings

Some of the findings reported during this engagement include:

- **Temporary Early-Liquidity Pool Monopolization:** An early liquidity provider can temporarily monopolize the pool. It will be possible to de-monopolize the pool, but it will require a flash loan and induce a flash loan fee.
- **False Positives In Overflow Condition Evaluation:** The amount of LP tokens users receive can be less than users expect.
- **Inner Flash Loan Transactions Can Self-Reference Pools:** Some edge-case transactions which would work in normal circumstances would fail if executed inside a flash loan context.

## Testing Methodology and Overall Results

The strategy used during the review was to search for general unintended behavior in contract functionalities. For each function, an intuition was built on the expected behavior of the function. Such an intuition was built based on both the function code and available documentation (source code comments). When reasonable, testing was used to confirm the intuition.

This review did not uncover substantial security flaws. It should be noted, however, that the absence of security bugs is not guaranteed, as the review was best-effort and should not be considered as an absolute safety guarantee.

As for improvements, it was observed that the amount of unit tests for a correctness-critical project of this type should be higher. Consider dedicating internal engineering time to the perform the following. For each function in the contract, intuition should be built on how it behaves in edge-cases. Write tests to confirm the behavior and attempt to discover unexpected quirks (missed by initial code review). If any additional quirks are discovered by this procedure, they should be closely examined in terms of their security consequences.

1. StableSwap - "Efficient mechanism for Stablecoin liquidity" paper



# 2 Finding Details

Low

## False Positives In Overflow Condition Evaluation

| | | | |
|---|---|---|---|
| **Overall Risk** | Low | **Finding ID** | NCC-E003791-R3D |
| **Impact** | Low | **Category** | Other |
| **Exploitability** | None | **Status** | New |

### Impact
The amount of LP tokens users receive can be less than users expect.

### Description
Whenever Algofi users provide liquidity to a pool, the `calculate_lp_issuance` function is used to estimate the number of LP tokens that need to be issued. In the case there is no initial liquidity in the pool, the geometric mean function is used:

```
    def calculate_lp_issuance(self, pool_is_empty):
        # calculate lp issuance based and adjusted pool amounts
        # pools being seeded for the first time will use the sqrt(a1*a2) as the initial lp
        ↳ issuance
        return If(pool_is_empty,
                Seq([
                    If(Int(MAX_INT_U64) / self.adjusted_pool_asset1_amount_store.load() > self.ad
                    ↳ justed_pool_asset2_amount_store.load(),
                        self.lp_issued_store.store(Sqrt(self.adjusted_pool_asset1_amount_store.load()
                        ↳ * self.adjusted_pool_asset2_amount_store.load())),
                        self.lp_issued_store.store(Sqrt(self.adjusted_pool_asset1_amount_store.load())
                        ↳ * Sqrt(self.adjusted_pool_asset2_amount_store.load()))
                    )
                ]),
# ..snip..
```

Denote `self.adjusted_pool_asset1_amount_store.load()` by `x` and `self.adjusted_pool_asset2_amount_store.load()` by `y`. The `Int(MAX_INT_U64) / x > y` check aims to validate whether `x*y` overflows the unsigned 64-bit integer. It is worth noting that this check yields false positives. For example, if `x = 2` and `y = 9223372036854775807` does not overflow the 64-bit unsigned integer, but the overflow condition gets triggered by the shown `If` condition.

Other instances where the same overflow check is applied include the following functions:

- `save_latest_cumsum_time_weighted_price`
- `save_latest_cumsum_volume`

Finally, it is worth noting that in the code snippet above, the non-overflow condition triggers calculating the geometric mean as `sqrt(x*y)`, whereas the overflow condition calculates it as `sqrt(x)*sqrt(y)`. Since this is integer arithmetic, the latter can be significantly less than the former. For example, the edge condition above will result in a significant difference between the released LP tokens. If `x = 2` and `y = 9223372036854775807`, `sqrt(x*y) = 4294967296` and `sqrt(x) * sqrt(y) = 3037000499`.

### Recommendation
Modify the check to remove the `MAX_INT_U64` overflow false positive.



## Location

https://github.com/algofiorg/algofi-amm/blob/daaaf4efa6912ac6290644906 77db6e9cee8be86/contracts/pool.py#L388



Low

# Temporary Early-Liquidity Pool Monopolization

| | | | |
|---|---|---|---|
| **Overall Risk** | Low | **Finding ID** | NCC-E003791-PF6 |
| **Impact** | Low | **Category** | Other |
| **Exploitability** | Low | **Status** | New |

## Impact

An early liquidity provider can temporarily monopolize the pool. It will be possible to de-monopolize the pool, but it will require a flash loan and induce a flash loan fee.

## Description

Early liquidity provider pool monopolization in AMM exchanges is an issue that was discussed in several public audit reports. For example, it is discussed in RSK Swap's audit report[2] from early 2020. In its initial form, the steps that result in monopolization are as follows:

- Before there is any liquidity in the pool, the attacker pools a small amount of pair tokens, resulting in a modest amount of Liquidity Pool (LP) shares.
- The attacker then sends a very large amount of the pair tokens to the contract, and invokes `sync`.

In Algofi's AMM implementation, the same process does *not* appear to work. While it would be possible for anyone to fund the pool's address with tokens on the corresponding Algorand Standard Assets (ASAs), it would not be possible to sync the ASA balances with the pool's `balance_1` and `balance_2` variables. As such, the pool would not be affected by the supplied tokens.

Another way to attempt to deflate the LP token would be through swapping the pair tokens during the early pool stages. Roughly, the attacker proceeds as follows:

- Before all other participants, mint a small number of LP tokens by providing a modest amount of tokens A and B.
- Next, swap a large amount of token A against token B, creating a severe disbalance between token A and token B holdings in the pool.
- Finally, swap sufficient tokens of token B against token A, so that the ratio between the two token holdings reflects the real token A vs. token B price ratio.

The number of LP tokens the pool emitted remains small, whereas the token pair liquidity is high, which makes minting new LP tokens prohibitively expensive. This attack was described in another audit report of a Uniswap-like protocol[3]. As pointed out by the Algofi Team, this attack does not work, due to the `validate_asset_ratio` function, called on all of the pool operations except when tokens are burned:

```
    def validate_asset_ratio(self):
        return Seq([
            Assert(self.balance_1.get() >= Int(MIN_POOL_BALANCE)),
```

2. RSK Swap's Uniswap V2 audit report, January-March 2020 https://rskswap.com/audit.html#orgc7f8ae1
3. Runtime Verification Audit of Pact.fi Nov-Feb 2021-2022 https://github.com/runtimeverification/publications/blob/main/reports/smart-contracts/Pact_Fi.pdf

```
            Assert(self.balance_2.get() >= Int(MIN_POOL_BALANCE)),
            Assert(self.balance_1.get() / self.balance_2.get() < Int(MAX_ASSET_RATIO)),
            Assert(self.balance_2.get() / self.balance_1.get() < Int(MAX_ASSET_RATIO)),
        ])
```

The question is whether the strict rules imposed by `validate_asset_ratio` can be abused by an attacker. For simplicity, consider tokens A and B with roughly equal price ratio. The closest scenario to the previously explain monopolization is as follows:

- Using a flash loan, initial liquidity is placed into the AMM. The ratio between tokens massively disregards the actual ratio and makes it 1:1000 instead of 1:1. As `MIN_POOL_BALANCE` is hard-coded to 1000, this requires a flash loan of 1000000 tokens B.
- Over the course of the same flash loan, most of the LP tokens are surrendered so that the remaining liquidity is minimal, but not zero. Assume that the remaining liquidity is now 1:1000. The expenses for the attacker so far are, roughly, the flash loan fee, 1000 B tokens and 1 token A.
- Legitimate pool users cannot swap tokens, as none of the swaps would pass the `validate_asset_ratio` asserts. Instead, liquidity needs to be provided before the ratio between the tokens can be corrected. The amount of required liquidity is very large and roughly similar to what the attacker needed in the first place.

As a result, the legitimate users needs to pay the flash loan fees in order to correct the pool ratio.

## Recommendation

No action suggested, as the described scenario does not result in permanent pool monopolization.

## Location

https://github.com/algofiorg/algofi-amm/blob/daaaf4efa6912ac629064490677db6e9cee8be86/contracts/pool.py#L386

Info

# Amplification Factor Ramp-Up Renders Checks Unnecessary

| | | | |
|---|---|---|---|
| **Overall Risk** | Informational | **Finding ID** | NCC-E003791-224 |
| **Impact** | None | **Category** | Other |
| **Exploitability** | None | **Status** | New |

## Impact

An implementation quirk in amplification factor ramp-up code results in a check that becomes redundant as soon as the first ramp-up is triggered.

## Description

Algofi's smart pool implementation allows administrators to ramp-up the amplification factor in a controlled manner. This finding lists two discrepancies between Algofi's and Curve.fi's amplification factor ramp-up mechanism.

As for the first discrepancy, in general, the rate of change in the amplification factor depends on the initial and target amplification factor values provided by the administrator, as well as on the time interval over which the ramp-up is performed. In this context, whenever the amplification factor is fetched, Algofi's implementation performs the following check:

```
    def get_amplification_factor(self):
        return Seq(
            If(self.future_amplification_factor_time.get() == Int(0))
            .Then(self.initial_amplification_factor.get())
            .Else(
                interpolate_amplification_factor(
                    self.initial_amplification_factor.get(),
                    self.future_amplification_factor.get(),
                    self.initial_amplification_factor_time.get(),
                    self.future_amplification_factor_time.get()
                )
            )
        )
```

The highlighted check evaluates to true only if the ramp up event never happened in the past. Starting from the moment a ramp-up is triggered for the first time, the check becomes redundant. At that point, since the check is a part of the `get_amplification_factor` function, it will be called unnecessarily on each swap, each "add liquidity" event, as well as on each flash swap.

The same check does not exist in Curve.fi's implementation. This is due to the fact that Algofi's implementation uses `initial_amplification_factor` as storage for the amplification factor before any ramp-up events were triggered. Curve.fi's implementation, however, keeps it in `future_amplification_factor` (`future_A` in Curve.fi's terminology). This difference allows Curve.fi to return the `future_A` from `interpolate_amplification_factor` regardless of whether the amplification factor ramp-up has been triggered in the past or not.



Another discrepancy is Algofi's omission of validation checks around the rate at which `A` can be modified. In Curve.fi's implementation:

```
def ramp_A(_future_A: uint256, _future_time: uint256):
    assert msg.sender == self.owner  # dev: only owner
    assert block.timestamp >= self.initial_A_time + MIN_RAMP_TIME
    assert _future_time >= block.timestamp + MIN_RAMP_TIME  # dev: insufficient time

    _initial_A: uint256 = self._A()
    assert (_future_A > 0) and (_future_A < MAX_A)
    assert ((_future_A >= _initial_A) and (_future_A <= _initial_A * MAX_A_CHANGE)) or\
           ((_future_A < _initial_A) and (_future_A * MAX_A_CHANGE >= _initial_A))
```

The highlighted checks are omitted in Algofi's implementation. The above limitations likely serve as a layer of protection for users from abrupt changes.

## Recommendation

Consider storing the initial amplification factor in the `future_amplification_factor` in order to be able to remove the check mentioned above.

## Location

https://github.com/algofiorg/algofi-amm/blob/daaaf4efa6912ac629064490677db6e9cee8be86/contracts/stable_pool.py#L83

Info

# Inner Flash Loan Transactions Can Self-Reference Pools

| | | | |
|---|---|---|---|
| **Overall Risk** | Informational | **Finding ID** | NCC-E003791-9LQ |
| **Impact** | Undetermined | **Category** | Other |
| **Exploitability** | Undetermined | **Status** | New |

## Impact

This finding outlines a deviation from the Uniswap V2 implementation. While the deviation appears substantial, NCC Group consultants have not identified any attacks based on the property. Certain edge-case flash loan transactions would fail in Uniswap, but would work in Algofi.

## Description

Uniswap V2 prohibits flash loan transactions from self-referencing the pool the flash loan originates from. This is done via the `lock` custom Solidity modifier. The transaction that is executed as a part of a flash loan cannot operate on the pool that lent out the funds.

This is not the case in Algofi's implementation. A valid flash loan transaction sequence is of the following format:

```
    def on_flash_loan(self):
        # group transaction structure
        # (1) noop transaction -> inner txn
        # (N) payment transaction
        # TODO derive asset id from repay txn to keep app args to 2 for ledger
```

Inner flash loan transactions can be of any type, including transactions that perform swaps, mints or burns on the pool from which the flash loan originates. Right after the flash loan transaction, the actual funds inside the ASA (or Algo funds) are significantly lower than the contract's balance variable. In fact, the pool's balance variable gets incremented further by flash loan fees, even though, technically, the fees have not been delivered at that point. This is because the fees are paid by the borrower in the last transaction.

In regular pool operation, the backing funds will always be higher or equal than reported by the contract's two balance variables. Flash loan inner transactions are unique in the sense that at no other point in the execution of the contract are the backing funds are *lower* than what is recorded by the contract balance variables.

The only consequence of this appears to be that valid edge-case transactions would not work as inner flash loan transactions. For example, inside a flash loan, it is not possible to remove all LP tokens from the pool, since this would result in an attempt to withdraw more liquidity than available at the moment. Interestingly enough, by using excess backing funds in the contract (e.g. by directly funding the ASA and effectively locking the funds forever), such inner transactions can be made to work.

## Recommendation

Document that when pool liquidity is low, flash loan transactions that may work otherwise may not work inside the flash loan context. Re-evaluate whether this property introduces vulnerabilities when the contract is upgraded.

## Location

https://github.com/algofiorg/algofi-amm/blob/daaaf4efa6912ac6290644906 77db6e9cee8be86/contracts/pool.py#L926



Info

# Fragility in Flash Loan Repay Validation

| | | | |
|---|---|---|---|
| **Overall Risk** | Informational | **Finding ID** | NCC-E003791-FWB |
| **Impact** | None | **Category** | Other |
| **Exploitability** | None | **Status** | New |

## Impact

Future contract upgrades must not add support for transaction groups that have payment-to-pool transactions as last transaction in the group. In addition, it is not possible for users to take flash loans from multiple pools atomically.

## Description

The flash loan transaction group is as follows:

```
# FLASH LOAN
# 1 - flash loan
# ...
# FINAL - flash loan repay txn
FLASH_LOAN_IDX = Int(0)
FLASH_LOAN_REPAY_IDX = Global.group_size() - Int(1)
```

Consider a scenario where the initial flash loan transaction is followed by another operation, represented by a sequence of transactions. Suppose that the last transaction in the sequence of transactions needs to be a payment to the pool. In that case, what was supposed to be two different payment transactions will collapse into one transaction and be validated by the flash loan handler. As a consequence, the pool must not have any operations that have a payment-to-pool transaction at the end.

It is also worth noting that performing the above attack by nesting flash loan transactions is prevented by hard-coding the index of the flash loan transaction to zero, as per the `FLASH_LOAN_IDX` field. A side consequence of this strategy is that it appears impossible to atomically perform flash loans from multiple Algofi pools. In addition, if other applications on the Algorand blockchain also hard-code the transaction number to zero, Algofi's flash loans cannot be used with such applications.

## Recommendation

Ensure that when adding new valid transaction groups to the contract, ASA or Algo transactions do not come at the end.

## Location

https://github.com/algofiorg/algofi-amm/blob/daaaf4efa6912ac629064490677db6e9cee8be86/contracts/globals.py#L58

# 3 Additional Observations

**Methods tolerate arbitrary extra arguments.** The Algofi contract does not validate that the number of arguments is exactly what it should be in any of the methods. Each application call remains valid if additional arguments are added into transactions. There does not appear to exist a way to abuse this behavior. Algofi may consider enforcing exact argument lengths to preventatively close issues that may arise in the future.

**Algofi tolerates clawback-enabled ASAs.** Initializing a pool involves opting the pool application into two Algorand Standard Assets (ASAs). ASAs support a wide range of features, including freezing assets and freely moving assets between the accounts. In order to reduce the number of pools operating on questionable tokens, Algofi may consider not opting into clawback and freeze enabled ASAs.

**Algofi transaction groups can be part of larger groups.** Key Algofi transactions are deemed valid only if they come in transaction groups of pre-specified format. For example, a `burn_asset2` validates if it is preceded by one `burn_asset1` transaction and an LP token payment transaction. It is worth noting that transaction group chaining can be established in two ways:

- Full transaction chaining: each transaction in the group validates that all other assumed transactions are present in the group.
- Partial transaction chaining: transaction type B validates that it is preceded by transaction type A and transaction type A validates the presence of all other required transactions in the group. For an example of the latter case, see `on_redeem_pool_asset1`, which validates only that it is preceded by a pool transaction and the pool transaction validates that it is a part of the expected group.

It is worth noting that valid Algofi transaction groups can be part of larger transaction groups. This allows atomic chaining of Algofi operations, which is an important feature and as such, presently, it does not make sense to add more strict validation of transaction group size.

**`update_balances` method name confusion.** There exist two different functions with the same name, but in different scopes. An `update_balances` function is defined in the `on_pool` scope and a different function (which also updates *reserves*, in addition to balances) is defined inside the `on_flash_loan` scope. The latter function appears to be a misnomer since it does not only update the balances, but also the reserves. Change the latter function's name and also prefix each function with `pool` or `flash_loan` to facilitate code understanding.

**Lack of Uniswap V2's sync operation.** Unlike in Uniswap V2, Algofi's contract does not appear to sync the ASA balances with the pool's balances. The ASA balances should never dip below the pool's pair view of the balances. Since there is a number of places where pool and ASA balances are updated, and due to the possibility of rounding errors, it is not immediately obvious that the invariant holds.

**Manager/reserve fee proportion can be set to arbitrary value.** Algofi swaps and flash loans incur fees, which indirectly benefit pool liquidity providers. A percentage of the fees is available to the manager application administrator via the `on_remove_reserves` function. There is currently no upper bound on the manager's fee percentage. It can be set to the full swap and flash loan fee, potentially stripping the pool liquidity providers of any income.

**Arbitrary flash loan fees.** In addition, it is worth noting that the manager contract administrator can set flash loan fees to an arbitrary percentage. The contract manager could arbitrarily increase the flash loan fee and reap the full fee amount.

**Amplification factor scaling discrepancy.** As discussed with the Algofi team, Curve.fi's and Algofi's implementations assume different scales in their corresponding `amp` amplification factors. Curve.fi's `amp` value assumes a value multipled by `n * (n-1)`, where `n` is the number of coins in the pool, whereas Algofi's does not. For `n` strictly greater than 2, Curve.fi's implementation does not follow the formula for `c` described eg. in this blog post. As such, in the `n=2` case, Algofi and Curve.fi implementations are equivalent.

## StableSwap `compute_D` Walkthrough & Notes

This subsection consists of a walkthrough of the `compute_D()` function with commentary.

```
@Subroutine(TealType.uint64)
def compute_D(
        asset1_amount: TealType.uint64,
        asset2_amount: TealType.uint64,
        amplification_param: TealType.uint64,
):
    """
    StableSwap two assets implementation
    D invariant calculation in non-overflowing integer operations
    iteratively
    A * sum(x_i) * n**n + D = A * D * n**n + D**(n+1) / (n**n * prod(x_i))
    Converging solution:
    D[j+1] = (A * n**n * sum(x_i) - D[j]**(n+1) / (n**n prod(x_i))) / (A * n**n - 1)
    The order of assets doesn't matter
    """
```

It's not immediately obvious that the given formula for `D[j+1]` is correct. The derivation of this formula was not verified, however it was confirmed that this matches the Curve.fi implementation: https://github.com/curvefi/curve-contract/blob/master/contracts/pool-templates/base/SwapTemplateBase.vy#L214.

```
    n_coins = Int(2)
    S = asset1_amount + asset2_amount

    D_estimate = ScratchVar(TealType.bytes)
    D_prev = ScratchVar(TealType.bytes)
    D_prod_denom = ScratchVar(TealType.bytes)
    D_prod = ScratchVar(TealType.bytes)
    AnnS = ScratchVar(TealType.bytes)
    i = ScratchVar(TealType.uint64)
```

Note that the number of supported coins is hardcoded at two. Some portions of the calculation assume this number is fixed, whereas other portions are flexible. It was also noted that these values are often converted to bytes prior to use, often multiple times. The value `S` is used twice as `Itob(S)`, and once as an `Int` for comparison purposes. It would slightly reduce the opcodes to store `S = Itob(asset1_amount + asset2_amount)`, as the

later equality operation also works when comparing a byte slice to an `Int`, per https://pyteal.readthedocs.io/en/stable/arithmetic_expression.html.

```
    D_prod_num_calc = BytesMul(D_prev.load(), BytesMul(D_prev.load(), D_prev.load())) #
    ↪ D**(n+1)
    D_prod_denom_calc = BytesMul(
        Itob(Exp(n_coins, n_coins)), BytesMul(Itob(asset2_amount), Itob(asset1_amount)) #
        ↪ nn*(P[x_i])
    )
    D_prod_calc = BytesDiv(D_prod_num_calc, D_prod_denom.load())
```

Here, the value `D_prod` is computed so it can be re-used multiple times. Note that the calculation of `D**(n+1)` is hardcoded as `D*D*D = D**3`, assuming a fixed value of `n=2`. Despite this assumption, the value `n**n` is computed as `Exp(n_coins, n_coins)`, a more general expression. The constant product invariant is computed as `asset2_amount * asset1_amount`.

While the code is correct, the mixture of approaches is unintuitive. It appears that there is no exponent function that operates on bytes, however the `Exp(n_coins, n_coins)` operation could be trivially replaced with its constant result of `4`. It was also observed that the implementation tends to use the standard (overloaded) Python operators for most arithmetic, aside from the use of `Exp()`. PyTeal would support the equivalent expression of `n_coins**n_coins`.

```
    Ann_calc = amplification_param * Exp(n_coins, n_coins)
    AnnS_calc = BytesDiv(BytesMul(Itob(Ann_calc), Itob(S)), Itob(Int(PARAMETER_SCALE_FACTOR)))
```

It was noted that the above computes `Ann = A * n**n`, as expected, but the reference implementation by Curve.fi computes a similar value differently: `Ann: uint256 = _amp * N_COINS`; see https://github.com/curvefi/curve-contract/blob/master/contracts/pool-templates/base/SwapTemplateBase.vy#L225. Line 136 of the same file clarifies:

> `@param _A Amplification coefficient multiplied by n * (n - 1)`

When `n=2`, the computed value of `Ann_calc` will match in both implementations, but will differ for `n_coins > 2`.

```
    D_estimate_num_calc = BytesMul(
        BytesAdd(
            AnnS.load(),
            BytesMul(D_prod.load(), Itob(n_coins)),
        ),
        D_prev.load(),
    )
    D_estimate_denom_calc = BytesAdd(
        BytesDiv(
            BytesMul(Itob(Ann_calc - Int(PARAMETER_SCALE_FACTOR)), D_prev.load()),
            Itob(Int(PARAMETER_SCALE_FACTOR)),
        ),
        BytesMul(D_prod.load(), Itob(n_coins + Int(1))),
    )

    D_estimate_calc = BytesDiv(D_estimate_num_calc, D_estimate_denom_calc)
```

The numerator is computed as `(AnnS + D_prod * n_coins) * D_prev`, which differs from the equation in the function documentation: `(A * n**n * sum(x_i) - D[j]**(n+1) / (n**n prod(x_i)))`. Unifying the notation we have:

- Header: `AnnS - D_prod`
- Implemented: `(AnnS + D_prod * n_coins) * D_prev`

Note that the implemented version matches the Curve.fi numerator when `n_coins = 2`. The denominator is computed as:

- Header: `(Ann - 1)`
- Implemented: `( (Ann - SCALE_FACTOR) * D_prev ) / SCALE_FACTOR + (D_prod * (n_coins + 1))`

Again, this matches the Curve.fi implementation when `n_coins = 2`. These deviations appear to reflect modifications made to prevent integer overflow. It was noted that the Curve.fi implementation operates on `u256` values, whereas the `bytes` type in Teal is up to 64 bytes, or 512 bits. Therefore, the overflow concerns may be less impactful in a Teal implementation. Additional investigation may be needed to confirm this observation.

The remainder of the function performs Newton-Raphson iterations in an attempt to converge on a numerical solution for `D_estimate` based on `D_prev`.

```
    calc = Seq(
        If(S == Int(0)).Then(Return(Int(0))),
        # Store expensive calcs that don't need to be recomputed
        AnnS.store(AnnS_calc),
        D_prod_denom.store(D_prod_denom_calc),
        # First guess
        D_estimate.store(Itob(S)),
        For(i.store(Int(0)), i.load() < Int(255), i.store(i.load() + Int(1))).Do(
            Seq(
                D_prev.store(D_estimate.load()),
                D_prod.store(D_prod_calc),
                D_estimate.store(D_estimate_calc),
                If(BytesGt(D_estimate.load(), D_prev.load()))
                .Then(
                    If(
                        BytesLe(
                            BytesMinus(D_estimate.load(), D_prev.load()), Itob(Int(1))
                        )
                    ).Then(Return(Btoi(D_estimate.load())))
                )
                .Else(
                    If(
                        BytesLe(
                            BytesMinus(D_prev.load(), D_estimate.load()), Itob(Int(1))
                        )
                    ).Then(Return(Btoi(D_estimate.load())))
                ),
            )
        ),
        Assert(i.load() < Int(255)),  # did not converge, throw error
```

```
        Return(Int(0)),  # unreachable code
    )

    return calc
```

**Potential for Optimization**
The value `n_coins**n_coins` is computed 3 times within the function, with the resulting Teal code containing the following 3 times:

```
int 2
int 2
exp
```

Using a constant value of `4` would eliminate these 3 `exp` operations.

Similarly, the value of `S` is computed 3 times, with the resulting code containing three instances of:

```
load 48
load 49
+
```

The resulting value could be stored such that two of the instances are replaced with a single load.

Local variables are stored as `uint64`, whereas the scratch variables are stored as `bytes`, such as:

```
Itob(Exp(n_coins, n_coins)), BytesMul(Itob(asset2_amount), Itob(asset1_amount)) # nn*(P[x_i])

int 2
int 2
exp
itob
load 49
itob
load 48
itob
```

This requires three distinct `itob` operations to evaluate a single expression. Refactoring to avoid redundant conversions would reduce the number of operations.


# 4 Finding Field Definitions

The following sections describe the risk rating and category assigned to issues NCC Group identified.

## Risk Scale

NCC Group uses a composite risk score that takes into account the severity of the risk, application's exposure and user population, technical difficulty of exploitation, and other factors. The risk rating is NCC Group's recommended prioritization for addressing findings. Every organization has a different risk sensitivity, so to some extent these recommendations are more relative than absolute guidelines.

### Overall Risk

Overall risk reflects NCC Group's estimation of the risk that a finding poses to the target system or systems. It takes into account the impact of the finding, the difficulty of exploitation, and any other relevant factors.

| Rating | Description |
|---|---|
| **Critical** | Implies an immediate, easily accessible threat of total compromise. |
| **High** | Implies an immediate threat of system compromise, or an easily accessible threat of large-scale breach. |
| **Medium** | A difficult to exploit threat of large-scale breach, or easy compromise of a small portion of the application. |
| **Low** | Implies a relatively minor threat to the application. |
| **Informational** | No immediate threat to the application. May provide suggestions for application improvement, functional issues with the application, or conditions that could later lead to an exploitable finding. |

### Impact

Impact reflects the effects that successful exploitation has upon the target system or systems. It takes into account potential losses of confidentiality, integrity and availability, as well as potential reputational losses.

| Rating | Description |
|---|---|
| **High** | Attackers can read or modify all data in a system, execute arbitrary code on the system, or escalate their privileges to superuser level. |
| **Medium** | Attackers can read or modify some unauthorized data on a system, deny access to that system, or gain significant internal technical information. |
| **Low** | Attackers can gain small amounts of unauthorized information or slightly degrade system performance. May have a negative public perception of security. |

### Exploitability

Exploitability reflects the ease with which attackers may exploit a finding. It takes into account the level of access required, availability of exploitation information, requirements relating to social engineering, race conditions, brute forcing, etc, and other impediments to exploitation.

| Rating | Description |
|---|---|
| **High** | Attackers can unilaterally exploit the finding without special permissions or significant roadblocks. |



| Rating | Description |
| --- | --- |
| **Medium** | Attackers would need to leverage a third party, gain non-public information, exploit a race condition, already have privileged access, or otherwise overcome moderate hurdles in order to exploit the finding. |
| **Low** | Exploitation requires implausible social engineering, a difficult race condition, guessing difficult-to-guess data, or is otherwise unlikely. |

## Category

NCC Group categorizes findings based on the security area to which those findings belong. This can help organizations identify gaps in secure development, deployment, patching, etc.

| Category Name | Description |
| --- | --- |
| **Access Controls** | Related to authorization of users, and assessment of rights. |
| **Auditing and Logging** | Related to auditing of actions, or logging of problems. |
| **Authentication** | Related to the identification of users. |
| **Configuration** | Related to security configurations of servers, devices, or software. |
| **Cryptography** | Related to mathematical protections for data. |
| **Data Exposure** | Related to unintended exposure of sensitive information. |
| **Data Validation** | Related to improper reliance on the structure or values of data. |
| **Denial of Service** | Related to causing system failure. |
| **Error Reporting** | Related to the reporting of error conditions in a secure fashion. |
| **Patching** | Related to keeping software up to date. |
| **Session Management** | Related to the identification of authenticated users. |
| **Timing** | Related to race conditions, locking, or order of operations. |