ANASTASIA LABS

Security Audit Report

Fixed Rate Lending

Date 24th May, 2025
Project Danogo
Version 2.2

ANASTASIA LABS

Contents

Disclosure

This document contains proprietary information belonging to Anastasia Labs. Duplication, redistribution, or use, in whole or in part, in any form, requires explicit consent from Anastasia Labs.

Nonetheless, both the customer **Danogo** and Anastasia Labs are authorized to share this document with the public to demonstrate security compliance and transparency regarding the outcomes of the Protocol.

Disclaimer and Scope

A code review represents a snapshot in time, and the findings and recommendations presented in this report reflect the information gathered during the assessment period. It is important to note that any modifications made outside of this timeframe will not be captured in this report.

While diligent efforts have been made to uncover potential vulnerabilities, it is essential to recognize that this assessment may not uncover all potential security issues in the protocol.

It is imperative to understand that the findings and recommendations provided in this audit report should not be construed as investment advice.

Furthermore, it is strongly recommended that projects consider undergoing multiple independent audits and/or participating in bug bounty programs to increase their protocol security.

Please be aware that the scope of this security audit does not extend to the compiler layer, such as the UPLC code generated by the compiler or any areas beyond the audited code.

The scope of the audit did not include additional creation of unit testing or property-based testing of the contracts.

Assessment overview

From **24th January**, **2025** to **15th May**, **2025**, **Danogo** engaged Anastasia Labs to evaluate and conduct a security assessment of its **Fixed Rate Lending** protocol. All code revision was performed following industry best practices.

Phases of code auditing activities include the following:

- **Planning** – Customer goals are gathered.
- **Discovery** – Perform code review to identify potential vulnerabilities, weak areas, and exploits.
- **Attack** – Confirm potential vulnerabilities through testing and perform additional discovery upon new access.
- **Reporting** – Document all found vulnerabilities.

The engineering team has also conducted a comprehensive review of protocol optimization strategies.

Each issue was logged and labeled with its corresponding severity level, making it easier for our audit team to manage and tackle each vulnerability.

Assessment components

Manual revision

Our manual code auditing is focused on a wide range of attack vectors, including but not limited to:

- UTXO Value Size Spam (Token Dust Attack)
- Large Datum or Unbounded Protocol Datum
- EUTXO Concurrency DoS
- Unauthorized Data modification
- Multisig PK Attack
- Infinite Mint
- Incorrect Parameterized Scripts
- Other Redeemer
- Other Token Name
- Arbitrary UTXO Datum
- Unbounded protocol value
- Foreign UTXO tokens
- Double or Multiple satisfaction
- Locked Ada
- Locked non Ada values
- Missing UTXO authentication
- UTXO contention

Executive summary

The Fixed Rate Lending protocol is a decentralized lending and borrowing platform that empowers liquidity providers the flexibility to choose their lending rates and term by locking them in independent liquidity pools. It allows them to earn yields that match or exceed those of non-maturity deposits on Liqwid. This availability of multiple pools enables borrowers to select their desired borrowing terms and rates while securing their loans with the stability of fixed interest.

The protocol heavily relies on two reference inputs authenticated by Protocol Config NFT and Protocol Script NFT to obtain `ProtocolConfigDatum` and `ProtocolScriptDatum` respectively.

- `ProtocolConfigDatum` supplies critical parameters such as a list of tokens eligible for lending or as collateral (including token metadata), protocol fees, and fee address to all validators.
- `ProtocolScriptDatum` serve as a source of authorized script credentials for protocol scripts to ensure correct script dependencies and facilitate upgrades.

It is crucial to note that manipulating these datum values can result in draining or locking of protocol funds and incorrect price calculations, resulting in financial losses. Currently, these NFTs are directly managed by the Danogo team using dedicated validators. They have envisioned to transition from centralized control to a DAO based management model in the future.

Code base

Repository

https://github.com/Danogo2023/Fixed-Rate-Lending.git

Commit

66c31fb4b5fd5b3f7ad3bfb040fba718a02d7cce

Files Audited

Files | SHA256 Checksum
validators/loan.ak bde5d8383cceda34788ae2f1442197ce0555d57fdaff2b26b1fc831edf148e12
validators/oracle_price_calc.ak 1dbe40c385b80bd2e5adc8e8f661b1c26004a5243f11b19b8e3ebf4b27fea131
validators/pool.ak c5cfa1aa7a5038dfd8c1acf8e0f1ec310d6ac11629e617f61b574510ca7f3e4e
lib/loan/create_loan.ak d1b8c5f58008b36b9f1700fd160c2f8639b31d44564eeff22a4d0c4f39ab3766
lib/loan/repay_loan.ak b1e5edfb8275b814d476b232114539ecc8cd49d3d37337e4890458058e1464f7
lib/loan/topup_collateral.ak 7108c4e33a9e8bc9d65f596c667f55ab577933f547e347b5b8451593f680cdb5
lib/pool/create_pool.ak 17794f01863e62ddb528a529591ec6760c448ba4e3f55f24d8ce57a8eafe9ed5
lib/pool/redeem_pool.ak 5a99d4c9fe189a113a5b162db4c25b7c5272169829ea6ff3df57e983e5f617cb

Files | SHA256 Checksum
lib/pool/supply_liqwid.ak cfed91384376e2b475ad3527caf460262e635129568a0dbfe03c5f33b0cf12ee
lib/constants.ak 6895ad3e516f25a6975de285e581d88662be79b1f9a22d6261d5294db42ab365
lib/helpers.ak 3f36fad2e794a36ec2ec36fdb73b2d6820c54ecc3fe35d42b81acca9d8bf37a4
lib/liqwid_types.ak 9a034a04710280d3f2aa52c4b0867ad9e9b8b314e1477ad227738065cf3a63c6
lib/tokens.ak b17ed6685ab4e3db1b617378cec04fed37958e8efd22eb998ac11234d1842a85
lib/types.ak e844277a76d3f510768b6e30385f3adb60951a447386fc56b28c01c346e6f312
lib/utils.ak b7fcc5e9032542e81760fd3f067f74e5468aecea2a53ed7f21e813a5ba65677b

Severity Classification

- **Critical**: This vulnerability has the potential to result in significant financial losses to the protocol. They often enable attackers to directly steal assets from contracts or users, or permanently lock funds within the contract.
- **Major**: Can lead to damage to the user or protocol, although the impact may be restricted to specific functionalities or temporal control. Attackers exploiting major vulnerabilities may cause harmor disrupt certain aspects of the protocol.
- **Medium**: May not directly result in financial losses, but they can temporarily impair the protocol's functionality. Examples include susceptibility to front-running attacks,which can undermine the integrity of transactions.
- **Minor**: Minor vulnerabilities do not typically result in financial losses or significant harm to users or the protocol. The attack vector may be inconsequential or the attacker's incentive to exploit it may be minimal.
- **Informational**: These findings do not pose immediate financial risks. These may include protocol optimizations, code style recommendations, alignment with naming conventions, overall contract design suggestions, and documentation discrepancies between the code and protocol specifications.

Finding severity ratings

The following table defines levels of severity and score range that are used throughout the document to assess vulnerability and risk impact

Level	Severity	Status
5	Critical	0
4	Major	6
3	Medium	1
2	Minor	1
1	Informational	11

Findings

ID-401 Lost Yield

	Level	Severity	Status
	4	Major	Resolved

Description

Pool Spending Validator allows supplying tokens to Liqwid to earn yield on them via `SupplyLiqwid` redeemer. This allows for funds locked in pools to be utilized while they wait to be borrowed. The current implementation allows anybody to either supply or withdraw any pool's funds from Liqwid. A malicious actor can deny pool owners their yield by repeatedly withdrawing their funds from Liqwid and putting it back into the pool.

Recommendation

The act of withdrawing funds from Liqwid and foregoing yield should be allowed to permissioned actors alone. These could either be pool owners or protocol administrators.

Resolution

Resolved in 3a07334d09148c33a82d94f85fcb0c954ee6c6aa

ID-402 Pool DDOS

	Level	Severity	Status
	4	Major	Resolved

Description

Pool Spending Validator allows supplying tokens to Liqwid to earn yield on them via `SupplyLiqwid` redeemer. This allows for funds locked in pools to be utilized while they wait to be borrowed. The current implementation allows anybody to spend the pool UTxOs without any change in its value i.e. without performing any supply/withdraw liquidity action. This happens due to a lack of check ensuring that input UTxO's value is not equal to output UTxO's value while enforcing that their dollar value remains the same (which is crucial as dollar value of the UTxO should not change while supplying or withdrawing liquidity from Liqwid).

lib/pool/supply_liqwid.ak

```aiken
let diff_pool_val_to_origin_token =
    assets.reduce(
      pool_in_val,
      pool_ou_val,
      fn(pid, tn, q, r) { assets.add(r, pid, tn, -q) },
    )
      |> utils.to_origin_token(prices, deposit_supported)

  (diff_pool_val_to_origin_token == assets.zero)?
```

A malicious actor can exploit this vulnerability by repeatedly spending pool UTxOs from the contract and sending it back again. This would halt all borrowing actions from the pools and adversely affect the protocol.

Recommendation

The redeemer action of `SupplyLiqwid` should be allowed to permissioned actors alone. These could either be pool owners or protocol administrators. Additionally, it must be ensured that input UTxO's value is not equal to output UTxO's value, preferrably with a minimum threshold of change being enforced.

Resolution

Resolved in 3a07334d09148c33a82d94f85fcb0c954ee6c6aa

ID-403 Pool Redemption DDOS

	Level	Severity	Status
	4	Major	Resolved

Description

The Pool Spending Validator locks all the pool UTxOs at its address. `RedeemPool` redeemer action allows for the spending of multiple pool inputs in a single transaction allowing for full or partial redemption of pools. The Pool Spending Validator delegates its validation to Pool Staking Validator using the Withdraw Zero Trick to perform transaction level validation of all the pool inputs spent and outputs returned.

Once the pools are eligible for redemption (tx_start > supply_maturity && loans_activated == 0) the current implementation allows any actor to spend these pools and return them back to the validator without any actual redemption (burning of principal or yield tokens) taking place. This happens as a result of two factors:

1. Staking validator allows a transaction without any burning of principal or yield tokens (the actual act of redemption) to enable permissionless recovery of protocol fees.
2. An absence of inequality check on script input and output utxo's values

A malicious actor can leverage this vulnerability to perform DDOS attack by repeatedly spending redeemable pool UTxOs and denying actual pool owners the ability to redeem.

Recommendation

It is recommended to check that every continuing script input's value changes in a pool redemption transaction.

Resolution

Resolved in 3a07334d09148c33a82d94f85fcb0c954ee6c6aa

ID-404 Bad loans

Level	Severity	Status
4	Major	Resolved

Description

```aiken
if tx_start > loan_maturity {
  True
} else {
  let total_collateral_amount =
    zip_collaterals(
      loan_collaterals,
      pool_collaterals,
      loan_token,
      prices,
    )
  total_collateral_amount <= loan_amount
}
```

Whenever the borrowing capacity (`liquidation_threshold * collateral_value` where `0 < liquidation_threshold <= 1`) of a loan's collateral falls below the loan amount it becomes under collateralized. In this scenario, protocol allows the loan to be liquidated permissionlessly by any actor who pays the loan amount and provides them with held collaterals. The liquidator is incentivized to do this only when total collateral value is greater than loan amount. During periods of high volatility the collateral value can fall rapidly and before liquidators can act; their value becomes less than loan amount. That's when a loan turns bad cause the loan amount provided cannot be recovered by selling the collaterals. A debt write-off becomes essential in such circumstances by providing the lender with the collaterals and recoup the loan amount partially.

Current protocol implementation doesn't allow debt write-off resulting in lenders having to pay the loan amount themselves along with protocol fees to obtain the collaterals. This further increases the losses lenders need to bear. Additionally, lenders might not react as quickly as liquidators would in an attempt to preserve collateral value from falling further.

Recommendation

It is recommended to implement debt write-off functionality in the protocol.

Resolution

Resolved in 3a07334d09148c33a82d94f85fcb0c954ee6c6aa

ID-405 Loan DDOS

	Level	Severity	Status
	4	Major	Resolved

Description

The Loan Spending Validator locks all the loan UTxOs at its address. `TopupCollateral` redeemer action allows any actor to spend multiple loan inputs in a single transaction to increase the collateral held in each of those loans. The implementation allows spending these inputs even without increasing the collateral value as long as their value does not decrease.

A malicious actor can leverage this vulnerability to perform DDOS attack by repeatedly spending loan UTxOs and denying actual loan owners or liquidators the ability to top up collateral or repay the loan. This attack when carried out under below circumstances can exacerbate financial losses of protocol users:

- during periods of high market volatility
- loans whose collateral values are closer to liquidation threshold
- loans which are near their maturity

Recommendation

To ensure that script input's collateral value is not equal to corresponding output's collateral value.

Resolution

Resolved in 3a07334d09148c33a82d94f85fcb0c954ee6c6aa

ID-406 Unchecked Liquidation Threshold

Level	Severity	Status
4	Major	Resolved

Description

```aiken
pub type PoolCollaterals =
  Pairs<CollateralToken, LiquidationThreshold>

pub type PoolDatum {
  supply_token: TupleAsset,
  pt_supply: Int,
  yt_supply: Int,
  // timestamp
  supply_maturity: Int,
  collaterals: PoolCollaterals,
  base_interest_rate: Basis,
  gradient: Basis,
  // millisecond
  max_duration: Int,
  loan_activated: Int,
  protocol_fee: Option<Int>,
}

pub type LiquidationThreshold = Basis
pub type Basis = Int

pub fn pool_collaterals_is_valid(
  collaterals: PoolCollaterals,
  deposit_supported: DepositSupported,
  supply_token: TupleAsset,
) {
  when
    list.foldr(
      collaterals,
      [],
      fn(Pair(tk, _), ret) {
```

```
        if tk == supply_token {
          fail @"collaterals can't contain supply token"
        } else if list.has(ret, tk) {contain duplicates
          fail @"collaterals is duplicate"
        } else if pairs.has_key(deposit_supported, tk) {
          [tk, ..ret]
        } else {
          fail @"collaterals is invalid"
        }
      },
    )
  is {
    [] -> True
    _ -> True
  }
}
```

The Pool Minting Policy does not check the liquidation threshold for the collaterals being set in `PoolDatum` at the time of pool creation. The offchain logic can intentionally or unintentionally set arbitrary liquation thresholds in the datum. Since datum is hardly checked or inspected (for non-technical users this is not feasible) by a user while signing a transaction, their provided liquidity would be at great financial risk. With no option to cancel the provided liquidity, their funds are locked till `supply_maturity`. A very high liquidation threshold (>>> 1) would allow a borrower to loan out all the funds by depositing collaterals worth fraction of the loan amount (should satisfy the condition `collateral value * liquidation_threshold > loan amount`). Pratically, stealing the supplier's funds.

Additionally, the current implementation allows setting an empty collateral list which would prevent any borrower from lending out the funds. This would render the supplier being unable to utilize the protocol for earning additional yield.

Recommendation

It is recommended to either reference liquidation thresholds from `ProtocolConfigDatum` or validate that all the thresholds adhere to the range (0, constants.basis]. Also, list of collaterals should never by empty.

Resolution

Resolved in 3a07334d09148c33a82d94f85fcb0c954ee6c6aa

ID-301 Price Inaccuracy Possible

	Level	Severity	Status
	3	Medium	Acknowledged

Description

The `LiqwidOracle` oracle source provides all the price information denominated in USD. The protocol uses this price information along with price pairs having asset backed stable coins pegged to the dollar like USDM, USDC etc. Basically, treating USD and dollar pegged stable coin price the same e.g. ADA/USD == ADA/USDM, which is a reasonable approximation as long as the stable coin maintains it's peg. In the unfortunate circumstance when a stable coin depegs, the price obtained from `LiqwidOracle` cannot be directly used in conjunction with a stable coin as the USD to USDM price conversion won't be 1:1. This would result in incorrect price calculation which can be taken advantage of at the expense of protocol and it's liquidity suppliers.

It is anticipated to use one oracle source in more than one `PricePathConfigs` (e.g. For SNEK/WMT price pair, price paths being 1. LiqwidOracle(SNEK), Orcfax(WMT, USDM) 2. LiqwidOracle(WMT), Orcfax(SNEK, USDM)). This won't prevent incorrect price calculation even when multiple price paths are used as multiple price paths could be affected by a depeg event.

Recommendation

It is recommended to have price monitoring systems tracking stable coin depeg events for those coins which are used in price calculations. Should a depeg event be observed, immediate actions must be taken to halt/modify price paths involving the affected stable coin.

Resolution

Acknowledged

ID-201 Skipped Pool NFT Burning

	Level	Severity	Status
	2	Minor	Resolved

Description

lib/redeem_pool.ak

```aiken
let pool_ou_dt_expected =
  PoolDatum {
    ..pool_in_dt,
    pt_supply: new_pt_supply,
    yt_supply: new_yt_supply,
    protocol_fee: Some(option.or_else(protocol_fee, fee_pool)),
  }
expect and {
    (pool_ou_addr == pool_in_addr)?,
    (pool_ou_dt == pool_ou_dt_expected)?,
  }
```

In a transaction where all the principal tokens (PT) and yield tokens (YT) for a pool are burnt, the Pool NFT should be burnt too. Additionally, the protocol's minimum ADA constant (5 ADA) should be paid out to it as fees. It is possible to bypass burning this NFT and paying min ada fees to protocol by setting zero quantity of PT and YT in pool datum. The protocol can however still burn the nft in subsequent transaction and claim its fees but at the expense of transaction fees, thereby reducing its earnings.

Recommendation

To prevent returning pool utxo to pool validator if both `new_pt_supply` and `new_yt_supply` are zero.

Resolution

Resolved in 3a07334d09148c33a82d94f85fcb0c954ee6c6aa

ID-101 Optimize Integer Serialization

Level	Severity	Status
1	Informational	Resolved

Description

lib/utils.ak

```aiken
pub fn get_nft_name(inputs: List<Input>) -> AssetName {
  expect Some(Input { output_reference: out_ref, .. }) = list.head(inputs)
  outref_hash.blake2b_224(out_ref)
}
```

daken/outref.ak

```aiken
pub fn blake2b_224(
  OutputReference { transaction_id, output_index }: OutputReference,
) -> Blake2b224Hash {
  convert.int_to_digit(output_index)
    |> bytearray.concat(transaction_id, _)
    |> crypto.blake2b_224
}

pub fn blake2b_256(
  OutputReference { transaction_id, output_index }: OutputReference,
) -> Blake2b256Hash {
  convert.int_to_digit(output_index)
    |> bytearray.concat(transaction_id, _)
    |> crypto.blake2b_256
}

pub fn int_to_digit(i: Int) -> ByteArray {
  if i < 0 {
    fail
  } else if i == 0 {
    "0"
  } else {
    do_int_to_digit(
      builtin.quotient_integer(i, 10),
```

```
      builtin.cons_bytearray(builtin.remainder_integer(i, 10) + 48, ""),
    )
  }
}
```

We observed usage of `int_to_digit` function for integer serialization in multiple util functions instead of `builtin.integer_to_bytearray`, which is more efficient for non zero integers. Below are the Aiken CEK machine execution units for comparison:

```aiken
test perf_int_to_digit() {
  let i = convert.int_to_digit(100)
  i == i
}

test perf_integer_to_byetarray() {
  let i = builtin.integer_to_bytearray(False, 0, 100)
  i == i
}
```

function	int	mem	cpu
int_to_digit	0	3803	773734
integer_to_bytearray	0	1502	1576243
int_to_digit	1	5605	1181620
integer_to_bytearray	1	1502	1576243
int_to_digit	10	9313	2330762
integer_to_bytearray	10	1502	1576243
int_to_digit	100	13021	3479904
integer_to_bytearray	100	1502	1576243

Recommendation

To use `builtin.integer_to_bytearray` everywhere instead of `int_to_digit` for integer serialization.

Resolution

Resolved in 3a07334d09148c33a82d94f85fcb0c954ee6c6aa

ID-102 Optimize Prefixes

Level	Severity	Status
1	Informational	Resolved

Description

lib/utils.ak

```aiken
pub fn get_pt_name(pool_id: AssetName) {
  constants.prefix_pt
    |> bytearray.concat(constants.dash)
    |> bytearray.concat(pool_id)
}

pub fn get_yt_name(pool_id: AssetName) {
  constants.prefix_yt
    |> bytearray.concat(constants.dash)
    |> bytearray.concat(pool_id)
}

pub fn get_loan_owner_name(loan_id: AssetName) {
  constants.prefix_loan
    |> bytearray.concat(constants.dash)
    |> bytearray.concat(loan_id)
    |> crypto.blake2b_256
}
```

The step of concatenating prefixes with a dash `-` incurs cost. It can be avoided by appending dash to the prefix constants itself.

Recommendation

To append a dash to `prefix_pt`, `prefix_yt` and `prefix_loan` constants.

Resolution

Resolved in 3a07334d09148c33a82d94f85fcb0c954ee6c6aa

ID-103 Optimize `to_origin_token`

	Level	Severity	Status
	1	Informational	Resolved

Description

lib/utils.ak

```aiken
pub fn to_origin_token(
  val: Value,
  prices: Pairs<(TupleAsset, TupleAsset), PRational>,
  deposit_supported: DepositSupported,
) -> Value {
  pairs.foldr(
    prices,
    val,
    fn(
      (
        (qtoken_pid, qtoken_name) as yield_token,
        (token_pid, token_name) as origin_token,
      ),
      PRational(qtoken_rate_num, qtoken_rate_deno),
      ret,
    ) {
      if is_valid_yield_token(yield_token, origin_token, deposit_supported) {
        let qtoken_qty = assets.quantity_of(ret, qtoken_pid, qtoken_name)
        assets.add(ret, qtoken_pid, qtoken_name, -qtoken_qty)
          |> assets.add(
              token_pid,
              token_name,
              calc_origin_token(qtoken_qty, qtoken_rate_num, qtoken_rate_deno),
             )
      } else {
        ret
      }
    },
  )
```

```
}
```

If `qtoken_qty` is zero, following value updates are meaningless leading to inefficiencies.

Recommendation

To skip the redundant computation where `qtoken_qty == 0`.

```aiken
pub fn to_origin_token(
  val: Value,
  prices: Pairs<(TupleAsset, TupleAsset), PRational>,
  deposit_supported: DepositSupported,
) -> Value {
  pairs.foldr(
    prices,
    val,
    fn(
      (
        (qtoken_pid, qtoken_name) as yield_token,
        (token_pid, token_name) as origin_token,
      ),
      PRational(qtoken_rate_num, qtoken_rate_deno),
      ret,
    ) {
      if is_valid_yield_token(yield_token, origin_token, deposit_supported) {
        let qtoken_qty = assets.quantity_of(ret, qtoken_pid, qtoken_name)
        if(qtoken_qty == 0) {
          ret
        }
        else {
          assets.add(ret, qtoken_pid, qtoken_name, -qtoken_qty)
          |> assets.add(
              token_pid,
              token_name,
              calc_origin_token(qtoken_qty, qtoken_rate_num, qtoken_rate_deno),
            )
        }
      } else {
        ret
```

```
        }
      },
    )
  }
```

Resolution

Resolved in 3a07334d09148c33a82d94f85fcb0c954ee6c6aa

ID-104 Optimize Value Difference Calculation

Level	Severity	Status
1	Informational	Resolved

Description

The protocol relies extensively on the difference in value of returned script outputs to spent inputs for validations. The difference in value is calculated using the below method:

```aiken
let diff_pool =
    assets.reduce(
      pool_in_val,
      pool_ou_val,
      fn(pid, tn, q, ret) { assets.add(ret, pid, tn, -q) },
    )
```

This implementation is efficient only if `pool_in_val` contains two distinct assets at most. In situations where `pool_in_val` contains more assets an alternate implemention proves to more efficient.

```aiken
let diff_pool =
    pool_in_val
      |> assets.negate
      |> assets.merge(pool_ou_val, _)
```

Below is a comparison of their execution units using Aiken's CEK machine:

function	no. of distinct assets in input	mem	cpu
original_impl	1	108_034	31_581_594
new_impl	1	150_584	43_732_234
original_impl	2	161_776	46_489_427
new_impl	2	163_977	47_961_317
original_impl	3	226_162	64_668_360
new_impl	3	188_014	55_461_500
original_impl	4	274_191	78_365_498
new_impl	4	217_629	64_638_458

Recommendation

To choose the appropriate difference calculation based on the expected number of assets in input value.

Resolution

Resolved in 3a07334d09148c33a82d94f85fcb0c954ee6c6aa

ID-105 Optimize Fetching Redeemer

	Level	Severity	Status
	1	Informational	Resolved

Description

lib/utils.ak

```aiken
pub fn require_mint_redeemer(
  redeemers: Pairs<ScriptPurpose, Redeemer>,
  pid: PolicyId,
  rdmr_expected: Data,
) {
  list.has(pairs.get_all(redeemers, Mint(pid)), rdmr_expected)
}

pub fn require_mint_one_redeemer(
  redeemers: Pairs<ScriptPurpose, Redeemer>,
  pid: PolicyId,
  rdmr_expected: Data,
) {
  expect [one_rdmr] = pairs.get_all(redeemers, Mint(pid))
  (one_rdmr == rdmr_expected)?
}

pub fn require_withdraw_zero(
  redeemers: Pairs<ScriptPurpose, Redeemer>,
  skh: ScriptHash,
  rdmr_expected: Data,
) {
  list.has(pairs.get_all(redeemers, Withdraw(Script(skh))), rdmr_expected)
}
```

The transaction context provided to a validator by the node includes `redeemers` which contains an association list of `ScriptPurpose` to `Redeemer`. The node allows only unique `ScriptPurpose`s in the list associated with a redeemer. The validation performed by above utils to check that each `ScriptPurpose` is present just once, is unnecessary and expensive as it involves traversing the entire list.

Recommendation

To use `pairs.get_first` instead of `pairs.get_all` to match with the expected redeemer.

Resolution

Resolved in 3a07334d09148c33a82d94f85fcb0c954ee6c6aa

ID-106 Optimize `pool_collaterals_is_valid`

	Level	Severity	Status
	1	Informational	Acknowledged

Description

lib/utils.ak

```aiken
pub fn pool_collaterals_is_valid(
  collaterals: PoolCollaterals,
  deposit_supported: DepositSupported,
  supply_token: TupleAsset,
) {
  when
    list.foldr(
      collaterals,
      [],
      fn(Pair(tk, _), ret) {
        if tk == supply_token {
          fail @"collaterals can't contain supply token"
        } else if list.has(ret, tk) {
          fail @"collaterals is duplicate"
        } else if pairs.has_key(deposit_supported, tk) {
          [tk, ..ret]
        } else {
          fail @"collaterals is invalid"
        }
      },
    )
  is {
    [] -> True
    _ -> True
  }
}
```

To validate absence of duplicates in collaterals, they are being iterated over multiple times resulting in higher execution units.

Recommendation

It is recommended to have the collaterals list lexicograhically sorted on `CollateralToken` which can make it more efficient as seen below.

```aiken
pub fn pool_collaterals_is_valid_1(
  collaterals: PoolCollaterals,
  deposit_supported: DepositSupported,
  supply_token: TupleAsset,
) {
  expect [x, ..xs] = collaterals
  expect _ =
    list.foldl(
      xs,
      x.1st,
      fn(Pair((pid, tn) as collateral, _), (prev_pid, prev_tn)) {
        let pid_order = bytearray.compare(pid, prev_pid)
        if collateral == supply_token {
          fail @"collaterals can't contain supply token"
        } else if (pid_order == Less || (pid_order == Equal && bytearray.compare(tn, prev_tn) != Greater)) {
          fail @"collaterals order is invalid"
        } else if pairs.has_key(deposit_supported, collateral) {
          collateral
        } else {
          fail @"collaterals is invalid"
        }
      },
    )
  True
}

test perf_pool_collaterals_is_valid() {
  let protocol_config = fx.base_protocol_config()
  let collaterals = [Pair(fx.djed, 1000), Pair(fx.iusd, 1000), Pair(fx.snek, 1000), Pair(fx.usdc, 1000)]
  expect pool_collaterals_is_valid(collaterals, protocol_config.deposit_supported, fx.ada)
}

test perf_pool_collaterals_is_valid_1() {
```

```
  let protocol_config = fx.base_protocol_config()
  let collaterals = [Pair(fx.djed, 1000), Pair(fx.iusd, 1000), Pair(fx.snek, 1000), Pair(fx.usdc, 1000)]
  expect pool_collaterals_is_valid_1(collaterals, protocol_config.deposit_supported, fx.ada)
}

┍━ utils ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━
    │ PASS [mem: 127785, cpu: 67216339] perf_pool_collaterals_is_valid
    │ PASS [mem: 120931, cpu: 61962264] perf_pool_collaterals_is_valid_1
    ┕━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 2 tests | 2 passed | 0 failed
```

Resolution

Acknowledged

ID-107 Incorrect Variable Name

	Level	Severity	Status
	1	Informational	Resolved

Description

validator/oracle_price_calc.ak

```aiken
validator oracle_price_calc(
  OracleGlobalConfig {
    liqwid_qtoken_rate_idx,
    liqwid_interest_rate_idx,
    liqwid_compound_rate_idx,
    danogo_float_borrow_apy_idx,
    danogo_float_dtoken_rate_idx,
    liqwid_oracle_validity_nft,
    liqwid_oracle_token_idx,
    liqwid_oracle_price_idx,
    orcfax_fact_statement_pointer_nft,
  }: OracleGlobalConfig,
  oracle_source_nft: TupleAsset,
  oracle_path_nft: TupleAsset,
)
```

Since `liqwid_oracle_validity_nft` and `oracle_path_nft` tokens each have a quantity greater than one, they do not classify as NFTs.

Recommendation

To rename them as `liqwid_oracle_validity_ft` and `oracle_path_ft`.

Resolution

Resolved in 3a07334d09148c33a82d94f85fcb0c954ee6c6aa

ID-108 Optimize Conditional Expression

Level	Severity	Status
1	Informational	Resolved

Description

lib/utils.{get_ref_data_and_remain_refs}

```aiken
if and {
  fs_skh != #"",
  ref_addr.payment_credential == Script(fs_skh),
 } {
  // Handle Orcfax feed
  expect FsDatum(FsStatement { feed_id, .. } as feed_value, _) = d
  expect tokens.contains_nft(ref_val, (fs_skh, constants.fs_name))
  ([Pair(feed_id, feed_value), ..r1], r2, r3, r5, r6)
 } else if tokens.contains_nft(ref_val, oracle_source_nft) {
  expect d: OracleSourceDatum = d
  (r1, r2, r3, Some(d), r6)
 } else if tokens.contains_nft(ref_val, oracle_path_nft) {
  expect OraclePathDatum { supply_token, price_paths }: OraclePathDatum =
   d
  (r1, r2, r3, r5, [Pair(supply_token, price_paths), ..r6])
 } else if tokens.contains_nft(ref_val, liqwid_oracle_validity_nft) {
  // Handle Liqwid Oracle
  expect [
   //
   asset, price,
  ] =
   get_data_liqwid_by_idx(
    d,
    [liqwid_oracle_token_idx, liqwid_oracle_price_idx],
   )
  expect price: TRational = price
  expect Some(asset): Option<(PolicyId, AssetName)> = asset
  (r1, [Pair(asset, price), ..r2], r3, r5, r6)
 } else {
  (r1, r2, [ref_input, ..r3], r5, r6)
```

```
}
```

While iterating over the list of reference inputs multiple conditional expressions are evaluated to catergorize that reference. It is inefficient to place a conditional expression like `tokens.contains_nft(ref_val, oracle_source_nft)` which stands to be true only once, before other conditionals which can be true multiple times.

Recommendation

By placing the `tokens.contains_nft(ref_val, oracle_source_nft)` check in the end before the `else` branch, the evaluation can be optimized.

Resolution

Resolved in 3a07334d09148c33a82d94f85fcb0c954ee6c6aa

ID-109 Simplify Loan Revenue Calculation

Level	Severity	Status
1	Informational	Resolved

Description

lib/repay_loan.ak

```aiken
let loan_in_val_negate_to_origin_token =
  loan_in_val
    |> assets.add(loan_pid, loan_nft_name, -1)
    |> assets.add(assets.ada_policy_id, assets.ada_asset_name, -env.min_ada)
    |> assets.negate()
    |> utils.to_origin_token(prices, deposit_supported)

// loan_revenue_total_negate = -total loan_in_val (without loan token and min ada) - total loan amount + total loan_fee + total loan collaterals updated
let loan_revenue_total_negate =
  pairs.foldr(
    loan_collaterals,
    loan_in_val_negate_to_origin_token
      |> assets.add(
          supply_token_pid,
          supply_token_name,
          -loan_amount + loan_fee,
        ),
    fn((pid, name), qty_collateral, ret) {
      let qty_in_loan = assets.quantity_of(ret, pid, name)
      assets.add(ret, pid, name, -math.max(qty_in_loan, -qty_collateral))
    },
  )
```

Since the new business requirement is to return interest earned on collaterals to borrower instead of supplier, above loan revenue calculation can be greatly simplified.

Recommendation

To simplify it as below:

```
let loan_revenue_total_negate= assets.from_asset(supply_token_pid, supply_token_name, -loan_amount + loan_fee)
```

Aiken

Resolution

Resolved in 3a07334d09148c33a82d94f85fcb0c954ee6c6aa

ID-110 Redundant Check

	Level	Severity	Status
	1	Informational	Resolved

Description

In `create_loan.ak`, it already validated that `(loan_maturity > tx_start)?` making the check `expect (loan_duration > 0)?` redundant since `let loan_duration = loan_maturity - tx_start`

Recommendation

To remove the redundant check.

Resolution

Resolved in 3a07334d09148c33a82d94f85fcb0c954ee6c6aa

ID-111 Optimize Input and Output Search

Level	Severity	Status
1	Informational	Resolved

Description

It is computationally expensive to search for an input or output containing a particular NFT by iterating over all inputs/outputs and checking their values. We found instances of such inefficient traversals at the following locations:

1. In `create_loan.ak` at line 44 `utils.get_pool_in_by_pool_id`
2. In `create_pool.ak` at line 57 `utils.get_pool_ou_by_pool_id`
3. `utils.get_script_and_config_protocol_datum` used at multiple places to find the script and config reference inputs
4. `transaction.find_input(inputs, out_ref)` and `get_pool_in_by_out_ref`
5. In `topup_collateral.ak` at line 30 `get_loan_ou_info`
6. In `supply_liqwid.ak` at line 26 to fetch `pool_ou_lookup`
7. In `oracle_price_calc.ak` at line 41 `utils.get_maybe_ref_datum_by_nft`

Recommendation

To provide input/output indices via redeemers in order to locate the utxo and then check for the appropriate NFT in the value only once, eliminating multiple costly searches on `Value`. For point 4, transaction builders like Lucid Evolution facilitate convenient input indexing via an abstraction called `RedeemerBuilder`.

Resolution

Resolved in 3a07334d09148c33a82d94f85fcb0c954ee6c6aa

Post Audit Findings

Following vulnerabilities were discovered while reviewing the fixes and enhancements made to the protocol.

ID-R501 Pool Drain

Level	Severity	Status
5	Critical	Resolved

Description

```aiken
// validators/pool.ak
CreateLoan { .. } -> {
  expect Some(p_script_idx) = p_script_idx
  expect ProtocolScriptDatum { pool_skh, loan_skh } =
    utils.get_input_datum_by_idx(
      reference_inputs,
      p_script_nft,
      p_script_idx,
    )
  and {
    (pk == Script(pool_skh))?,
    utils.require_mint_redeemer(redeemers, loan_skh, indexers)?,
  }
}
```

In case of loan creation, just checking for a match between `RedeemerIndexer` type of Pool Spending Validator and Loan Minting Policy is not sufficient. As it allows for different pool ids being set in `LendingAction{pool_in_idx, ..}` while having same incorrect `indexers`. This allows for multiple pools being spent in a transaction during a loan creation with just one pool input undergoing validation. This allows remaining pools to be stolen.

Recommendation

To check that Pool Spending Validator and Loan Minting Policy receive the exact redeemer.

Resolution

Resolved in d3d2bc068e775945f9638972bdc2a5ec5008758a

ID-R401 Incorrect Price Calculation

Level	Severity	Status
4	Major	Resolved

Description

```
// lib/helpers.ak

pub fn calc_price_with_deviation_threshold(
  prices: List<PRational>,
  price_deviation_threshold: Basis,
) -> PRational {
  when prices is {
    [] -> fail @"prices is empty"
    [init_price, ..remain_prices] ->
      when remain_prices is {
        [] -> init_price
        _ -> {
          //
          let PRational { numerator: average_num, denominator: average_den } =
            arithmetic_mean(prices)
          //
          let tolerance_num = average_num * price_deviation_threshold
          let tolerance_den = constants.basis * average_den
          let upper_bound =
            PRational(
              average_num * tolerance_den + tolerance_num * average_den,
              average_den * tolerance_den,
            )
          let lower_bound =
            PRational(
              average_num * tolerance_den - tolerance_num * average_den,
              average_den * tolerance_den,
            )
          list.foldr(
            remain_prices,
            init_price,
```

```
            fn(x, prev_x) {
              let compared_upper_bound = compare_with(x, <=, upper_bound)
              let compared_lower_bound = compare_with(x, >=, lower_bound)
              if compared_lower_bound && compared_upper_bound {
                // get min
                // prev_x < x
                if compare_with(prev_x, <, x) {
                  prev_x
                } else {
                  // prev_x >= x
                  x
                }
              } else {
                fail @"prices is invalid"
              }
            },
          )
        }
      }
  }
}
```

The helper function `calc_price_with_deviation_threshold` forgets to check whether `init_price` falls within the price tolerance range when `remain_prices != []`. This allows price to be incorrectly set to `init_price` should it be the lowest price.

Recommendation

Add a check to confirm `init_price` falls within price tolerance range.

Resolution

Resolved in d3d2bc068e775945f9638972bdc2a5ec5008758a

ID-R301 Missing Price Expiration Check

Level	Severity	Status
3	Medium	Resolved

Description

Price feed information obtained from Indigo and Djed don't undergo expiration check resulting in usage of outdated prices.

Recommendation

To implement check on expiry of price feeds for both the oracles.

Resolution

Resolved in d3d2bc068e775945f9638972bdc2a5ec5008758a

ID-R302 Unbounded `PoolCollaterals` List

	Level	Severity	Status
	3	Medium	Acknowledged

Description

The number of unique collaterals present in `PoolCollaterals: List<TupleAsset>` field of `PoolDatum` is currently unbounded. This could result in unrelated tokens being added to datum by offchain library without being noticed by the user. This increases transaction execution costs whenever the pool is spent.

Recommendation

To have an upper bound on the number of collaterals that can be present.

Resolution

Acknowledged by employing offchain measures like limit on collateral selection through protocol's user interface, disclaimer at user facing technical documentation and cautionary note of this limitation in the form of internal library documentation or code comments.

ID-R303 Different Liqwid Price Versions

Level	Severity	Status
3	Medium	Resolved

Description

Liqwid currently provides price feed information for an asset via version 1 and 2 datum structures. Should both the price versions be present for an asset in a transaction with differing values, the protocol won't be able to differentiate the versions and allow usage of one price over the other based on the order of Liqwid Oracle UTxO appearance. This could technically result in incorrect price calculation.

Recommendation

To allow just one version of price feed for an asset in a transaction by checking for duplicate token price entries in the list of Liqwid Oracle feeds.

Resolution

Resolved, as per Danogo team's verification of on-chain Liqwid Oracle data, different price versions cannot exist for the same asset at any point of time.