ANASTASIA LABS

Security Audit Report

Date: October 11, 2024
Project: PBG Decentralized Vault Portfolios
Version 1.0

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 PBG Capital 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 July 10th, 2024 to October 4th, 2024, PBG Capital engaged Anastasia Labs to evaluate and conduct a security assessment of its PBG Decentralized Vault Portfolios 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 **current financial system** suffers from inefficiencies, high costs, and a lack of transparency. These issues are particularly prominent in the **asset management industry**, where investors encounter limited access to asset classes, reliance on custodians, and opaque portfolio management processes. This increases risks and reduces control.

PBG addresses these issues through **Decentralized Vault Portfolios (DVPs)**—blockchain-powered, non-custodial digital asset portfolios that prioritize transparency and security. Key features of the DVP protocol include:

1. Tokenization of Shares

Each DVP represents a tokenized portfolio, where each token corresponds to a share of the vault's assets. The total portfolio value is regularly updated via Oracle price feeds. Tokens are minted or burned as investors enter or exit the portfolio, with the on-chain token price reflecting the ratio of the portfolio's total value to the number of tokens in circulation.

2. Fee Structure

- **Mint Fee**: Charged upon entry to support protocol operation costs, typically ranging from 0.1% to 2%.
- **Burn Fee**: Charged upon withdrawal to support protocol operation costs, this fee also ranges between 0.1% and 2%.
- **Management Fee**: Charged daily to maintain infrastructure, this fee is applied through proportional token dilution, influencing the token price over time.
- **Success Fee**: A performance-based fee that incentivizes positive returns, deducted only if the portfolio surpasses a set benchmark. This fee is tracked on-chain with NFT vouchers to ensure fair reimbursement of success fee dilution at the end of each cycle.

3. Non-Custodial Vault

Assets within a DVP are securely held in a **vault smart contract**. The asset manager is restricted to executing value-for-value swaps, with Oracle price feeds ensuring each swap is either neutral or beneficial to the vault. User funds are thus safeguarded by enforcing strict operational boundaries on the asset manager.

4. Large Investment Universe

DVPs can accommodate a broad array of assets, with on-chain counters tracking quantities. These counters can be added or removed via governance actions, and investors can enter or exit using any combination of assets within the vault's supported asset universe.

5. On-Chain Governance

All smart contract parameters can only be modified through governance actions, accompanied by a delay to allow investors time to withdraw if desired. Governance actions are witnessed by a governance delegate, which can be any on-chain logic, ranging from a multi-signature script to a decentralized autonomous organization (DAO) contract.

6. AML/CFT Compliance

Investors interact with the vault via orders, and only AML and CFT compliant orders are processed. Irregular orders are ignored to prevent contamination of the vault. Investors retain the option to cancel pending orders at any time.

Conclusion

The **PBG Decentralized Vault Portfolio (DVP)** protocol mirrors the functionalities of traditional asset management funds, providing transparent asset management and fee structures within a non-custodial, blockchain framework. Smart contract parameters are modifiable through a balanced governance model, while compliant operations are maintained through a secure order-based system.

Code base

Repository

https://github.com/PBGToken/validators

Commit

cb76208b23b97dfd13515b3da2f42a93dada6b83

Files audited

SHA256 Checksum	Files
e066300cfd2d9e7fb8d49d159107447a793af7de8dd67d05f489ecf4182b10d9	src/assets_validator.hl
79e16e92e183cf4cb73c60cf939250c42f201ed9c8825b270f0f31e43aa9c401	src/benchmark_delegate.hl
7489c075592d18e323c054b052e7ac288c52ba41202aabe81d18cfb7b6d7bbab	src/burn_order_validator.hl
4af98a9d99a76122c20d15b7b7856f1c809ed81d57cf0c2c182446f3f8b6e507	src/config_validator.hl
643bf55e9305a1a6fe1bb79f4681301098eff5e422cfd9e690b370a217058622	src/fund_policy.hl
b7087bd932898579b78b3b13510542898fc685032e4b7d69fa9ca999f71e17e8	src/governance_delegate.hl
f8b6eb3d3670c9248f8d7690623fb5073b321f70500e1831f50029013f539d42	src/metadata_validator.hl
c2b013da2ba442fe311542abc89fe697f3af67d6e79f83fcda0035adec30c673	src/mint_order_validator.hl
4baeeca9b0844a3bb7aa155652da2893fa5dcf9d9d12d5dcdbd7fced0a7fe265	src/oracle_delegate.hl
36d71a27d0edc3b6b113a24a70af3e429406a0bb7c79fc371273def02af28152	src/portfolio_validator.hl
404581a3168155e0eefa0a1cefa4427a08fb8e4cca6c4f6c9cfa3ff0946393b4	src/price_validator.hl
a493def266ffbb3fb3657d3bf72c29a46bf0857c4eca44c78e6bc616f7372181	src/reimbursement_validator.hl
87f757ae26cdc3b1a32927eaec9239d27ac257fe229731232f0e4fbc47136bd1	src/supply_validator.hl
5f398fb1c7f90d66757f17590350536f6bb2895906874d530dfa61413855a825	src/voucher_validator.hl

SHA256 Checksum	Files
165fecb50e506973a2a788dc7c48adf692 21faf9ba239133d2c5b0a9fba006b7	src/lib/Addresses.hl
d6ea1273488d9cace73734cf28652fd45 03024a097f371b0b3c04db2fabc8fa9	src/lib/Asset.hl
f4bd54b4820ae501572be05532889d4d 4f036498bb8578a6ff3a6f31023453b1	src/lib/AssetGroup.hl
389a817a8c1b4ee553ac77dc4db2d2f95 91113101694646b96ec89c203efb0f8	src/lib/AssetPtr.hl
787845800aa775a23ffb2d3c1c72683ad bd5f9549d38ab61f4c623fea9629d17	src/lib/BurnOrder.hl
4a7a8671370186d82d8b0424aa74c922 d70a512b33e3f5f5e9ad90d2984b46f9	src/lib/Config.hl
0f996ec797660f704def998ea66b53304 d64c5092e18e12b64040a62a26ba26c	src/lib/Metadata.hl
1cb346c7d3862ad3bd0842604c35d19b 46d125f0e778b5e17494f266a0216148	src/lib/MintOrder.hl
8cedaf878fbb30808ee6be9e386a0ae668 192a59c11d15d22a38f3a9964c47df	src/lib/Portfolio.hl
64c66b227a3f7e6b02d6a0e854eb13582 49f8700daf6f14fcdeedec40a6f50c9	src/lib/Price.hl
2e59511c1dd55aaf412b6ad7d060f6911f b0b018cc3b4b1da01923d0bd7b1ab8	src/lib/Reimbursement.hl
b2fcc8fc211cb1c6be70878a2d3eb7f0d1 81318f9d98c2883e43235d58211c8c	src/lib/SuccessFee.hl
39d3dccde26761b00fee11dfbe8db50bfa e5a88d1ac5fa289e6da5651d829675	src/lib/Supply.hl
bd901f28ddee701ce83d4915b5ca77c18 bca4aa14e930c429c74d165940f3349	src/lib/TokenNames.hl
23fe3a35c6ae95461e007cc9c4f0d39989 c3aad8ce72d504c09d91d995baf0d4	src/lib/Tokens.hl
c415d2b7a347df007baa30313851f99cc 7314a86ec2a1110f5660df41645908f	src/lib/Vault.hl
596ecf965f06e3eac81a5089c1c43aa92e 42f369ed9fa1bbf7cef7ebd8fd6785	src/lib/Voucher.hl

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 harm or 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	Findings
5	Critical	5
4	Major	7
3	Medium	1
2	Minor	4
1	Informational	8

Findings

ID-501 Unauthorized Asset Additions

	Level	Severity	Status
	5	Critical	Resolved

Description

The Portfolio Validator's `validate_move_assets` (`MoveAssets` action/redeemer), does not confirm the number of inputs being spent from Assets validator equals the number of outputs returned to it. This allows one or more asset NFT to be claimed by a malicious agent in a move assets transaction. This NFT can later be returned to Assets Validator with new unauthorized assets with arbitrary price and count. Potentially leading to DVP token price manipulation at agent's will.

The below on-chain state describes the conditions under which this attack is possible:

Listing 1: src/portfolio_validator.hl

```
// Portfolio datum
Portfolio{
  n_groups: 3,
  reduction: Idle
}

// Asset Group datums
const assetGroup1 = AssetGroup{
  assets: [assetA, assetB, assetC]
}
// Note: Max size for assets array is three
const assetGroup2 = AssetGroup{
  // assetF removed after addition of it via correct
  //governance action. Leaving one asset entry available in datum
  assets: [assetD, assetE]
}
const assetGroup3 = AssetGroup{
  assets: [assetG]
}

// MoveAssets changes the datum as follows:
const assetGroup1 = AssetGroup{
  assets: [assetA, assetB, assetC]
}
const assetGroup2 = AssetGroup{
  assets: [assetD, assetE, assetG]
}
// Note: Since all the assets are present in the new datum and
// no new assets are added (yet), the validation passes. The agent
// claims the "asset 3" NFT and then sends it back to Asset Validator
```

```
    // with arbitrary prices and counts
    const assetGroup3 = AssetGroup{
      assets: [assetH, assetI, assetJ]
    }
```

Recommendation

The number of inputs being spent from Assets validator should equal the number of outputs returned to it, each having the asset NFT. Additionally, the datums of all outputs must be checked to conform to `AssetGroup`.

Resolution

Resolved in 0b08946c6d988330598e2ae0817ad47178883b09

ID-502 Unbounded Voucher Claim

	Level	Severity	Status
	5	Critical	Resolved

Description

The total number of tokens contained in the vouchers is not checked during the minting of DVP token. Additionally, `mint_order_validator` just has a lower bound for the amount of tokens in each voucher. This allows the agent to set extremely high token amount in voucher(s) for his own mint orders. These vouchers can be later used to claim all the funds locked in Reimbursement UTxO before any reimbursement can be provided to legitimate investors (as the agent himself does the reimbursement processing). Thereby denying success fee reimbursement to majority/all of the entitled users.

Recommendation

To check that total number of tokens contained in vouchers does not exceed the total number of DVP token minted.

Resolution

Resolved in 0b08946c6d988330598e2ae0817ad47178883b09

ID-503 Incorrect Reimbursement Calculation

Level	Severity	Status
5	Critical	Resolved

Description

The Reimbursement Validator calculates voucher alpha incorrectly. It calculates it as `voucher_price/start_price` instead of `end_price/voucher_price`. This leads to incorrect `voucher_phi_alpha_ratio` which is used for obtaining expected reimbursement:

Listing 2: src/reimbursement_validator.hl

```
expected_reimbursed_tokens = (voucher.tokens*(main_phi_alpha_ratio
    - voucher_phi_alpha_ratio)).floor().bound_min(0)
```

Additionally, since agent is allowed to mint vouchers when `voucher_price < start_price`, voucher alpha in this case turns out to be less than one. This results in `voucher_phi_alpha_ratio == 0` which allows agent to obtain maximum amount of reimbursement possible for his vouchers. The agent can potentially claim the entire Reimbursement UTxO amount if he has significant number of vouchers (which can be easily created when the DVP token price has fallen below `start_price`), denying reimbursement to all the deserving users.

Recommendation

To correct voucher alpha calculation to `end_price/voucher_price`.

Resolution

Resolved in 0b08946c6d988330598e2ae0817ad47178883b09

ID-504 Price Manipulation

Level	Severity	Status
5	Critical	Resolved

Description

The Price Validator misses to check whether the Supply input referenced has the same tick as that of `PortfolioReduction`'s `Reducing` constructor (the tick of Supply input itself when reduction started). This allows agent to use a more recent Supply input (an updated tick) for calculation of DVP token price.

Formula for calculation DVP token price: `price = total_value_of_assets/number_of_tokens`

Note: number_of_tokens is obtained from `Supply` datum while total_value_of_assets from `PortfolioReductionMode::TotalAssetValue`.

An agent can hence easily manipulate the price by changing the supply of DVP tokens using mint/burn user orders.

Recommendation

To check that `supply.tick == portfolio.start_tick` in Price Validator.

Resolution

Resolved in 0b08946c6d988330598e2ae0817ad47178883b09

ID-505 Locked Assets Group

Level	Severity	Status
5	Critical	Resolved

Description

The Supply Validator does not ensure that only Asset Token ("assets <group_id>") is present in output returned to Assets Validator apart from ADA. If more than one native tokens are sent by the agent either intentionally or erroneously, the UTxO is permanently locked in Assets Validator. Because Assets Validator relies on having a single native token in the UTxO being spent to determine the policy of supply or portfolio token to complete its validation. Upon finding multiple tokens the validation will always fail with an error.

Listing 3: src/lib/Tokens.hl

```
func indirect_policy() -> MintingPolicyHash {
    input = get_current_input();

    // ignores ADA
    input.value.get_singleton_asset_class().mph
}

const policy: MintingPolicyHash = current_script.switch{
    fund_policy           => direct_policy,
    mint_order_validator  => direct_policy,
    burn_order_validator  => direct_policy,
    supply_validator      => indirect_policy(),
    // The Asset Validator gets the required policy
    assets_validator      => indirect_policy(),
    portfolio_validator   => indirect_policy(),
    price_validator       => indirect_policy(),
    reimbursement_validator => {
        input = get_current_input();
        input.value.get_singleton_asset_class().mph
    },
    voucher_validator     => indirect_policy(),
    config_validator      => indirect_policy(),
    metadata_validator    => indirect_policy(),
    oracle_delegate       => direct_policy,
    benchmark_delegate    => direct_policy,
    governance_delegate   => direct_policy
}
```

Similarly, in Portfolio Validator's `validate_move_assets` the use of `Tokens::contains_only_any_assets` leads to different policy tokens being locked in

the Asset Validator output due to the helper method not checking assets belonging to all policies.

Listing 4: src/lib/Tokens.hl

```
    func contains_only_any_assets[V: Valuable](v: V) -> Bool {
        // presence of other policy ids is not checked
        tokens = v.value.get_policy(policy);

        if (tokens.length != 1) {
            false
        } else {
            (token_name, qty) = tokens.head;

            if (qty != 1) {
                error("expected only 1 assets token")
            } else {
                TokenNames::parse_assets(token_name).switch{
                    Some => true,
                    None => false
                }
            }
        }
    }
```

The Asset UTxOs are needed to be spent for critical protocol actions like minting/burning DVP tokens or swapping out portfolio tokens. If they are locked permanently, the entire protocol is bricked.

Recommendation

To assert that only one native token (assets token) is present in the output returned to asset validator.

Listing 5: src/lib/Vault.hl

```
    func diff_counted(d_lovelace: Int, expected_tick: Int) -> Value {
        m: Map[AssetClass]Int = tx.inputs.fold(
            (m: Map[AssetClass]Int, input: TxInput) -> {
                if (input.address == Addresses::assets) {
                ...

                    id = TokenNames::parse_assets(token_name).unwrap();
                    group_asset_class = Tokens::assets(id);

                    output = tx.outputs.find((output: TxOutput) -> {
                        output.address == Addresses::assets
                        && output.value.get_singleton_asset_class()
                        // Instead of
```

```
                    // output.value.get_safe(group_asset_class) == 1
                    == group_asset_class
                });
        ...
```

Resolution

Resolved in 0b08946c6d988330598e2ae0817ad47178883b09

ID-401 Missing Success Fee Validation

Level	Severity	Status
4	Major	Resolved

Description

The Supply Validator does not check that `success_fee: SuccessFee` field of `Reimbursement` is set to `config.fees.success_fee.fee: SuccessFee` in `validate_reward_success`; the configuration used to calculate the success fee, upon which the user reimbursement actually depends. Without this check, the agent can initialize `success_fee` to any value. A higher than actual success fee robs users of their reimbursement while a lower one denies the protocol of its earnings.

Recommendation

To validate that `reimbursement.success_fee == config.fees.success_fee.fee`

Resolution

Resolved in 0b08946c6d988330598e2ae0817ad47178883b09

ID-402 Reimbursement Multiple Satisfaction

Level	Severity	Status
4	Major	Resolved

Description

The Reimbursement Validator does not enforce the following conditions:

- To allow sResolved in 0b08946c6d988330598e2ae0817ad47178883b09 of just one Reimbursement UTxO in a transaction.
- The vouchers being burnt have the same period_id as for which the Reimbursement UTxO belongs.

Additionally, it relies on `voucher_validator.hl` to check that transaction spends Reimbursement UTxO having the same period_id as that of voucher. However, this validation does allows vouchers with different period_ids to be spent as long as the transaction spends Reimbursement UTxOs for all the period_ids.

The above allows a dishonest agent to acquire DVP tokens from many Reimbursement UTxOs via Multiple Satisfaction Attack. The vouchers need to be selected such that its possible to either burn all the Reimbursement NFTs (when `n_vouchers_burned >= reim.n_remaining_vouchers` holds true, that particular Reimbursement NFT can be burnt and UTxO's holdings claimed) or have just one of them returned to the validator with correct balance.

While it can be presumed that reimbursement validator will not have more than one Reimbursement UTxO at any given point in time, given the protocol's success fee earning is locked till all the reimbursements are carried out. This presumption is challenged when the success fee period is of relatively shorter durations (period <<< 1 year). This happens to be the case currently, as the protocol is initiated with two weeks as success fee period.

Recommendation

By enforcing that all vouchers being burnt have the same period_id as for which the Reimbursement UTxO belongs in `validate_burned_vouchers`.

Resolution

Resolved in 0b08946c6d988330598e2ae0817ad47178883b09

ID-403 Invalid Voucher NFT

Level	Severity	Status
4	Major	Resolved

Description

Listing 6: src/supply_validator.hl

```
// CIP-68 Voucher NFT
const voucher_infix: ByteArray = "voucher ".encode_utf8()
const voucher_ref_prefix: ByteArray = Cip67::reference_token_label
    + voucher_infix
const voucher_nft_prefix: ByteArray = Cip67::user_nft_label
    + voucher_infix

const voucher_id_ref_token_name = voucher_ref_prefix
    + id.show().encode_utf8()
const voucher_id_nft_token_name = voucher_nft_prefix
    + id.show().encode_utf8()
```

The `Supply` datum maintains the `last_voucher_id::Int` field which keeps track of the last voucher id minted. However, upon success fee claim by the agent, `last_voucher_id` is again set to 0, implying that for a new success period, the voucher minting once again begins from id 1. Since, success fee period id is not included in voucher token name (its maintained in the datum of UTxO with reference token), this leads to voucher token names being duplicated thereby rendering them FTs instead of NFTs.

The most prominent issue as a result of above surfaces in the below scenario: At Voucher Validator - voucher_ref_token_1 (datum: period_id = 2, return_address = user_2_address)

At User 1 Wallet - voucher_nft_token_1

At User 2 Wallet - voucher_nft_token_1

Note: User 1 got voucher_nft_token_1 when he minted DVP token during success period_id 1. Its corresponding voucher_ref_token_1 (datum: period_id = 1, return_address = user_1_address) was burnt during reimbursement process after the success period ended.

In addition to being reimbursed for his voucher, User 1 can use the voucher belonging to User 2 by placing a burn order of his DVP token along with his voucher_nft_token_1. He receives a discount on success fee because of the voucher; with both voucher_ref_token_1 and voucher_nft_token_1 getting burnt in the process. This robs User 2 of his voucher.

Recommendation

To not reset the `last_voucher_id` but instead, persist it's value during every success fee claim.

Resolution

Resolved in 0b08946c6d988330598e2ae0817ad47178883b09

ID-404 Lost Management Fee

Level	Severity	Status
4	Major	Resolved

Description

The protocol intends to collect management fees on a daily basis currently. Once the management fee is taken, the `management_fee_timestamp::Time` of `Supply` datum is updated such that `supply_out.management_fee_timestamp >= tx1.time_range.end`. The next management fee can only be claimed after a day's time from `supply_out.management_fee_timestamp` such that

```
supply_out.management_fee_timestamp <= tx2.time_range.start
- config.fees.management_fee.period
```

This methodology introduces a small time difference between the intended duration in which management fee must be taken (24 hours) versus the actual duration (24 hours + the validity range of the transaction). While this difference is small, it gets accumulated over the year.

Management Fee Loss Estimate (Best Case): Assuming a reasonable tx validity range (to cover cases of heavy chain load): 10 minutes After 144 successive managment fee claims, time lost: 144 * 10 minutes = 24 * 60 mintues = 1 day After 288 successive managment fee claims, time lost: 288 * 10 minutes = 2 * 24 * 60 mintues = 2 days

So in effect 2 days worth of management fees (at 0.011% per day) would be lost in a year. It must be noted that, this is when the `tx.time_range.start` is exactly `supply_in.management_fee_timestamp + config.fees.management_fee.period`, `supply_out.management_fee_timestamp = tx.time_range.end` and every submitted transaction is always confirmed on chain. Carrying this out meticulously would make the offchain logic more complex too as the exact timestamp to begin the transaction from will keep changing. Not to mention, any time delay in taking the fee after it was allowed would contribute to days for which management fee cannot be claimed, potentially leading to substantial number of days without management fee. This reduces the actual management fee earned by `number_of_days_lost * 0.011%`.

Recommendation

The recommendation is to make the time keeping of management fee somewhat similar to that of success fee via these conditions:

Listing 7: src/portfolio_validator.hl

```
// In fn main
else if (supply1.management_fee_timestamp != supply0.management_fee_timestamp
            || supply0.management_fee_timestamp
```

```
            + config.fees.management_fee.period < tx.time_range.end
) {
    validate_reward_management(supply0, supply1, D)
}

// In fn validate_reward_management
&& supply1.management_fee_timestamp == supply0.management_fee_timestamp
     + config.fees.management_fee.period
&& supply0.management_fee_timestamp < tx.time_range.start
&& supply1.management_fee_timestamp < tx.time_range.end
// already enforced
&& tx.time_range.end - tx.time_range.start < Duration::DAY
```

Note: This allows management fee to be taken once, any time during the period, even when it just started. Additionally, it enforces the agent to take management fees as opposed to skipping it earlier.

Resolution

Resolved in 0b08946c6d988330598e2ae0817ad47178883b09

ID-405 Reimbursement Failure

	Level	Severity	Status
	4	Major	Resolved

Description

Reimbursement Validator requires voucher reference nfts to be burnt in the transaction for reimbursement and success fee claim to happen. The Fund Policy, which is the minting policy for all the protocol tokens including voucher reference nfts, has not provisioned minting or burning when reimbursement token is spent in the transaction. This results in permanent locking of success fees and user reimbursements in the reimbursement validator.

Recommendation

To allow Fund Policy to burn only voucher reference tokens and/or reimbursement token when reimbursement token is spent in a transaction.

Resolution

Resolved in 0b08946c6d988330598e2ae0817ad47178883b09

ID-406 Cannot Provide Vouchers

Level	Severity	Status
4	Major	Resolved

Description

Upon mint order fulfillment, the `return_address` needs to be provided the minted DVP token along with voucher (if the current benchmark price is greater than start of the year benchmark price).

Listing 8: src/lib/MintOrder.hl

```
    func voucher_id(self) -> Int {
        // check returned voucher
        (voucher_nft_name, qty) =
        self.diff().get_policy(Tokens::policy)
            .find((token_name: ByteArray, _)
            -> {
          TokenNames::has_voucher_nft_prefix(token_name)
      });
      // qty will always be negative
      assert(qty >= 1, "expected at least one token");

      TokenNames::parse_voucher_nft(voucher_nft_name).unwrap()
    }

    func diff(self) -> Value {
        input: TxInput = current_script.switch{
            mint_order_validator => get_current_input(),
            else => error("unexpected")
        };

        return = self.find_return();

        input.value - return.value
    }
```

Obtaining the `(voucher_nft_name, qty)` from `self.diff()` will always result in negative `qty` as difference is calculated as `input.value - return.value`. This will always prevent voucher nft being sent to user leading to failed validation in Mint Order Validator. Therefore, no mint orders can be fulfilled leading to protocol halting.

Recommendation

A check on negated quantity `(-1)*qty` should be done instead.

Resolution

Resolved in 0b08946c6d988330598e2ae0817ad47178883b09

ID-407 Unnecessary Config Spend

Level	Severity	Status
4	Major	Resolved

Description

Almost all the protocol actions require transactions signed by a trusted agent (available in `Config`). Current implementation requires sResolved in 0b08946c6d988330598e2ae0817ad47178883b09 of config UTxO to fetch the agent information rather than simply referencing the UTxO.

Listing 9: src/lib/Config.hl

```
func signed_by_agent(
    agent: PubKeyHash = Config::find_input().agent
) -> Bool {
    tx.is_signed_by(agent)
}

func find_input() -> Config {
    input = current_script.switch{
        config_validator => get_current_input(),
        else => tx.inputs.find((input: TxInput) -> {
            input.address == Addresses::config
        })
    };

    assert(Tokens::contains_config(input),
        "doesn't contain the config token");

    input.datum.inline.as[Config]
}
```

This can lead to protocol halting as config UTxO cannot be spent without valid config changes made with the approval of governance delegate. Additionally, there is an update delay of two weeks for every config change to be applied. This allows the agent to take any action only once in two weeks.

Listing 10: src/lib/Config.hl

```
func witnessed_by_oracle(
    oracle: StakingValidatorHash = Config::find_input().oracle
) -> Bool {
    tx.withdrawals.any((scred: StakingCredential, _) -> Bool {
        scred.switch{
            Hash{h} => {
                h.switch{
```

```
                    Validator{svh} => svh == oracle,
                    else => false
                }
            },
            else => false
        }
    })
}
```

Similarly, oracle witness during price updates also requires config spend which prevents frequent price update of asset tokens and DVP token.

Recommendation

To allow config UTxO to be referenced too, instead of just sResolved in 0b08946c6d988330598e2ae0817ad47178883b09 for authenticating the agent or oracle witness.

Resolution

Resolved in 0b08946c6d988330598e2ae0817ad47178883b09

ID-301 Token Dust

	Level	Severity	Status
	3	Medium	Resolved

Description

Values of all UTxOs sent to a script address must have an upper bound for their size, and the upper bound should be low enough to not prevent consumption of the UTxO as an input in a future transaction. If this isnt́ taken care of, a script UTxO can be subject to being filled with many random tokens which can increase the transaction fees of subsequent transactions. It can also make the script UTxO unspendable in cases where either new token(s) need to be added to it or script execution budgets are exhausted while finding the required token in the value.

This common vulnerability is known as "Token Dust Attack." We found this attack vector at the following places:

- Returned mint/burn order output can contain many useless (`voucher.price < supply.start_price`) voucher nft tokens which are not checked for (facilitated by `self.diff().delete_policy(Tokens::policy)`). An agent can mint all the voucher nfts in a previous transaction to carry out this attack later.
- The reimbursement UTxO sent to userś `return_address` is not checked for dust tokens.
- `contains_only_any_assets`, other policies are not checked for. While it's usage is fine for validating inputs, its insufficient for outputs.
- `AssetPtr::resolve_output`, an unused function yet stated here for completeness.

Recommendation

To fail validation upon finding unnecessary tokens in output values.

Resolution

Resolved in 0b08946c6d988330598e2ae0817ad47178883b09

ID-201 Locked Portfolio Reduction

	Level	Severity	Status
	2	Minor	Resolved

Description

The transaction of addition/removal of an asset class requires the portfolio reduction state to be in `Reducing` mode instead of `Idle`. This is not necessary as the Config datum has been updated to contain the on-chain proof of existence or non-existence of an asset. A `Reducing` mode prevents updation of prices or movement of assets till the update delay is finished (two weeks currently) which stalls the protocol. The alternative would be to reset the reduction to `Idle` and then recompute the reduction before the update is to be applied, which is expensive and nonoptimal.

Listing 11: src/portfolio_validator.hl

```
    func validate_add_asset_class(config0: Config, config_is_spent: Bool,
        portfolio0: Portfolio, _portfolio1: Portfolio) -> Bool {
        (id, group0) = AssetGroup::find_single_input();
        group1 = AssetGroup::find_output(id);
        // needs reduction proof to be present
        DoesNotExist{asset_class} = portfolio0.get_reduction_result();
        AddingAssetClass{expected_asset_class}
            = config0.state.get_proposal();

        group1.assets == group0.assets.append(Asset::new(asset_class))
        && group1.is_not_overfull()
        && asset_class == expected_asset_class
        && config_is_spent
        && Tokens::nothing_minted() // x01 duplicate check
    }

    func validate_remove_asset_class(config0: Config, config_is_spent: Bool,
        portfolio0: Portfolio, _portfolio1: Portfolio) -> Bool {
        (id, group0) = AssetGroup::find_single_input();
        group1 = AssetGroup::find_output(id);
        // needs reduction proof to be present
        Exists{asset_class, found} = portfolio0.get_reduction_result();
        RemovingAssetClass{expected_asset_class}
            = config0.state.get_proposal();

        asset = group0.assets.filter((asset: Asset) -> {
            asset.asset_class == asset_class
        }).get_singleton();

        found
```

```
        && asset.count == 0
        && group1.assets == group0.assets.filter((asset: Asset)
            -> {asset.asset_class != asset_class})
        && asset_class == expected_asset_class
        && config_is_spent
        && Tokens::nothing_minted()
    }
```

Recommendation

To obtain the asset class to be added/removed from Config datum and not rely on `PortfolioReductionMode` of Portfolio.

Resolution

Resolved in 0b08946c6d988330598e2ae0817ad47178883b09

ID-202 Mandatory Reimbursement Return

	Level	Severity	Status
	2	Minor	Resolved

Description

The current Reimbursement Validator logic demands a reimbursement UTxO to be sent to `return_address` even when the actual reimbursement is zero or negligible, instead of avoiding it altogether. This results in higher transaction costs and additional minimum ADA costs for the protocol.

Recommendation

A lower threshold for reimbursement token can be decided below which reimbursement need not be provided.

Resolution

Resolved in 0b08946c6d988330598e2ae0817ad47178883b09

ID-203 Restricted Only-ADA Output

Level	Severity	Status
2	Minor	Resolved

Description

It is intended that Vault also locks ADA in its own separate UTxO (without any CNT), the below condition will prevent it from happening.

Listing 12: src/lib/Vault.hl

```
    func diff() -> Value {
        addr = Addresses::vault;
        ...
        in = tx.outputs.fold((prev: Value, output: TxOutput) -> {
            if (
                output.address == addr
                && output.datum.inline.as[ByteArray] == VAULT_DATUM
                // this check will fail for UTxO with only lovelaces
                && output.value.delete_lovelace().flatten().length == 1
            ) {
                prev + output.value
            } else {
                prev
            }
        }, Value::ZERO);

        in - out
    }
```

Recommendation

To modify the check to `output.value.delete_lovelace().flatten().length <= 1`

Resolution

Resolved in 0b08946c6d988330598e2ae0817ad47178883b09

ID-204 Voucher Overcompensation

Level	Severity	Status
2	Minor	Resolved

Description

Listing 13: src/burn_order_validator.hl

```
    delta_alpha = calc_provisional_success_fee(
        price,
        order.diff(),
        n_burn
    );

    n_expected = deduct_burn_fee(n_burn - delta_alpha);
```

It is possible to incur a negative provisional success fee if large number of vouchers are provided in the burn order when `delta_alpha < 0`. This would warrant additional value to be provided to the user (> n_burn) which would not be allowed by Supply Validator's limit on value extracted from vault. Leading to failed processing of burn order.

Recommendation

To impose a lower bound of zero on `delta_alpha` value.

Resolution

Resolved in 0b08946c6d988330598e2ae0817ad47178883b09

ID-101 Asset Count Tick Redundant

	Level	Severity	Status
	1	Informational	Resolved

Description

`Asset` object contains `count_tick` to keep track of update to its count. From its current usage (in `sum_total_asset_value` invoked by `validate_start_reduction` and `validate_continue_reduction`), `asset.count_tick`'s utility can be eliminated entirely by just relying on tick from supply reference input directly. The supply validator assures the tick in the Supply datum always has the largest value (`supply.tick >= asset.count_tick` for all assets).

Recommendation

Omit `count_tick` field from `Asset` and remove it's usages by relying on tick from `Supply` datum.

Resolution

Resolved in 0b08946c6d988330598e2ae0817ad47178883b09

ID-102 Inefficient Fund Policy Action Ordering

Level	Severity	Status
1	Informational	Resolved

Description

Listing 14: src/fund_policy.hl

```
func main(args: MixedArgs) -> Bool {
    args.switch{
        SResolved in 0b08946c6d988330598e2ae0817ad47178883b09 => {
            validate_vault_sResolved in 0b08946c6d988330598e2ae0817ad47178883b09()
        },
        Other => {
            if (tx.inputs.any((input: TxInput) -> {input.output_id
                == SEED_ID})) {
                validate_initialization()
            } else if (Tokens::get_minted()
                .any_key(TokenNames::has_assets_prefix)) {
                validate_mint_or_burn_asset_groups()
            } else {
                validate_mint_or_burn_dvp_tokens_vouchers_or_reimbursement()
            }
        }
    }
}
```

The Fund Policy relies on script context for determining the exact action performed instead of obtaining it from redeemer. Actions must be checked for in the descending order of their frequency to reduce non-value adding computations. Checking for inputs containing the Seed input would always return false after the initialization has happened.

Recommendation

The ordering can be modified as below to check for the most frequent action first:

Listing 15: src/fund_policy.hl

```
    func main(args: MixedArgs) -> Bool {
        args.switch{
            SResolved in 0b08946c6d988330598e2ae0817ad47178883b09 => {
                validate_vault_sResolved in 0b08946c6d988330598e2ae0817ad47178883b0
            },
            Other => {
                const minted_tokens = Tokens::get_minted();
                if (minted_tokens.contains_dvp_tokens()){
                    witnessed_by_supply()
```

```
                } else if (minted_tokens
                    .any_key(TokenNames::has_assets_prefix)) {
                    witnessed_by_portfolio()
                } else if (witnessed_by_reimbursement()) {
                    true
                } else {
                    tx.inputs.any((input: TxInput)
                        -> {input.output_id == SEED_ID})
                    && validate_initialization()
                }
            }
        }
    }
```

Resolution

Resolved in 0b08946c6d988330598e2ae0817ad47178883b09

ID-103 Inefficient Reduction - Exists

Level	Severity	Status
1	Informational	Resolved

Description

Portfolio reduction mode `Exists` helps in creating an on-chain proof of existence of a particular asset class in the portfolio. It expects to iterate over all the asset groups even if the required asset is already found leading to unnecessary on-chain computations.

Recommendation

Allow completion of portfolio reduction mode `Exists` if an asset class is found in the current transaction by expecting `group_iter` of reduction be equal to `portfolio.n_groups`.

Resolution

Resolved in 0b08946c6d988330598e2ae0817ad47178883b09

ID-104 Optimize Function diff_counted

Level	Severity	Status
1	Informational	Resolved

Description

Listing 16: src/lib/Vault.hl

```
func diff_counted(d_lovelace: Int, expected_tick: Int) -> Value {
    m: Map[AssetClass]Int =
      tx.inputs.fold((m: Map[AssetClass]Int, input: TxInput) -> {

        ...
            id = TokenNames::parse_assets(token_name).unwrap();
            group_asset_class = Tokens::assets(id);

            output = tx.outputs.find((output: TxOutput) -> {
                output.address == Addresses::assets
                && output.value.get_safe(group_asset_class) == 1
            });

        ...
}
```

`diff_counted` helps in checking `Vault::counters_are_consistent`, used by Supply Validator. The process of finding the output with the required assets group id can be optimized by providing output indices as a redeemer. Currently, all the outputs are iterated by checking individual address and value which is quite expensive, especially if large number of orders are being processed using many asset groups.

Recommendation

The Supply Validator redeemer can be updated to contain `[]AssetPtr` and asset group output indices (`[]Int`). The order of output indices must match the order of asset group inputs. For e.g. If first asset group input has group id 2, then its corresponding output's index must be the first element of the ouput indices array.

Resolution

Resolved in 0b08946c6d988330598e2ae0817ad47178883b09

ID-105 Redundant Mint Check

	Level	Severity	Status
	1	Informational	Resolved

Description

`validate_add_asset_class` and `validate_remove_asset_class` check for `Tokens::nothing_minted()` which is redundant.

Recommendation

To remove the check.

Resolution

Resolved in 0b08946c6d988330598e2ae0817ad47178883b09

ID-106 Redundant Output Asset Iteration

Level	Severity	Status
1	Informational	Resolved

Description

Ensuring that every asset present in the `AssetGroup` inputs is present in the output and `n_assets_in_inputs == n_assets_in_outputs` is sufficient. Checking for every asset in the `AssetGroup` outputs present in the input is redundant and expensive.

Listing 17: src/portfolio_validator.hl

```
    n_assets_in_outputs = tx.outputs.fold((n_assets: Int, output: TxOutput)
      -> {
        if (output.address == Addresses::assets) {
            assert(Tokens::contains_only_any_assets(output),
                "doesn't contain only 1 assets token");

            group = output.datum.inline.as[AssetGroup];
            assert(group.is_not_overfull(),
                "output assetgroup is overfull");

            // redundant check
            group.assets.fold((n_assets: Int, asset1: Asset) -> {
                asset0 = AssetGroup::find_input_asset(asset1.asset_class);

                assert(asset0 == asset1, "asset can't change");

                n_assets + 1
            }, n_assets)
        } else {
            n_assets
        }
    }, 0);
```

Recommendation

To remove the redundant check.

Resolution

Resolved in 0b08946c6d988330598e2ae0817ad47178883b09

ID-107 Duplicated Voucher Check

Level	Severity	Status
1	Informational	Resolved

Description

The check on `voucher.price` and `voucher.period_id` is performed twice, in Mint Order Validator and by Supply Validator in `validate_minted_vouchers`.

Recommendation

It is sufficient to have the check carried out only once.

Resolution

Resolved in 0b08946c6d988330598e2ae0817ad47178883b09

ID-108 Unnecessary Output Traversal

Level	Severity	Status
1	Informational	Resolved

Description

All the outputs are traversed by `validate_minted_vouchers` to ensure that voucher ids are created in an incremental order. However, the traversal is expected even if no vouchers are minted (when current_price < year_start_price) which is unnecessary. It is also expensive because there can be many outputs, a consequence of numerous mint orders being processed.

Listing 18: src/lib/Voucher.hl

```
    func validate_minted_vouchers(price: Ratio, period_id: Int,
        last_voucher_id: Int) -> (Int, Int) {
          ...

        last_voucher_id = tx.outputs.fold((prev_id: Int, output: TxOutput)
        -> {
            if (output.address == Addresses::voucher) {
                id = prev_id + 1;

                assert(Tokens::contains_only_voucher_ref(output, id),
                "voucher doesn't have expected id");

                id
            } else {
                prev_id
            }
        }, last_voucher_id);

        (n_vouchers_minted, last_voucher_id)
    }
```

Recommendation

Avoid output traversal when `n_vouchers_minted == 0`.

Resolution

Resolved in 0b08946c6d988330598e2ae0817ad47178883b09