Flow Lending v1 smart contract audit report

Christian Schmitz, **Helios**

2025-06-17

revision 1

Contents

1. Audit summary

This is a report of the first audit performed on v1 of the Flow Lending smart contract.

The Flow Lending smart contract codebase has been audited by doing a manual review. The imported libraries, compiler toolchains and unit-tests are out of scope.

The audited version of the protocol is commit **9e23936fe45c2c86293025a60f6f0fcc34b31824** of **https://github.com/flow-lending/flow-lending-smart-contracts**

The audit findings are separated into three categories:

- Security
- Performance
- Maintainability

Per category, the findings are sorted by decreasing importance.

This audit found:

- **15** security findings, of which **3** are critical
- **9** performance findings, all minor or informational
- **5** maintainability findings, all minor or informational

As part of this audit, some additional commits, up-to-and-including commit **5032c534bb54e0b0416fbd361eda60c89b2ca85b**, have been reviewed as well (though not taken into account in the checklists in the appendices).

These additional commits resolve **15/15** security findings, **1/9** performance findings, and **1/5** maintainability findings.

All unresolved findings are either minor or informational, and have been acknowledged by the Flow Lending team to be taken into consideration for v2 of the contract.

> **Disclaimer**: This audit is provided for informational purposes only and does not constitute a warranty or guarantee of the smart contract's security or functionality.

2. Contract description

Flow Lending is a fully collateralized lending protocol.

Lenders deposit liquidity into pools. Pools are configured with a single reserve asset class, but allow multiple collateral asset classes.

Pool orders are batched, and batches are witnessed by the `pool` validator. In fact, most validations are delegated to the `pool` validator.

There are 5 types of pool orders:

1. deposit liquidity into pools (in exchange for pool ownership tokens called cTokens)
2. withdraw liquidity from pools (by burning corresponding cTokens)
3. borrow (by sending collateral into a vault coupled to the pool)
4. repay (by paying sufficient reserve assets into the pool, allowing redeeming the collateral)
5. liquidate (anyone can do this with someone else's loan collateral if the loan becomes undercollateralized)

Each batch transaction can only handle a single order type at a time.

The audited version of the smart contract is somewhat centralized, and gives a lot of power to the Admin actor.

Other important properties:

- Loans must be repayed or liquidated in their entirety
- The Batcher actor can currently send liquidated collateral wherever they please
- Collateral can be added during the lifetime of a loan

The Flow Lending contract uses a version of the Aiken compiler that targets Plutus V3. Importantly, this means that transaction mint values don't contain a redundant 0 lovelace value (in Plutus V2 they do contain a redundant 0 lovelace value).

2.1. Validators

The smart contracts consists of the following validators:

Validator	Description	Delegates to `pool` validator
`batcher_mp`	Witnesses the minting/burning of batcher license NFTs and the creation of batcher license outputs. Batched order executions witnessed by the `pool` validator must spend a valid batcher license UTXO.	no
`ctoken`	Witnesses the minting/burning of pool ownership tokens. Liquidity providers can redeem their fair share of the pool liquidity and profit (excluding accumulated reserves), by burning these tokens.	yes
`order`	Witnesses the spending of order UTXOs. A transaction that spends an order UTXO must either be signed by the order owner, or be witnessed by the `pool` validator.	yes
`pool_auth_token`	Witnesses the minting/burning of pool tokens and the creation of pool and pool info outputs.	no
`pool_info`	Witnesses the spending of pool info UTXOs, which only requires the Admin actor signature.	no
`pool`	Witnesses the spending of pool UTXOs, validating batched deposits, withdrawals, borrowing, repayments and liquidations. Pool UTXOs can be spent at will by the Admin actor.	-
`vault_authtoken`	Witnesses the minting/burning of vault auth tokens, which mark valid vault UTXOs.	yes
`vault`	Witnesses the spending of vault UTXOs. Vault UTXOs contain loan collateral. The owner of the vault UTXO can add collateral at any time. The Admin actor can spend the vault UTXO at will.	yes

2.2. Transactions

The smart contract allows the following transactions:

Transaction	Inputs	Mints/burns	Outputs	Actors
Create pool	any	+1 pool NFT, +1 pool info NFT	pool UTXO, pool info UTXO	Admin
Close pool(s)	pool UTXOs, pool info UTXOs	any	any	Admin
Assign pool batcher	any	+1 batcher license NFT	any	Admin
Create deposit order	any	any	order UTXO	Liquidity provider
Create withdrawal order	any	any	order UTXO	Liquidity provider
Create borrow order	any	any	order UTXO	Borrower
Create repay order	any	any	order UTXO	Borrower
Create liquidate order	any	any	order UTXO	Anyone
Cancel order	order UTXO	any	any	Order owner
Batch deposit orders	pool UTXO, order UTXOs	+x cTokens	pool UTXO	Batcher
Batch withdrawal orders	pool UTXO, order UTXOs	-x cTokens	pool UTXO, return UTXOs	Batcher
Batch borrow orders	pool UTXO, order UTXOs	+x vault auth tokens	pool UTXO, return UTXOs, vault UTXOs	Batcher
Batch repay orders	pool UTXO, order UTXOs, vault UTXOs	-x vault auth tokens	pool UTXO, return UTXOs	Batcher
Batch liquidate orders	pool UTXO, order UTXOs, vault UTXOs	-x vault auth tokens	pool UTXO, return UTXOs	Batcher
Add collateral	vault UTXO	any	vault UTXO	Borrower
Slash loan	vault UTXO	any	any	Admin
Change pool info	pool info UTXO	any	pool info UTXO	Admin

2.3. State

The protocol contract state is formed by the following types of UTXOs:

- batcher license UTXOs
- order UTXOs
- pool UTXOs
- pool info UTXOs
- vault UTXOs

Each UTXO consists of:

- address
- value
- inline datum

Note: Reference scripts in outputs can also be considered part of the contract state, but there is no risk of them making UTXOs unspendable.

Reference scripts only add to the total cost of spending transactions, and there is no upper limit on the transaction fee.

For these reasons this audit doesn't consider the spurious attachment of reference scripts to contract UTXOs.

2.4. Assets

The Flow Lending contract involves minting the following assets:

- batcher license NFT
- cToken (represents ownership of pool liquidity)
- pool NFT
- pool info NFT
- vault auth token (marks valid vault UTXOs)

3. Security findings

S01: the `VAddCollateral` branch of the `vault` validator allows the output containing the collateral to be sent anywhere

Severity: critical

Description:

The `VAddCollateral` branch of the `vault` validator checks all aspects of the vault output state, except its address.

The owner of the vault UTXO can exploit this by sending the vault output to their personal wallet, and thus extracting all collateral.

Recommendation:

Check the address of `own_output` in the `VAddCollateral` branch of the `vault` validator, ensuring it is equal to the `vault` validator address.

Status: RESOLVED (see commit 6dafb9b9497af3607996f2a3612defb05b795e2c)

S02: the `vault` validator requires the vault output to contain exactly the same amount of lovelace as the input

Severity: critical

Description:

Due to the equality check of the vault output value, the lovelace quantity must be identical to the input's lovelace quantity.

In cases where the collateral asset isn't ADA, the vault input will usually contain precisely the minimum amount of lovelace required by the ledger (i.e. the minimum deposit).

This means that large increases of `VaultDatum.collateral_amount` aren't possible because the minimum lovelace deposit amount will increase as well, but the validation doesn't allow changing the lovelace quantity.

This also means that network parameter changes can make vault UTXOs unspendable.

Recommendation:

Use >= instead of == when checking the vault output value, and only allow a distinct collateral asset in addition to ADA (similar to the output value checks in the `pool` validator).

Status: RESOLVED (see commit d000b7c52939c413943db53e8ceff8500534fe79)

S03: the `pool` validator often requires the pool output to contain an exact amount of lovelace

Severity: critical

Description:

This is similar to finding **S02**. UTXOs should be able to contain a flexible amount of lovelace, otherwise they might become unspendable due to datum size increases or network parameter changes.

Recommendation:

Change == operator to >= whenever checking that an output contains a certain amount of lovelace.

Status: RESOLVED (see commit 2e4dc44461d6448f5315b1ca1bde77dab874257f)

S04: vault UTXOs can be spent at will by any repayment or liquidation batch transactions

Severity: major

Description:

The `vault` validator `VSpend` branch delegates all its validations to the `pool` validator.

The `pool` validator however doesn't count the number of vault UTXOs effectively being spent from the vault address.

Recommendation:

In the `ApplyRepay` and `ApplyLiquidate` branches of the `pool` validator, count all spent vault UTXOs, containing vault auth tokens, and ensure that it is equal to the number of orders being batched.

Status: RESOLVED (see commits 20a3c77237a6bfff4748d2e2a55ffd7cf85c67f3 and 5032c534bb54e0b0416fbd361eda60c89b2ca85b)

S05: order UTXOs can be spent at will by any batcher action

Severity: major

Description:

The `order` validator `Apply` redeemer delegates all validations to the `pool` validator, by ensuring that a pool input is spent with the same `pool_id`.

The `pool` validator is given a list of orders (`order_indices` in `PoolRedeemer`), but nothing seems to prevent the Batcher actor who builds a pool spending transactions to leave out orders from that list. So that the Batcher actor can ignore orders, or even extract all value from the orders without fulfilling them.

Recommendation:

Count all spent inputs sitting at the order addresses.

Status: RESOLVED (see commit 60bc2bd80d7e7c0f6c21e1f2948435941c436b70)

S06: the Batcher actor can mint any number of vault auth tokens using `ApplyLiquidate` in the `pool` validator

Severity: major

Description:

The `ApplyLiquidate` branch of the `pool` validator is the only branch (besides `Close`) that doesn't check the minted quantity.

The `vault_authtoken` validator delegates its minting validation to the `pool` validator, thus allowing an unlimited number of vault auth tokens to be minted during a batched liquidation transactions.

Recommendation:

Check the `mint` value exactly like it is checked in `ApplyRepay`.

Status: RESOLVED (see commit 07f450172ba4fd07e851a7b30a6619a788d35add)

S07: the Batcher actor can substitute any other input for an order

Severity: major

Description:

In the `pool` validator, none of the action branches check that the spent orders are unique orders that sit at the order address.

Assuming finding **S05** will be resolved as recommended, a malicious Batcher actor will be able to use arbitrary inputs instead of actual orders, while the `order` validator allows spending real orders because the counts match.

Recommendation:

Check that order inputs are spent from the `order` validator address and aren't spent twice.

Status: RESOLVED (see commit af3a885d405942e262891f194b7ef8f63ed1a9cf)

S08: the `ApplyLiquidate` branch of the `pool` validator allows spending vault UTXOs that don't contain the vault auth token

Severity: major

Description:

In `pool` validator, unlike `ApplyRepay`, `ApplyLiquidate` doesn't check if the vault input being spent contains the vault auth token.

This allows a malicious Batcher actor to use a self-created vault UTXO to fulfill a liquidation order.

Recommendation:

In the `ApplyLiquidate` branch of the `pool` validatior, check that the spent vault UTXO contains the vault auth token.

Status: RESOLVED (see commit 2d70f7e70475e2942b1cef41d0580b957eb34684)

S09: the Batcher actor can set the `start_time` in the vault output datum to any number smaller than `start_valid_time_range`

Severity: minor

Description:

In the `ApplyBorrow` branch of the `pool` validator, the Batcher actor can set the `start_time` in the vault output datum to anything before `start_valid_time_range`.

The older the `start_time`, the more interests must be payed. This is to the disadvantage to the vault output owner.

Recommendation:

Allow the order owner to specify the `start_time` as part of the order datum, and check that the vault output datum `start_time` is equal to the `start_time` value in the order (in addition to checking that `start_time <= start_valid_time_range`).

Status: RESOLVED (see commit 0bb8a4fc7015362195271268c9edf3f73148e6ec)

S10: the `Burn` redeemer of the `pool_auth_token` validator can be abused

Severity: minor

Description:

The Admin actor can use the `pool_auth_token` validator `Burn` redeemer to mint any number of pool NFTs and thus instantiate any number pools.

This is an example of a wrong redeemer exploit, where the redeemer can be abused to trigger a validator code-path that doesn't correspond to the current transaction.

Recommendation:

Check that nothing is minted when burning pool auth tokens (i.e. that all minted quantities are `<= 0`). This is actually mentioned in the contract requirements, but missing from the code-base.

Status: RESOLVED (see commit 923b0c0cd0a3b7afdec7254df3b1a05962b77360)

S11: `pool_auth_token` `Mint` transactions can be created an unlimited number of times

Severity: minor

Description:

A Flow Lending validators are almost all parametrized with pool-specific data, and thus each pool will have unique validator addresses and policies.

This means that being able to mint the pool NFT and pool info NFT more than once doesn't make any sense.

Recommendation:

When minting the pool and pool info NFTs, check that an input is spent with an `OutputReference` that is equal to the pool id.

Status: RESOLVED (see commit 923b0c0cd0a3b7afdec7254df3b1a05962b77360)

S12: `pool_auth_token` mint validation doesn't check `pool_id` in `PoolDatum` of output

Severity: minor

Description:

This is similar to issue S11.

Recommendation:

Check that the `pool_id` of the pool output and the pool info output, is equal to the specified `pool_id` parameter.

Status: RESOLVED (see commit 923b0c0cd0a3b7afdec7254df3b1a05962b77360)

S13: pool outputs can be sent to any address when minted

Severity: informational

Description:

The pool output creation, which is witnessed by the `pool_auth_token` validator, and contains the pool NFT, can be sent to any address.

This makes the pool output datum requirements in the `pool_auth_token` validator completely redundant, as they can be circumvented by the Admin actor before sending the pool output to the actual `pool` validator address.

Recommendation:

Specifying the `pool` validator hash inside the `pool_auth_token` validator is difficult (requires a redesign of the contract), and can be left for v2 of the contract.

In the meantime the `pool` validator hash can be passed in via the redeemer. Then checking that the `pool` output is sent to that address at least provides some protection against erroneous contract deployment.

Status: RESOLVED (see commit f81b5f508cb2e02406766d6aba1d407030f3d184

S14: most pool datum fields can be chosen at will by the Admin actor

Severity: informational

Description:

The outputs of a `pool_auth_token` mint transaction isn't fully witnessed.

This means the Admin actor has the power to choose most pool datum fields, and pool info datum fields, at will.

This is a consequence of how v1 of the contract is designed, but is a good for improvement when designing v2 of the contract.

Recommendation:

Only allow a single deployment per pool id. (Same recommendation as **S11**).

Status: RESOLVED (see commit 923b0c0cd0a3b7afdec7254df3b1a05962b77360)

S15: Admin can do anything with the `Close` redeemer of the `pool` validator

Severity: informational

Description:

The `Close` redeemer branch of the `pool` validator only requires that the spending transaction is signed by the Admin actor.

Using the `Close` redeemer, the Admin actor can do almost anything in the contract, as most other contract validators delegate their validations to the `pool` validator.

Recommendation:

Rename the `Close` redeemer to something else, to make it clear that the Admin actor can do almost anything.

Status: RESOLVED (see commit 539ff17967fd1fc09a421f7608ffb3e5307d872f)

4. Performance findings

P01: duplicate common calculations in the `pool` validator

Severity: minor

Description:

Each of the 5 batcher redeemers in the `pool` validator starts with the same set of calculations.

The `pool` validator is the largest validator of the protocol and is invoked during almost every transaction.

Refactoring this large chunk of common code should significantly decrease the on-chain size of the `pool` validator, and thus decrease the transaction fees.

Recommendation:

Use two nested enums in the Pool redeemer. The first level can be the Apply and Close redeemers, and inside the Apply redeemer the 5 batcher actions can be defined as a nested enum.

By splitting the Pool redeemer like that, it should be straightforward to perform all common batcher calculations before doing action-specific calculations.

Status: ACKNOWLEDGED

P02: redundant order counting in the `ApplyBorrow` and `ApplyRepay` branches of the `pool` validator

Severity: minor

Description:

Assuming finding **S05** will be resolved as recommended, the order count calculated during the order folds in the `ApplyBorrow` and `ApplyRepay` branches can be removed, and the count of inputs at the order address can be reused.

Recommendation:

Don't calculate `count` using the main order fold loop in the `ApplyBorrow` and `ApplyRepay` branches of the `pool` validator.

Status: RESOLVED (see commit 20a3c77237a6bfff4748d2e2a55ffd7cf85c67f3)

P03: many redundant input checks in the `pool` validator

Severity: minor

Description:

Input checks are needed when for example using precalculated indices from the redeemer, or for making sure that the input corresponds to the output.

The `pool` validator however contains many unnecessary input checks.

For example: inside the `ApplyDeposit` branch, each order input is checked to contain some minimal value, but this is not necessary as it is the Batcher actor's responsability to only include orders that contain enough value.

Recommendation:

Remove order input value checks in the `pool` validator at:

- `ApplyDeposit` (lines 428-445)
- `ApplyWithdraw` (lines 588-595)
- `ApplyBorrow` (lines 807-813,819-820,826-831)
- `ApplyRepay` (lines 1051, 1085-1101)
- `ApplyLiquidate` (lines 1267-1291).

Status: ACKNOWLEDGED

P04: pool info output checks are unnecessary as long as the `pool_info` validator only requires the Admin signature

Severity: informational

Description:

Pool info UTXO datums, values, and addresses don't need to be checked in the `pool_auth_token` validator, because that UTXO is fully controlled by the Admin actor anyway as long as the `pool_info` validator only requires the Admin signature.

Recommendation:

Remove validations related to the pool info output in `pool_auth_token` validator if the `pool_info` remains unchanged.

Alternatively: perform at least the same validations on the pool info output as the `pool_auth_token` validator. Additionally, the immutability of the script hashes can be enforced, by checking that the pool info output is sent back to the same address with mostly the same information.

Status: ACKNOWLEDGED

P05: redundant pool redeemer checks in the `ctoken` and `vault_authtoken` validators

Severity: informational

Description:
The pool redeemer checks in the `ctoken` and `vault_authtoken` validators, which delegate all validations to the `pool` validator, are redundant because the `pool` validator only allows minting/burning a single asset in each of the five batching branches (assuming finding **S06** will be resolved as recommended).

Recommendation:
Remove the pool redeemer check from the `ctoken` and `vault_authtoken` validators.

By removing these checks, these validators can be merged into a single `ctoken_and_vault_authtoken` validator, using different asset names for ctokens and vault auth tokens (but having the same policy).

Status: ACKNOWLEDGED

P06: redundant `Close` redeemer handling in `VSpend` branch of `vault` validator

Severity: informational

Description:
The Admin actor already has the right to do anything with the vault output by using the `VSlash` redeemer.

Recommendation:
Remove the `Close -> True` branch when checking the `pool_redeemer` value in the `vault` validator.

Status: ACKNOWLEDGED

P07: redundant lovelace content check of vault output if lovelace isn't used as collateral nor used as reserves

Severity: informational

Description:

In `pool` validator, in the `ApplyBorrow` branch, the vault output is checked to contain at least a certain amount of lovelace even if the lovelace isn't use as collateral nor as liquidity.

Recommendation:

Remove the min lovelace requirement for vault outputs in the `pool` validator `ApplyBorrow` branch, if the collateral asset isn't ADA.

Status: ACKNOWLEDGED

P08: order min lovelace check in `ApplyBorrow` branch of the `pool` validator is redundant for the ADA collateral case

Severity: informational

Description:

This finding is similar to finding **S07**, but concern order inputs instead of vault outputs.

In the `ApplyBorrow` branch of the `pool` validator, the line with `expect in_lovelace >= min_lovelace`, which checks that the order input contains at least a certain amount of lovelace, is redundant for the ADA collateral case.

This is because in the ADA collateral case, the following check is performed as well:

```
expect
    assets.lovelace_of(in.output.value) >= min_lovelace +
o_collateral_amount * collateral_info.collateral_decimals
```

Recommendation:

Move the `expect in_lovelace >= min_lovelace` check into the `False` branch of the subsequent `when` statement.

Status: ACKNOWLEDGED

P09: redundant withdraw order datum field `o_amount`

Severity: informational

Description:

`o_amount` can be extracted from the withdraw order UTXO value, and doesn't need to be part of the UTXO datum as well.

Recommendation:

Extract the `o_amount` quantity directly from the withdraw order input value, instead of indirectly from the datum.

Status: ACKNOWLEDGED

5. Maintainability findings

This is the least important group of findings but still relevant to the audit.

Badly or confusingly structured code makes it more difficult to understand the contract.

M01: `year_in_seconds` is confusing

Severity: minor

Description:

`year_in_seconds` defined in the `utils.ak` file is actually the number of milliseconds per year.

Recommendation:

Rename `year_in_seconds` to `year_in_millis`

Status: RESOLVED (see commit 640d66a1e0b67ebc17542d100aff2ae61473a016)

M02: pool delegation logic duplicated

Severity: informational

Description:

The logic to delegate to the `pool` validator is duplicated in several places (eg. in the `ctoken` and `vault_authtoken` validators).

Recommendation:

Refactor this logic.

Status: ACKNOWLEDGED

M03: unnecessary indentation levels in `pool` validator

Severity: informational

Description:

In the `pool` validator, in each fold over the `order_indices`, the order datum variant is destructured using `when order_datum is ...`, but in that `when` expression only a single order datum type checked at a time.

The extra indentation and nested scope are confusing, because within it validations are done which would make more sense to be part of the outer scope

Recommendation:

Use an `expect` statement to verify the order type and destructure the order datum fields.

Status: RESOLVED (see commit 43588ecb7a7158bfc7740c3d7d23dbc9bda3c572)

M04: the `batcher_mp` validator defines unused staking actions

Severity: informational

Description:

The `batcher_mp` validator can be used as a staking witness for:

- withdrawing ADA staking rewards
- performing actions with the associated staking certificate.

This however doesn't seem to be used anywhere in the contract, and its possible external use is also undocumented.

Recommendation:

Add documentation explaining why and how this is used.

Status: ACKNOWLEDGED

M05: unused `VaultRedeemer` type defined in `types.ak`

Severity: informational

Description:

The `VaultRedeemer` type defined in `types.ak` is unused.

The actually used `VaultRedeemer` type is defined in `vault.ak` instead.

Recommendation:

Get rid of the `VaultRedeemer` type definition in `types.ak`

Status: ACKNOWLEDGED

A. Output state checklist

A.1. Batcher license outputs

Batcher license outputs are spent by batcher transactions. The license deadline is the only contract-related state contained in these outputs.

The license deadline is encoded in the asset name of the batcher license NFT, and is immutable due to the nature of the Cardano blockchain.

A.2. Order outputs

Order output creation is unwitnessed and can be created in whatever state the owner of the order wants.

Order outputs don't form threads, and are destroyed when they are consumed.

The Batcher actor will only include spend well-formed orders during batcher transactions.

Thus, the output state of orders doesn't need to be analyzed here.

A.3. Vault outputs

A.3.1. Creation

Vault output creation is witnessed by the `vault_authtoken.ak` validator, which delegates its validations to the `pool` validator.

Specifically, these validations are done during the `ApplyBorrow` branch of the `pool` validator.

Aspect	**Validated**	**Info**
address	yes	pool.ak:797
value	yes	pool.ak:807-812,821-823,832-840
`VaultDatum.pool_id`	yes	pool.ak:787
`VaultDatum.borrow_id`	yes	pool.ak:788
`VaultDatum.owner_pkh`	yes	pool.ak:789
`VaultDatum.owner_stake_key`	yes	pool.ak:790
`VaultDatum.collateral_amount`	yes	pool.ak:792
`VaultDatum.collateral_asset`	yes	pool.ak:791
`VaultDatum.collateral_decimals`	yes	pool.ak:793-794
`VaultDatum.interest_rate`	yes	pool.ak:801
`VaultDatum.start_time`	**unbounded**	pool.ak:795, see **finding S09**
`VaultDatum.principal`	yes	pool.ak:796

Vault outputs are sent to the `vault` validator address.

A.3.2. Repay, liquidate, or slash

Repaying/liquidating or slashing vault outputs, don't change the contract state.

Thus, these transactions are revelant to the analysis of vault output state.

A.3.3. Add collateral

After vault outputs are created, the only transactions that update the vault output state are those that add collateral.

The `vault` validator witnesses addition of collateral, in its `VAddCollateral` branch.

Aspect	Validated	Info
address	**no**	see **finding S01**
value	yes	vault.ak:212
`VaultDatum.pool_id`	yes	vault.ak:190
`VaultDatum.borrow_id`	yes	vault.ak:192
`VaultDatum.owner_pkh`	yes	vault.ak:193
`VaultDatum.owner_stake_key`	yes	vault.ak:194-195
`VaultDatum.collateral_amount`	yes	vault.ak:203:204
`VaultDatum.collateral_asset`	yes	vault.ak:196-197
`VaultDatum.collateral_decimals`	yes	vault.ak:198-199
`VaultDatum.interest_rate`	yes	vault.ak:200
`VaultDatum.start_time`	yes	vault.ak:202
`VaultDatum.principal`	yes	vault.ak:201

A.4. Pool info outputs

A.4.1. Creation

Witnessed by `pool_auth_token.ak`

Aspect	Validated	Info
address	**no**	
value	yes	pool_auth_token.ak:88-93,96
`PoolInfoDatum.pool_id`	**no**	
`PoolInfoDatum.envelope_amount`	**unbounded**	pool_auth_token.ak:45
`PoolInfoDatum.pool_asset`	**no**	
`PoolInfoDatum .pool_asset_decimals`	**unbounded**	pool_auth_token.ak:46
`PoolInfoDatum.ctoken`	**no**	
`PoolInfoDatum.vault_authtoken`	**no**	
`PoolInfoDatum.batcher_policy_id`	**no**	
`PoolInfoDatum.vault_script_hash`	**no**	
`PoolInfoDatum.pool_stake_key`	**no**	
`PoolInfoDatum .reserve_factor_percentage`	**unbounded**	pool_auth_token.ak:47
`PoolInfoDatum.collateral_infos`	**unbounded**	pool_auth_token.ak:52-61
`CollateralInfo.collateral_asset`	**no**	
`CollateralInfo .collateral_decimals`	**unbounded**	pool_auth_token.ak:58

Aspect	Validated	Info
`CollateralInfo.oracle_nft`	**no**	
`CollateralInfo .liquidation_threshold`	**unbounded**	pool_auth_token.ak:56
`CollateralInfo.max_borrow_ltv`	**unbounded**	pool_auth_token.ak:57
`PoolInfoDatum.kink`	**unbounded**	pool_auth_token.ak:48
`PoolInfoDatum.base_rate`	**unbounded**	pool_auth_token.ak:49
`PoolInfoDatum.slope_low`	**unbounded**	pool_auth_token.ak:50
`PoolInfoDatum.slope_high`	**unbounded**	pool_auth_token.ak:51

A.4.2. Change pool info

Aspect	Validated	Info
address	**no**	
value	**no**	
`PoolInfoDatum.pool_id`	**no**	
`PoolInfoDatum.envelope_amount`	**no**	
`PoolInfoDatum.pool_asset`	**no**	
`PoolInfoDatum.pool_asset_decimals`	**no**	
`PoolInfoDatum.ctoken`	**no**	
`PoolInfoDatum.vault_authtoken`	**no**	
`PoolInfoDatum.batcher_policy_id`	**no**	
`PoolInfoDatum.vault_script_hash`	**no**	
`PoolInfoDatum.pool_stake_key`	**no**	
`PoolInfoDatum.reserve_factor_percentage`	**no**	
`PoolInfoDatum.collateral_infos`	**no**	
`PoolInfoDatum.kink`	**no**	
`PoolInfoDatum.base_rate`	**no**	
`PoolInfoDatum.slope_low`	**no**	
`PoolInfoDatum.slope_high`	**no**	

The pool info UTXO state can currently be changed at will by the Admin actor.

A.5. Pool outputs

Each pool has a pool UTXO containing the pool NFT.

The pool value contains the liquidity asset, at least the envelope amount of lovelace.

The pool UTXO is always sent to the pool validator address, which optionally has a staking part.

A.5.1. Creation

Pool creation is witnessed by the `pool_authtoken.ak` validator. The creation must however be signed by the Admin actor, and the Admin actor can also do whatever they want in the pool.ak validator.

Aspect	Validated	Info
address	**no**	
value	yes	pool_auth_token.ak:94-95,97
`PoolDatum.pool_id`	**no**	
`PoolDatum.total_supplied`	yes	pool_auth_token.ak:65
`PoolDatum.total_borrowed`	yes	pool_auth_token.ak:66
`PoolDatum.reserve`	yes	pool_auth_token.ak:67
`PoolDatum.total_ctoken`	yes	pool_auth_token.ak:68

A.5.2. Batch deposits

Aspect	Validated	Info
address	yes	pool.ak:352
value	yes	pool.ak:362-367,480-500
`PoolDatum.pool_id`	yes	pool.ak:355
`PoolDatum.total_supplied`	yes	pool.ak:475-476
`PoolDatum.total_borrowed`	yes	pool.ak:477
`PoolDatum.reserve`	yes	pool.ak:478
`PoolDatum.total_ctoken`	yes	pool.ak:473-474

A.5.3. Batch withdrawals

Aspect	Validated	Info
address	yes	pool.ak:513
value	yes	pool.ak:525-530,649-670
`PoolDatum.pool_id`	yes	pool.ak:518
`PoolDatum.total_supplied`	yes	pool.ak:644-645
`PoolDatum.total_borrowed`	yes	pool.ak:646
`PoolDatum.reserve`	yes	pool.ak:647
`PoolDatum.total_ctoken`	yes	pool.ak:642-643

A.5.4. Batch borrow

Aspect	Validated	Info
address	yes	pool.ak:684
value	yes	pool.ak:694-699,894-915
`PoolDatum.pool_id`	yes	pool.ak:687
`PoolDatum.total_supplied`	yes	pool.ak:889
`PoolDatum.total_borrowed`	yes	pool.ak:890-891
`PoolDatum.reserve`	yes	pool.ak:892
`PoolDatum.total_ctoken`	yes	pool.ak:888

A.5.5. Batch repay

Aspect	Validated	Info
address	yes	pool.ak:929
value	yes	pool.ak:939-944,1136-1156
`PoolDatum.pool_id`	yes	pool.ak:932
`PoolDatum.total_supplied`	yes	pool.ak:1127
`PoolDatum.total_borrowed`	yes	pool.ak:1131
`PoolDatum.reserve`	yes	pool.ak:1133
`PoolDatum.total_ctoken`	yes	pool.ak:1121

A.5.6. Batch liquidate

Aspect	Validated	Info
address	yes	pool.ak:1170
value	yes	pool.ak:1181-1185,1313-1334
`PoolDatum.pool_id`	yes	pool.ak:1173
`PoolDatum.total_supplied`	yes	pool.ak:1304-1307
`PoolDatum.total_borrowed`	yes	pool.ak:1308-1309
`PoolDatum.reserve`	yes	pool.ak:1310-1311
`PoolDatum.total_ctoken`	yes	pool.ak:1299

A.5.7. Close pool

Aspect	Validated	Info
address	**no**	see **finding S15**
value	**no**	see **finding S15**
`PoolDatum.pool_id`	**no**	see **finding S15**
`PoolDatum.total_supplied`	**no**	see **finding S15**
`PoolDatum.total_borrowed`	**no**	see **finding S15**
`PoolDatum.reserve`	**no**	see **finding S15**
`PoolDatum.total_ctoken`	**no**	see **finding S15**

B. Redeemer checklist

Redeemers contain user input, and validations based on redeemer values can give a false sense of security.

Each redeemer field must be verified not to be exploitable.

B.1. `batcher_mp` validator

B.1.1. `LicenseRedeemer` cases matrix

redeemer\tx	**Mint**	**Burn**
Mint	-	Fails because mint quantity isn't 1
Burn	Fails because mint quantity is positive	-

B.1.2. `LicenseRedeemer` fields checklist

Field	Validated	Info
`Mint.dead_line`	yes	`batcher_mp.ak`:46

B.2. `ctoken` validator

B.2.1. `CTokenRedeemer` fields checklist

Field	Validated	Info
`pool_in_idx`	yes	`ctoken.ak`:50-55

B.3. `order` validator

B.3.1. `OrderRedeemer` cases matrix

redeemer\tx	**Apply**	**Cancel**
Apply	-	Fails because the pool output isn't being spent
Cancel	Fails because the tx isn't signed by the order owner	-

B.3.2. `OrderRedeemer` fields

Field	Validated	Info
`Apply.pool_in_idx`	yes	`order.ak`:63-67
`Cancel.order_in_idx`	yes	`order.ak`:77,81,85,89,93

B.4. `pool_auth_token` validator

B.4.1. `MintRedeemer` cases matrix

redeemer\tx	**Mint**	**Burn**
Mint	-	Fails because two positive quantities aren't minted
Burn	**Succeeds**, see **finding S10**	-

B.4.2. `MintRedeemer` fields

Field	Validated	Info
`Mint.pool_info_out_idx`	yes	`pool_auth_token.ak`:88-93
`Mint.pool_out_idx`	yes	`pool_auth_token.ak`:94-95

B.5. `pool_info` validator

Redeemer isn't used.

B.6. `pool` validator

B.6.1. `PoolRedeemer` cases

The `Close` case doesn't matter here because the Admin actor can do as they please with it, so the Admin actor wouldn't need to use another redeemer. And the Batcher actor can't use the `Close` redeemer because they are unable to sign the transaction as the Admin actor.

The other off-diagonal cases will fail because input order types won't match the expected order type.

B.6.2. `PoolRedeemer` fields

Field	Validated	Info
`ApplyDeposit.own_input_idx`	yes	`pool.ak`:356-361
`ApplyDeposit.own_output_idx`	yes	`pool.ak`:362-367
`ApplyDeposit.pool_info_idx`	yes	`pool.ak`:370-375
`ApplyDeposit.license_idx`	yes	`pool.ak`:383,`utils.ak`:28-32
`ApplyDeposit.order_indices`	**no**	see **finding S05**
`ApplyDeposit.order_indices[:][0]`	**no**	see **finding S07**
`ApplyDeposit.order_indices[:][1]`	yes	`pool.ak`:397,418
`ApplyWithdraw.own_input_idx`	yes	`pool.ak`:519-524
`ApplyWithdraw.own_output_idx`	yes	`pool.ak`:525-530
`ApplyWithdraw.pool_info_idx`	yes	`pool.ak`:533-538
`ApplyWithdraw.license_idx`	yes	`pool.ak`:545-549,`utils.ak`:28-32
`ApplyWithdraw.order_indices`	**no**	see **finding S05**
`ApplyWithdraw.order_indices[:][0]`	**no**	see **finding S07**
`ApplyWithdraw.order_indices[:][1]`	yes	`pool.ak`:560,581
`ApplyBorrow.own_input_idx`	yes	`pool.ak`:688-693
`ApplyBorrow.own_output_idx`	yes	`pool.ak`:694-699
`ApplyBorrow.pool_info_idx`	yes	`pool.ak`:702-707
`ApplyBorrow.license_idx`	yes	`pool.ak`:714-718,`utils.ak`:28-32
`ApplyBorrow.order_indices`	**no**	see **finding S05**

Field	Validated	Info
`ApplyBorrow.order_indices[:][0]` (order input index)	**no**	see **finding S07**
`ApplyBorrow.order_indices[:][1]` (return output index)	yes	`pool.ak`:729,778
`ApplyBorrow.order_indices[:][2]` (vault output index)	yes	`pool.ak`:788,797
`ApplyBorrow.order_indices[:][3]` (oracle ref input index)	yes	`pool.ak`:758-762
`ApplyRepay.own_input_idx`	yes	`pool.ak`:933-938
`ApplyRepay.own_output_idx`	yes	`pool.ak`:939-944
`ApplyRepay.pool_info_idx`	yes	`pool.ak`:947-952
`ApplyRepay.license_idx`	yes	`pool.ak`:959-963,`utils.ak`:28-32
`ApplyRepay.order_indices`	**no**	see **finding S05**
`ApplyRepay.order_indices[:][0]` (order input index)	**no**	see **finding S07**
`ApplyRepay.order_indices[:][1]` (return output index)	yes	`pool.ak`:974,1038
`ApplyRepay.order_indices[:][2]` (vault input index)	yes	`pool.ak`:981-986,1005
`ApplyRepay.order_indices[:][3]` (oracle ref input index)	yes	`pool.ak`:1012-1016
`ApplyLiquidate.own_input_idx`	yes	`pool.ak`:1174-1179
`ApplyLiquidate.own_output_idx`	yes	`pool.ak`:1180-1185
`ApplyLiquidate.pool_info_idx`	yes	`pool.ak`:1188-1193
`ApplyLiquidate.license_idx`	yes	`pool.ak`:1200-1204,`utils.ak`:28-32
`ApplyLiquidate.order_indices`	**no**	see **finding S05**
`ApplyLiquidate.order_indices[:][0]` (order input index)	**no**	see **finding S07**
`ApplyLiquidate.order_indices[:][1]` (vault input index)	**partial**	`pool.ak`:1232,1264; see **finding S08**
`ApplyLiquidate.order_indices[:][2]` (oracle ref input index)	yes	`pool.ak`:1238-1242

B.7. `vault_authtoken` validator

B.7.1. `VaultAuthTokenRedeemer` fields

Field	Validated	Info
`VaultAuthTokenRedeemer.pool_in_idx`	yes	`vault_authtoken.ak`:56-61

B.8. `vault` validator

B.8.1. `VaultRedeemer` cases matrix

redeemer\tx	**`VSpend`**	**`VAddCollateral`**	**`VSlash`**
`VSpend`	-	Fails because the pool output isn't being spent	Might succeeds, but Admin can do anything anyway
`VAddCollateral`	Fails due to conflicting pool validations	-	Might succeed, but Admin can do anything anyway
`VSlash`	Fails because not signed by Admin	Fails because not signed by admin	-

B.8.2. `VaultRedeemer` fields

Field	**Validated**	**Info**
`VSpend.pool_in_idx`	yes	`vault.ak`:169-174
`VAddCollateral.own_input_idx`	yes	`vault.ak`:187
`VAddCollateral.own_output_idx`	yes	`vault.ak`:192
`VAddCollateral.add_amount`	yes	`vault.ak`:205-212

C. Mint scarcity checklist

C.1. pool NFT and pool info NFT

Not scarce, the Admin actor can create as many `pool_auth_token` mint transactions as they want.

See **finding S11**.

C.2. batcher license NFT

Not scarce, the Admin actor can create as many `batcher_mp` mint transactions as they want.

This is by protocol design, and no finding is reported for this issue.

C.3. cTokens

The `ctoken` validator delegates cToken mint/burn validation to the pool validator.

Transaction	Validated	Info
Batch deposits	yes	pool.ak:465-472
Batch withdrawals	yes	pool.ak:634-641
Batch borrow	yes	ctoken.ak:60,pool.ak:880-887
Batch repay	yes	ctoken.ak:60,pool.ak:1113-1119
Batch liquidate	yes	ctoken.ak:60
Close pool	**no**	see **finding S15**

C.4. vault auth tokens

The `vault_authtoken` validator is similar to the `ctoken` validator, and delegates vault auth token mint/burn validation to the pool validator.

Transaction	Validated	File:Line
Batch deposits	yes	vault_authtoken.ak:67
Batch withdrawals	yes	vault_authtoken.ak:67
Batch borrow	yes	pool.ak:880-887
Batch repay	yes	pool.ak:1113-1119
Batch liquidate	**no**	**Finding S06**
Close pool	**no**	see **finding S15**

D. Permissive min UTXO lovelace deposit checklist

The lovelace quantities in the outputs should be allowed to fluctuate slightly so that they can absorb datum size increases and network parameter changes.

If this is not the case, UTXOs can even become unspendable across epoch boundaries.

UTXO type	Action	Permissive lovelace	Info
Batcher license output	Creation	value not checked	`batcher_mp.ak`
Batcher license output	Batcher transaction	value not checked	`pool.ak`
Deposit order return	Batched deposits	yes	`pool.ak`:446-458
Withdraw order return	Batched withdrawals	yes	`pool.ak`:600-626
Borrow order return	Batched borrow	yes	`pool.ak`:847-868
Borrow vault output	Batched borrow	yes	`pool.ak`:821-822,832-833
Repay order return	Batched repay	yes	`pool.ak`:1057-1058,1065-1066
Pool output	Batched deposits	**no**	`pool.ak`:482-484,488-491; see **finding S03**
Pool output	Batched withdrawals	**no**	`pool.ak`:651-653,657-660; see **finding S03**
Pool output	Batched borrow	**no**	`pool.ak`:896-898,902-905; see **finding S03**
Pool output	Batched repay	**no**	`pool.ak`:1137-1139,1143-1146; see **finding S03**
Pool output	Batched liquidate	**no**	`pool.ak`:1314-1316,1320-1323; see **finding S03**
Vault output	Add collateral	**no**	`vault.ak`:212; see **finding S02**

E. Value agency checklist

The Flow Lending contract is at its core a value transfering mechanism through orders.

During the evaluation of value from address to address, the agency of each UTXO's value must be verified.

Output type	**Owner(s) maintains value?**	**Info**
Deposit order	yes	`pool.ak`:446-457
Pool output during `ApplyDeposit`	yes	`pool.ak`:479-502
Withdraw order	yes	`pool.ak`:596-627
Pool output during `ApplyWithdraw`	yes	`pool.ak`:634-641,648-671
Borrow order	yes	`pool.ak`:804-805,843-868
Vault output during `ApplyBorrow`	yes	`pool.ak`:821-823,832-839
Pool output during `ApplyBorrow`	yes	`pool.ak`:893-916
Repay order	yes	`pool.ak`:1057-1058,1065-1072
Pool output during `ApplyRepay`	yes	`pool.ak`:1134-1157
Liquidate order	not yet implemented	
Pool output during `ApplyLiquidate`	yes	`pool.ak`:1311-1333

F. Finding rename mapping

Findings have been communicated with the team and triaged in an iterative process. For each revision of this report the findings have been renamed to reflect their latest severity ranking.

Findings that aren't mentioned in this table have either never been renamed, or have been removed.

Draft	Revision 1
S03	S04
S04	S05
S05	S06
S06	S09
S09	S13
S12	S14
S13	S15
S14	S12
S17	S03
P02	P04
P03	P05
P04	P02
P05	P03
M01	M04
M02	M02
M03	M01
M06	M03